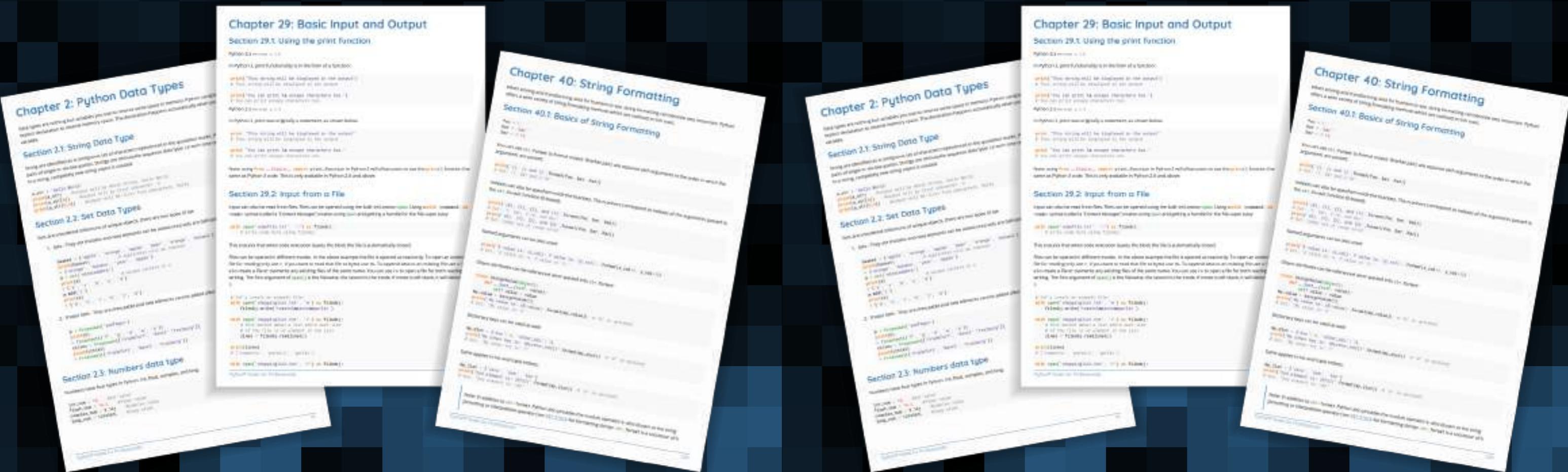


# Python® 专业人士笔记

# Python® Notes for Professionals



800+ 页  
专业提示和技巧

800+ pages  
of professional hints and tricks

# 目录

<a href="#">关于</a>	1
<b>第1章：Python语言入门</b>	2
<a href="#">第1.1节：入门</a>	2
<a href="#">第1.2节：创建变量并赋值</a>	6
<a href="#">第1.3节：块缩进</a>	10
<a href="#">第1.4节：数据类型</a>	11
<a href="#">第1.5节：集合类型</a>	15
<a href="#">第1.6节：IDLE - Python图形用户界面</a>	19
<a href="#">第1.7节：用户输入</a>	21
<a href="#">第1.8节：内置模块和函数</a>	21
<a href="#">第1.9节：创建模块</a>	25
<a href="#">第1.10节：安装Python 2.7.x和3.x</a>	26
<a href="#">第1.11节：字符串函数 - str() 和 repr()</a>	28
<a href="#">第1.12节：使用 pip 安装外部模块</a>	29
<a href="#">第1.13节：帮助工具</a>	31
<b>第2章：Python数据类型</b>	33
<a href="#">第2.1节：字符串数据类型</a>	33
<a href="#">第2.2节：集合数据类型</a>	33
<a href="#">第2.3节：数字数据类型</a>	33
<a href="#">第2.4节：列表数据类型</a>	34
<a href="#">第2.5节：字典数据类型</a>	34
<a href="#">第2.6节：元组数据类型</a>	34
<b>第3章：缩进</b>	35
<a href="#">第3.1节：简单示例</a>	35
<a href="#">第3.2节：缩进的解析方式</a>	35
<a href="#">第3.3节：缩进错误</a>	36
<b>第4章：注释与文档</b>	37
<a href="#">第4.1节：单行、内联和多行注释</a>	37
<a href="#">第4.2节：以编程方式访问文档字符串</a>	37
<a href="#">第4.3节：使用文档字符串编写文档</a>	38
<b>第5章：日期和时间</b>	42
<a href="#">第5.1节：将字符串解析为带时区的日期时间对象</a>	42
<a href="#">第5.2节：构造带时区的日期时间</a>	42
<a href="#">第5.3节：计算时间差</a>	44
<a href="#">第5.4节：基本日期时间对象的使用</a>	44
<a href="#">第5.5节：时区切换</a>	45
<a href="#">第5.6节：简单日期运算</a>	45
<a href="#">第5.7节：时间戳转换为日期时间</a>	46
<a href="#">第5.8节：准确地从日期中减去月份</a>	46
<a href="#">第5.9节：使用最少的库解析任意ISO 8601时间戳</a>	46
<a href="#">第5.10节：获取ISO 8601时间戳</a>	47
<a href="#">第5.11节：将带有简短时区名称的字符串解析为带时区信息的日期时间对象</a>	47
<a href="#">第5.12节：模糊日期时间解析（从文本中提取日期时间）</a>	48
<a href="#">第5.13节：遍历日期</a>	49
<b>第6章：日期格式化</b>	50
<a href="#">第6.1节：两个日期时间之间的时间</a>	50
<a href="#">第6.2节：将日期时间对象输出为字符串</a>	50

# Contents

<a href="#">About</a>	1
<b>Chapter 1: Getting started with Python Language</b>	2
<a href="#">Section 1.1: Getting Started</a>	2
<a href="#">Section 1.2: Creating variables and assigning values</a>	6
<a href="#">Section 1.3: Block Indentation</a>	10
<a href="#">Section 1.4: Datatypes</a>	11
<a href="#">Section 1.5: Collection Types</a>	15
<a href="#">Section 1.6: IDLE - Python GUI</a>	19
<a href="#">Section 1.7: User Input</a>	21
<a href="#">Section 1.8: Built in Modules and Functions</a>	21
<a href="#">Section 1.9: Creating a module</a>	25
<a href="#">Section 1.10: Installation of Python 2.7.x and 3.x</a>	26
<a href="#">Section 1.11: String function - str() and repr()</a>	28
<a href="#">Section 1.12: Installing external modules using pip</a>	29
<a href="#">Section 1.13: Help Utility</a>	31
<b>Chapter 2: Python Data Types</b>	33
<a href="#">Section 2.1: String Data Type</a>	33
<a href="#">Section 2.2: Set Data Types</a>	33
<a href="#">Section 2.3: Numbers data type</a>	33
<a href="#">Section 2.4: List Data Type</a>	34
<a href="#">Section 2.5: Dictionary Data Type</a>	34
<a href="#">Section 2.6: Tuple Data Type</a>	34
<b>Chapter 3: Indentation</b>	35
<a href="#">Section 3.1: Simple example</a>	35
<a href="#">Section 3.2: How Indentation is Parsed</a>	35
<a href="#">Section 3.3: Indentation Errors</a>	36
<b>Chapter 4: Comments and Documentation</b>	37
<a href="#">Section 4.1: Single line, inline and multiline comments</a>	37
<a href="#">Section 4.2: Programmatically accessing docstrings</a>	37
<a href="#">Section 4.3: Write documentation using docstrings</a>	38
<b>Chapter 5: Date and Time</b>	42
<a href="#">Section 5.1: Parsing a string into a timezone aware datetime object</a>	42
<a href="#">Section 5.2: Constructing timezone-aware datetimes</a>	42
<a href="#">Section 5.3: Computing time differences</a>	44
<a href="#">Section 5.4: Basic datetime objects usage</a>	44
<a href="#">Section 5.5: Switching between time zones</a>	45
<a href="#">Section 5.6: Simple date arithmetic</a>	45
<a href="#">Section 5.7: Converting timestamp to datetime</a>	46
<a href="#">Section 5.8: Subtracting months from a date accurately</a>	46
<a href="#">Section 5.9: Parsing an arbitrary ISO 8601 timestamp with minimal libraries</a>	46
<a href="#">Section 5.10: Get an ISO 8601 timestamp</a>	47
<a href="#">Section 5.11: Parsing a string with a short time zone name into a timezone aware datetime object</a>	47
<a href="#">Section 5.12: Fuzzy datetime parsing (extracting datetime out of a text)</a>	48
<a href="#">Section 5.13: Iterate over dates</a>	49
<b>Chapter 6: Date Formatting</b>	50
<a href="#">Section 6.1: Time between two date-times</a>	50
<a href="#">Section 6.2: Outputting datetime object to string</a>	50

第6.3节：将字符串解析为日期时间对象	50
<b>第7章：枚举</b>	51
第7.1节：创建枚举（Python 2.4至3.3）	51
第7.2节：迭代	51
<b>第8章：集合</b>	52
第8.1节：集合的运算	52
第8.2节：获取列表中的唯一元素	53
第8.3节：集合的集合	53
第8.4节：使用方法和内置函数的集合操作	53
第8.5节：集合与多重集合	55
<b>第9章：简单的数学运算符</b>	57
第9.1节：除法	57
第9.2节：加法	58
第9.3节：指数运算	59
第9.4节：三角函数	60
第9.5节：原地操作	61
第9.6节：减法	61
第9.7节：乘法	61
第9.8节：对数	62
第9.9节：模运算	62
<b>第10章：按位运算符</b>	65
第10.1节：按位取反	65
第10.2节：按位异或（排他或）	66
第10.3节：按位与	67
第10.4节：按位或	67
第10.5节：按位左移	67
第10.6节：按位右移	68
第10.7节：就地操作	68
<b>第11章：布尔运算符</b>	69
第11.1节：`and` 和 `or` 不保证返回布尔值	69
第11.2节：一个简单的例子	69
第11.3节：短路求值	69
第11.4节：and	70
第11.5节：或	70
第11.6节：not	71
<b>第12章：运算符优先级</b>	72
第12.1节：Python中的简单运算符优先级示例	72
<b>第13章：变量作用域与绑定</b>	73
第13.1节：非局部变量	73
第13.2节：全局变量	73
第13.3节：局部变量	74
第13.4节：del命令	75
第13.5节：函数在查找名称时跳过类作用域	76
第13.6节：局部作用域与全局作用域	77
第13.7节：绑定发生	79
<b>第14章：条件语句</b>	80
第14.1节：条件表达式（或“三元运算符”）	80
第14.2节：if、elif 和 else	80
第14.3节：真值	80

Section 6.3: Parsing string to datetime object	50
<b>Chapter 7: Enum</b>	51
Section 7.1: Creating an enum (Python 2.4 through 3.3)	51
Section 7.2: Iteration	51
<b>Chapter 8: Set</b>	52
Section 8.1: Operations on sets	52
Section 8.2: Get the unique elements of a list	53
Section 8.3: Set of Sets	53
Section 8.4: Set Operations using Methods and Builtins	53
Section 8.5: Sets versus multisets	55
<b>Chapter 9: Simple Mathematical Operators</b>	57
Section 9.1: Division	57
Section 9.2: Addition	58
Section 9.3: Exponentiation	59
Section 9.4: Trigonometric Functions	60
Section 9.5: Inplace Operations	61
Section 9.6: Subtraction	61
Section 9.7: Multiplication	61
Section 9.8: Logarithms	62
Section 9.9: Modulus	62
<b>Chapter 10: Bitwise Operators</b>	65
Section 10.1: Bitwise NOT	65
Section 10.2: Bitwise XOR (Exclusive OR)	66
Section 10.3: Bitwise AND	67
Section 10.4: Bitwise OR	67
Section 10.5: Bitwise Left Shift	67
Section 10.6: Bitwise Right Shift	68
Section 10.7: Inplace Operations	68
<b>Chapter 11: Boolean Operators</b>	69
Section 11.1: `and` and `or` are not guaranteed to return a boolean	69
Section 11.2: A simple example	69
Section 11.3: Short-circuit evaluation	69
Section 11.4: and	70
Section 11.5: or	70
Section 11.6: not	71
<b>Chapter 12: Operator Precedence</b>	72
Section 12.1: Simple Operator Precedence Examples in python	72
<b>Chapter 13: Variable Scope and Binding</b>	73
Section 13.1: Nonlocal Variables	73
Section 13.2: Global Variables	73
Section 13.3: Local Variables	74
Section 13.4: The del command	75
Section 13.5: Functions skip class scope when looking up names	76
Section 13.6: Local vs Global Scope	77
Section 13.7: Binding Occurrence	79
<b>Chapter 14: Conditionals</b>	80
Section 14.1: Conditional Expression (or "The Ternary Operator")	80
Section 14.2: if, elif, and else	80
Section 14.3: Truth Values	80

<a href="#">第14章：布尔逻辑表达式</a>	81	<a href="#">Section 14.4: Boolean Logic Expressions</a>	81
<a href="#">第14.5节：使用cmp函数获取两个对象的比较结果</a>	83	<a href="#">Section 14.5: Using the cmp function to get the comparison result of two objects</a>	83
<a href="#">第14.6节：Else语句</a>	83	<a href="#">Section 14.6: Else statement</a>	83
<a href="#">第14.7节：测试对象是否为None并赋值</a>	84	<a href="#">Section 14.7: Testing if an object is None and assigning it</a>	84
<a href="#">第14.8节：If语句</a>	84	<a href="#">Section 14.8: If statement</a>	84
<b>第15章：比较</b>	86	<b>Chapter 15: Comparisons</b>	86
<a href="#">第15.1节：链式比较</a>	86	<a href="#">Section 15.1: Chain Comparisons</a>	86
<a href="#">第15.2节：使用`is`与`==`进行比较</a>	87	<a href="#">Section 15.2: Comparison by `is` vs `==`</a>	87
<a href="#">第15.3节：大于或小于</a>	88	<a href="#">Section 15.3: Greater than or less than</a>	88
<a href="#">第15.4节：不等于</a>	88	<a href="#">Section 15.4: Not equal to</a>	88
<a href="#">第15.5节：等于</a>	89	<a href="#">Section 15.5: Equal To</a>	89
<a href="#">第15.6节：对象比较</a>	89	<a href="#">Section 15.6: Comparing Objects</a>	89
<b>第16章：循环</b>	91	<b>Chapter 16: Loops</b>	91
<a href="#">第16.1节：循环中的break和continue</a>	91	<a href="#">Section 16.1: Break and Continue in Loops</a>	91
<a href="#">第16.2节：for循环</a>	93	<a href="#">Section 16.2: For loops</a>	93
<a href="#">第16.3节：遍历列表</a>	93	<a href="#">Section 16.3: Iterating over lists</a>	93
<a href="#">第16.4节：带有“else”子句的循环</a>	94	<a href="#">Section 16.4: Loops with an "else" clause</a>	94
<a href="#">第16.5节：pass语句</a>	96	<a href="#">Section 16.5: The Pass Statement</a>	96
<a href="#">第16.6节：遍历字典</a>	97	<a href="#">Section 16.6: Iterating over dictionaries</a>	97
<a href="#">第16.7节：“半循环”do-while</a>	98	<a href="#">Section 16.7: The "half loop" do-while</a>	98
<a href="#">第16.8节：循环与解包</a>	98	<a href="#">Section 16.8: Looping and Unpacking</a>	98
<a href="#">第16.9节：以不同步长遍历列表的不同部分</a>	99	<a href="#">Section 16.9: Iterating different portion of a list with different step size</a>	99
<a href="#">第16.10节：While循环</a>	100	<a href="#">Section 16.10: While Loop</a>	100
<b>第17章：数组</b>	102	<b>Chapter 17: Arrays</b>	102
<a href="#">第17.1节：通过索引访问单个元素</a>	102	<a href="#">Section 17.1: Access individual elements through indexes</a>	102
<a href="#">第17.2节：数组基础介绍</a>	102	<a href="#">Section 17.2: Basic Introduction to Arrays</a>	102
<a href="#">第17.3节：使用append()方法向数组追加任意值</a>	103	<a href="#">Section 17.3: Append any value to the array using append() method</a>	103
<a href="#">第17.4节：使用insert()方法在数组中插入值</a>	103	<a href="#">Section 17.4: Insert value in an array using insert() method</a>	103
<a href="#">第17.5节：使用extend()方法扩展Python数组</a>	103	<a href="#">Section 17.5: Extend python array using extend() method</a>	103
<a href="#">第17.6节：使用fromlist()方法将列表中的项目添加到数组中</a>	104	<a href="#">Section 17.6: Add items from list into array using fromlist() method</a>	104
<a href="#">第17.7节：使用remove()方法删除任意数组元素</a>	104	<a href="#">Section 17.7: Remove any array element using remove() method</a>	104
<a href="#">第17.8节：使用pop()方法删除数组的最后一个元素</a>	104	<a href="#">Section 17.8: Remove last array element using pop() method</a>	104
<a href="#">第17.9节：使用index()方法通过索引获取任意元素</a>	104	<a href="#">Section 17.9: Fetch any element through its index using index() method</a>	104
<a href="#">第17.10节：使用reverse()方法反转Python数组</a>	104	<a href="#">Section 17.10: Reverse a python array using reverse() method</a>	104
<a href="#">第17.11节：通过buffer_info()方法获取数组的缓冲区信息</a>	105	<a href="#">Section 17.11: Get array buffer information through buffer_info() method</a>	105
<a href="#">第17.12节：使用count()方法检查元素的出现次数</a>	105	<a href="#">Section 17.12: Check for number of occurrences of an element using count() method</a>	105
<a href="#">第17.13节：使用tostring()方法将数组转换为字符串</a>	105	<a href="#">Section 17.13: Convert array to string using tostring() method</a>	105
<a href="#">第17.14节：使用tolist()方法将数组转换为具有相同元素的Python列表</a>	105	<a href="#">Section 17.14: Convert array to a python list with same elements using tolist() method</a>	105
<a href="#">第17.15节：使用fromstring()方法向字符数组追加字符串</a>	105	<a href="#">Section 17.15: Append a string to char array using fromstring() method</a>	105
<b>第18章：多维数组</b>	106	<b>Chapter 18: Multidimensional arrays</b>	106
<a href="#">第18.1节：列表中的列表</a>	106	<a href="#">Section 18.1: Lists in lists</a>	106
<a href="#">第18.2节：列表中的列表中的列表</a>	106	<a href="#">Section 18.2: Lists in lists in lists</a>	106
<b>第19章：字典</b>	108	<b>Chapter 19: Dictionary</b>	108
<a href="#">第19.1节：字典简介</a>	108	<a href="#">Section 19.1: Introduction to Dictionary</a>	108
<a href="#">第19.2节：避免KeyError异常</a>	109	<a href="#">Section 19.2: Avoiding KeyError Exceptions</a>	109
<a href="#">第19.3节：遍历字典</a>	109	<a href="#">Section 19.3: Iterating Over a Dictionary</a>	109
<a href="#">第19.4节：带默认值的字典</a>	110	<a href="#">Section 19.4: Dictionary with default values</a>	110
<a href="#">第19.5节：合并字典</a>	111	<a href="#">Section 19.5: Merging dictionaries</a>	111
<a href="#">第19.6节：访问键和值</a>	111	<a href="#">Section 19.6: Accessing keys and values</a>	111
<a href="#">第19.7节：访问字典的值</a>	112	<a href="#">Section 19.7: Accessing values of a dictionary</a>	112

第19.8节：创建字典	112
第19.9节：创建有序字典	113
第19.10节：使用**运算符解包字典	113
第19.11节：尾随逗号	114
第19.12节：dict()构造函数	114
第19.13节：字典示例	114
第19.14节：字典值的所有组合	115
<b>第20章：列表</b>	117
第20.1节：列表方法和支持的操作符	117
第20.2节：访问列表值	122
第20.3节：检查列表是否为空	123
第20.4节：遍历列表	123
第20.5节：检查某项是否在列表中	124
第20.6节：任意与全部	124
第20.7节：反转列表元素	125
第20.8节：连接与合并列表	125
第20.9节：列表长度	126
第20.10节：移除列表中的重复值	126
第20.11节：列表比较	127
第20.12节：访问嵌套列表中的值	127
第20.13节：初始化固定元素数量的列表	128
<b>第21章：列表推导式</b>	130
第21.1节：列表推导式	130
第21.2节：条件列表推导式	132
第21.3节：使用条件子句避免重复且昂贵的操作	134
第21.4节：字典推导式	135
第21.5节：带嵌套循环的列表推导式	136
第21.6节：生成器表达式	138
第21.7节：集合推导式	140
第21.8节：将filter和map重构为列表推导式	140
第21.9节：涉及元组的推导式	141
第21.10节：使用推导式计数出现次数	142
第21.11节：列表中类型的转换	142
第21.12节：嵌套列表推导式	142
第21.13节：在列表推导式中同时迭代两个或多个列表	143
<b>第22章：列表切片（选择列表的部分内容）</b>	144
第22.1节：使用第三个“步长”参数	144
第22.2节：从列表中选择子列表	144
第22.3节：使用切片反转列表	144
第22.4节：使用切片移动列表	144
<b>第23章：groupby()</b>	146
第23.1节：示例4	146
第23.2节：示例2	146
第23.3节：示例3	147
<b>第24章：链表</b>	149
第24.1节：单链表示例	149
<b>第25章：链表节点</b>	154
第25.1节：用Python编写一个简单的链表节点	154
<b>第26章：滤波器</b>	155

Section 19.8: Creating a dictionary	112
Section 19.9: Creating an ordered dictionary	113
Section 19.10: Unpacking dictionaries using the ** operator	113
Section 19.11: The trailing comma	114
Section 19.12: The dict() constructor	114
Section 19.13: Dictionaries Example	114
Section 19.14: All combinations of dictionary values	115
<b>Chapter 20: List</b>	117
Section 20.1: List methods and supported operators	117
Section 20.2: Accessing list values	122
Section 20.3: Checking if list is empty	123
Section 20.4: Iterating over a list	123
Section 20.5: Checking whether an item is in a list	124
Section 20.6: Any and All	124
Section 20.7: Reversing list elements	125
Section 20.8: Concatenate and Merge lists	125
Section 20.9: Length of a list	126
Section 20.10: Remove duplicate values in list	126
Section 20.11: Comparison of lists	127
Section 20.12: Accessing values in nested list	127
Section 20.13: Initializing a List to a Fixed Number of Elements	128
<b>Chapter 21: List comprehensions</b>	130
Section 21.1: List Comprehensions	130
Section 21.2: Conditional List Comprehensions	132
Section 21.3: Avoid repetitive and expensive operations using conditional clause	134
Section 21.4: Dictionary Comprehensions	135
Section 21.5: List Comprehensions with Nested Loops	136
Section 21.6: Generator Expressions	138
Section 21.7: Set Comprehensions	140
Section 21.8: Refactoring filter and map to list comprehensions	140
Section 21.9: Comprehensions involving tuples	141
Section 21.10: Counting Occurrences Using Comprehension	142
Section 21.11: Changing Types in a List	142
Section 21.12: Nested List Comprehensions	142
Section 21.13: Iterate two or more list simultaneously within list comprehension	143
<b>Chapter 22: List slicing (selecting parts of lists)</b>	144
Section 22.1: Using the third "step" argument	144
Section 22.2: Selecting a sublist from a list	144
Section 22.3: Reversing a list with slicing	144
Section 22.4: Shifting a list using slicing	144
<b>Chapter 23: groupby()</b>	146
Section 23.1: Example 4	146
Section 23.2: Example 2	146
Section 23.3: Example 3	147
<b>Chapter 24: Linked lists</b>	149
Section 24.1: Single linked list example	149
<b>Chapter 25: Linked List Node</b>	154
Section 25.1: Write a simple Linked List Node in python	154
<b>Chapter 26: Filter</b>	155

第26.1节：过滤器的基本使用	155
第26.2节：无函数的过滤器	155
第26.3节：作为短路检查的过滤器	156
第26.4节：补充函数：filterfalse, ifilterfalse	156
<b>第27章：堆队列（Heapq）</b>	158
第27.1节：集合中的最大和最小项	158
第27.2节：集合中的最小项	158
<b>第28章：元组</b>	160
第28.1节：元组	160
第28.2节：元组是不可变的	161
第28.3节：元组的打包与解包	161
第28.4节：内置元组函数	162
第28.5节：元组是元素级可哈希和可比较的	163
第28.6节：元组索引	164
第28.7节：元素反转	164
<b>第29章：基本输入输出</b>	165
第29.1节：使用print函数	165
第29.2节：从文件输入	165
第29.3节：从标准输入读取	167
第29.4节：使用input()和raw_input()	167
第29.5节：提示用户输入数字的函数	167
第29.6节：打印字符串时不换行	168
<b>第30章：文件与文件夹输入输出</b>	171
第30.1节：文件模式	171
第30.2节：逐行读取文件	172
第30.3节：递归遍历文件	173
第30.4节：获取文件的全部内容	173
第30.5节：写入文件	174
第30.6节：检查文件或路径是否存在	175
第30.7节：使用mmap进行随机文件访问	176
第30.8节：替换文件中的文本	176
第30.9节：检查文件是否为空	176
第30.10节：读取指定行范围内的文件	177
第30.11节：复制目录树	177
第30.12节：将一个文件的内容复制到另一个文件	177
<b>第31章：os.path</b>	178
第31.1节：路径拼接	178
第31.2节：路径组件操作	178
第31.3节：获取父目录	178
第31.4节：如果给定路径存在	178
第31.5节：检查给定路径是否为目录、文件、符号链接、挂载点等	179
第31.6节：从相对路径获取绝对路径	179
<b>第32章：可迭代对象和迭代器</b>	180
第32.1节：迭代器与可迭代对象与生成器	180
第32.2节：逐个提取值	181
第32.3节：遍历整个可迭代对象	181
第32.4节：验证可迭代对象中只有一个元素	181
第32.5节：什么可以是可迭代对象	182
第32.6节：迭代器不是可重入的！	182

Section 26.1: Basic use of filter	155
Section 26.2: Filter without function	155
Section 26.3: Filter as short-circuit check	156
Section 26.4: Complementary function: filterfalse, ifilterfalse	156
<b>Chapter 27: Heapq</b>	158
Section 27.1: Largest and smallest items in a collection	158
Section 27.2: Smallest item in a collection	158
<b>Chapter 28: Tuple</b>	160
Section 28.1: Tuple	160
Section 28.2: Tuples are immutable	161
Section 28.3: Packing and Unpacking Tuples	161
Section 28.4: Built-in Tuple Functions	162
Section 28.5: Tuple Are Element-wise Hashable and Equatable	163
Section 28.6: Indexing Tuples	164
Section 28.7: Reversing Elements	164
<b>Chapter 29: Basic Input and Output</b>	165
Section 29.1: Using the print function	165
Section 29.2: Input from a File	165
Section 29.3: Read from stdin	167
Section 29.4: Using input() and raw_input()	167
Section 29.5: Function to prompt user for a number	167
Section 29.6: Printing a string without a newline at the end	168
<b>Chapter 30: Files &amp; Folders I/O</b>	171
Section 30.1: File modes	171
Section 30.2: Reading a file line-by-line	172
Section 30.3: Iterate files (recursively)	173
Section 30.4: Getting the full contents of a file	173
Section 30.5: Writing to a file	174
Section 30.6: Check whether a file or path exists	175
Section 30.7: Random File Access Using mmap	176
Section 30.8: Replacing text in a file	176
Section 30.9: Checking if a file is empty	176
Section 30.10: Read a file between a range of lines	177
Section 30.11: Copy a directory tree	177
Section 30.12: Copying contents of one file to a different file	177
<b>Chapter 31: os.path</b>	178
Section 31.1: Join Paths	178
Section 31.2: Path Component Manipulation	178
Section 31.3: Get the parent directory	178
Section 31.4: If the given path exists	178
Section 31.5: check if the given path is a directory, file, symbolic link, mount point etc	179
Section 31.6: Absolute Path from Relative Path	179
<b>Chapter 32: Iterables and Iterators</b>	180
Section 32.1: Iterator vs Iterable vs Generator	180
Section 32.2: Extract values one by one	181
Section 32.3: Iterating over entire iterable	181
Section 32.4: Verify only one element in iterable	181
Section 32.5: What can be iterable	182
Section 32.6: Iterator isn't reentrant!	182

<b>第33章：函数</b>	183
第33.1节：定义和调用简单函数	183
第33.2节：定义带任意数量参数的函数	184
第33.3节：Lambda（内联/匿名）函数	187
第33.4节：定义带有可选参数的函数	189
第33.5节：定义带有可变可选参数的函数	190
第33.6节：参数传递与可变性	191
第33.7节：从函数返回值	192
第33.8节：闭包	192
第33.9节：强制使用命名参数	193
第33.10节：嵌套函数	194
第33.11节：递归限制	194
第33.12节：使用赋值变量的递归Lambda	195
第33.13节：递归函数	195
第33.14节：定义带参数的函数	196
第33.15节：可迭代对象和字典拆包	196
第33.16节：定义带多个参数的函数	198
<b>第34章：定义带列表参数的函数</b>	199
第34.1节：函数与调用	199
<b>第35章：Python中的函数式编程</b>	201
第35.1节：Lambda函数	201
第35.2节：Map函数	201
第35.3节：Reduce函数	201
第35.4节：Filter函数	201
<b>第36章：偏函数</b>	202
第36.1节：幂运算	202
<b>第37章：装饰器</b>	203
第37.1节：装饰器函数	203
第37.2节：装饰器类	204
第37.3节：带参数的装饰器（装饰器工厂）	205
第37.4节：让装饰器看起来像被装饰的函数	207
第37.5节：使用装饰器计时函数	207
第37.6节：用装饰器创建单例类	208
<b>第38章：类</b>	209
第38.1节：类简介	209
第38.2节：绑定方法、非绑定方法和静态方法	210
第38.3节：基本继承	212
第38.4节：猴子补丁（Monkey Patching）	214
第38.5节：新式类与旧式类	214
第38.6节：类方法：替代初始化方法	215
第38.7节：多重继承	217
第38.8节：属性	219
第38.9节：实例变量的默认值	220
第38.10节：类变量和实例变量	221
第38.11节：类的组成	222
第38.12节：列出所有类成员	223
第38.13节：单例类	224
第38.14节：描述符和点查找	225
<b>第39章：元类</b>	226
<b>Chapter 33: Functions</b>	183
Section 33.1: Defining and calling simple functions	183
Section 33.2: Defining a function with an arbitrary number of arguments	184
Section 33.3: Lambda (Inline/Anonymous) Functions	187
Section 33.4: Defining a function with optional arguments	189
Section 33.5: Defining a function with optional mutable arguments	190
Section 33.6: Argument passing and mutability	191
Section 33.7: Returning values from functions	192
Section 33.8: Closure	192
Section 33.9: Forcing the use of named parameters	193
Section 33.10: Nested functions	194
Section 33.11: Recursion limit	194
Section 33.12: Recursive Lambda using assigned variable	195
Section 33.13: Recursive functions	195
Section 33.14: Defining a function with arguments	196
Section 33.15: Iterable and dictionary unpacking	196
Section 33.16: Defining a function with multiple arguments	198
<b>Chapter 34: Defining functions with list arguments</b>	199
Section 34.1: Function and Call	199
<b>Chapter 35: Functional Programming in Python</b>	201
Section 35.1: Lambda Function	201
Section 35.2: Map Function	201
Section 35.3: Reduce Function	201
Section 35.4: Filter Function	201
<b>Chapter 36: Partial functions</b>	202
Section 36.1: Raise the power	202
<b>Chapter 37: Decorators</b>	203
Section 37.1:Decorator function	203
Section 37.2:Decorator class	204
Section 37.3:Decorator with arguments (decorator factory)	205
Section 37.4:Making a decorator look like the decorated function	207
Section 37.5:Using a decorator to time a function	207
Section 37.6:Create singleton class with a decorator	208
<b>Chapter 38: Classes</b>	209
Section 38.1:Introduction to classes	209
Section 38.2:Bound, unbound, and static methods	210
Section 38.3:Basic inheritance	212
Section 38.4:Monkey Patching	214
Section 38.5:New-style vs. old-style classes	214
Section 38.6:Class methods: alternate initializers	215
Section 38.7:Multiple Inheritance	217
Section 38.8:Properties	219
Section 38.9:Default values for instance variables	220
Section 38.10:Class and instance variables	221
Section 38.11:Class composition	222
Section 38.12:Listing All Class Members	223
Section 38.13:Singleton class	224
Section 38.14:Descriptors and Dotted Lookups	225
<b>Chapter 39: Metaclasses</b>	226

第39.1节：基本元类	226	<a href="#">Section 39.1: Basic Metaclasses</a>	226
第39.2节：使用元类的单例	227	<a href="#">Section 39.2: Singletons using metaclasses</a>	227
第39.3节：使用元类	227	<a href="#">Section 39.3: Using a metaclass</a>	227
第39.4节：元类简介	227	<a href="#">Section 39.4: Introduction to Metaclasses</a>	227
第39.5节：使用元类实现自定义功能	228	<a href="#">Section 39.5: Custom functionality with metaclasses</a>	228
第39.6节：默认元类	229	<a href="#">Section 39.6: The default metaclass</a>	229
<b>第40章：字符串格式化</b>	232	<b>Chapter 40: String Formatting</b>	232
第40.1节：字符串格式化基础	232	<a href="#">Section 40.1: Basics of String Formatting</a>	232
第40.2节：对齐与填充	233	<a href="#">Section 40.2: Alignment and padding</a>	233
第40.3节：格式字面量（f字符串）	234	<a href="#">Section 40.3: Format literals (f-string)</a>	234
第40.4节：浮点数格式化	234	<a href="#">Section 40.4: Float formatting</a>	234
第40.5节：命名占位符	235	<a href="#">Section 40.5: Named placeholders</a>	235
第40.6节：使用日期时间的字符串格式化	236	<a href="#">Section 40.6: String formatting with datetime</a>	236
第40.7节：数值格式化	236	<a href="#">Section 40.7: Formatting Numerical Values</a>	236
第40.8节：嵌套格式化	237	<a href="#">Section 40.8: Nested formatting</a>	237
第40.9节：使用Getitem和getattr进行格式化	237	<a href="#">Section 40.9: Format using Getitem and getattr</a>	237
第40.10节：字符串的填充和截断，组合使用	237	<a href="#">Section 40.10: Padding and truncating strings, combined</a>	237
第40.11节：类的自定义格式化	238	<a href="#">Section 40.11: Custom formatting for a class</a>	238
<b>第41章：字符串方法</b>	240	<b>Chapter 41: String Methods</b>	240
第41.1节：改变字符串的大小写	240	<a href="#">Section 41.1: Changing the capitalization of a string</a>	240
第41.2节：str.translate：字符串中字符的转换	241	<a href="#">Section 41.2: str.translate: Translating characters in a string</a>	241
第41.3节：str.format和f字符串：将值格式化为字符串	242	<a href="#">Section 41.3: str.format and f-strings: Format values into a string</a>	242
第41.4节：字符串模块的有用常量	243	<a href="#">Section 41.4: String module's useful constants</a>	243
第41.5节：去除字符串中不需要的前导/尾随字符	244	<a href="#">Section 41.5: Stripping unwanted leading/trailing characters from a string</a>	244
第41.6节：字符串反转	245	<a href="#">Section 41.6: Reversing a string</a>	245
第41.7节：基于分隔符将字符串拆分为字符串列表	245	<a href="#">Section 41.7: Split a string based on a delimiter into a list of strings</a>	245
第41.8节：将所有出现的一个子字符串替换为另一个子字符串	246	<a href="#">Section 41.8: Replace all occurrences of one substring with another substring</a>	246
第41.9节：测试字符串的组成	247	<a href="#">Section 41.9: Testing what a string is composed of</a>	247
第41.10节：字符串包含	249	<a href="#">Section 41.10: String Contains</a>	249
第41.11节：将字符串列表连接成一个字符串	249	<a href="#">Section 41.11: Join a list of strings into one string</a>	249
第41.12节：计算子字符串在字符串中出现的次数	250	<a href="#">Section 41.12: Counting number of times a substring appears in a string</a>	250
第41.13节：不区分大小写的字符串比较	250	<a href="#">Section 41.13: Case insensitive string comparisons</a>	250
第41.14节：字符串对齐	251	<a href="#">Section 41.14: Justify strings</a>	251
第41.15节：测试字符串的起始和结束字符	252	<a href="#">Section 41.15: Test the starting and ending characters of a string</a>	252
第41.16节：str或bytes数据与Unicode字符之间的转换	253	<a href="#">Section 41.16: Conversion between str or bytes data and unicode characters</a>	253
<b>第42章：在函数中使用循环</b>	255	<b>Chapter 42: Using loops within functions</b>	255
第42.1节：函数中循环内的返回语句	255	<a href="#">Section 42.1: Return statement inside loop in a function</a>	255
<b>第43章：导入模块</b>	256	<b>Chapter 43: Importing modules</b>	256
第43.1节：导入模块	256	<a href="#">Section 43.1: Importing a module</a>	256
第43.2节：all特殊变量	257	<a href="#">Section 43.2: The all special variable</a>	257
第43.3节：从任意文件系统位置导入模块	258	<a href="#">Section 43.3: Import modules from an arbitrary filesystem location</a>	258
第43.4节：从模块导入所有名称	258	<a href="#">Section 43.4: Importing all names from a module</a>	258
第43.5节：编程式导入	259	<a href="#">Section 43.5: Programmatic importing</a>	259
第43.6节：导入的PEP8规则	259	<a href="#">Section 43.6: PEP8 rules for Imports</a>	259
第43.7节：从模块导入特定名称	260	<a href="#">Section 43.7: Importing specific names from a module</a>	260
第43.8节：导入子模块	260	<a href="#">Section 43.8: Importing submodules</a>	260
第43.9节：重新导入模块	260	<a href="#">Section 43.9: Re-importing a module</a>	260
第43.10节：import()函数	261	<a href="#">Section 43.10: import() function</a>	261
<b>第44章：模块与包的区别</b>	262	<b>Chapter 44: Difference between Module and Package</b>	262
第44.1节：模块	262	<a href="#">Section 44.1: Modules</a>	262

第44.2节：包	262
<b>第45章：数学模块</b>	264
第45.1节：四舍五入：round, floor, ceil, trunc	264
第45.2节：三角函数	265
第45.3节：用于更快幂运算的pow	266
第45.4节：无穷大和NaN（“不是数字”）	266
第45.5节：对数	269
第45.6节：常数	269
第45.7节：虚数	270
第45.8节：复制符号	270
第45.9节：复数与cmath模块	270
<b>第46章：复数数学</b>	273
第46.1节：高级复数运算	273
第46.2节：基础复数运算	274
<b>第47章：集合模块</b>	275
第47.1节：collections.Counter	275
第47.2节：collections.OrderedDict	276
第47.3节：collections.defaultdict	277
第47.4节：collections.namedtuple	278
第47.5节：collections.deque	279
第47.6节：collections.ChainMap	280
<b>第48章：操作符模块</b>	282
第48.1节：Itemgetter	282
第48.2节：操作符作为中缀操作符的替代	282
第48.3节：Methodcaller	282
<b>第49章：JSON模块</b>	284
第49.1节：将数据存储到文件中	284
第49.2节：从文件中检索数据	284
第49.3节：格式化JSON输出	284
第49.4节：`load` 与 `loads`, `dump` 与 `dumps`	285
第49.5节：从命令行调用 `json.tool` 以美化打印JSON输出	286
第49.6节：JSON编码自定义对象	286
第49.7节：从Python字典创建JSON	287
第49.8节：从JSON创建Python字典	287
<b>第50章：Sqlite3模块</b>	289
第50.1节：Sqlite3 - 不需要单独的服务器进程	289
第50.2节：从数据库获取值及错误处理	289
<b>第51章：os模块</b>	291
第51.1节：makedirs - 递归创建目录	291
第51.2节：创建目录	292
第51.3节：获取当前目录	292
第51.4节：确定操作系统名称	292
第51.5节：删除目录	292
第51.6节：跟随符号链接（POSIX）	292
第51.7节：更改文件权限	292
<b>第52章：locale模块</b>	293
第52.1节：使用locale模块格式化美元货币	293
<b>第53章：itertools模块</b>	294
第53.1节：Itertools模块中的组合方法	294

Section 44.2: Packages	262
<b>Chapter 45: Math Module</b>	264
Section 45.1: Rounding: round, floor, ceil, trunc	264
Section 45.2: Trigonometry	265
Section 45.3: Pow for faster exponentiation	266
Section 45.4: Infinity and NaN ("not a number")	266
Section 45.5: Logarithms	269
Section 45.6: Constants	269
Section 45.7: Imaginary Numbers	270
Section 45.8: Copying signs	270
Section 45.9: Complex numbers and the cmath module	270
<b>Chapter 46: Complex math</b>	273
Section 46.1: Advanced complex arithmetic	273
Section 46.2: Basic complex arithmetic	274
<b>Chapter 47: Collections module</b>	275
Section 47.1: collections.Counter	275
Section 47.2: collections.OrderedDict	276
Section 47.3: collections.defaultdict	277
Section 47.4: collections.namedtuple	278
Section 47.5: collections.deque	279
Section 47.6: collections.ChainMap	280
<b>Chapter 48: Operator module</b>	282
Section 48.1: Itemgetter	282
Section 48.2: Operators as alternative to an infix operator	282
Section 48.3: Methodcaller	282
<b>Chapter 49: JSON Module</b>	284
Section 49.1: Storing data in a file	284
Section 49.2: Retrieving data from a file	284
Section 49.3: Formatting JSON output	284
Section 49.4: `load` vs `loads`, `dump` vs `dumps`	285
Section 49.5: Calling `json.tool` from the command line to pretty-print JSON output	286
Section 49.6: JSON encoding custom objects	286
Section 49.7: Creating JSON from Python dict	287
Section 49.8: Creating Python dict from JSON	287
<b>Chapter 50: Sqlite3 Module</b>	289
Section 50.1: Sqlite3 - Not require separate server process	289
Section 50.2: Getting the values from the database and Error handling	289
<b>Chapter 51: The os Module</b>	291
Section 51.1: makedirs - recursive directory creation	291
Section 51.2: Create a directory	292
Section 51.3: Get current directory	292
Section 51.4: Determine the name of the operating system	292
Section 51.5: Remove a directory	292
Section 51.6: Follow a symlink (POSIX)	292
Section 51.7: Change permissions on a file	292
<b>Chapter 52: The locale Module</b>	293
Section 52.1: Currency Formatting US Dollars Using the locale Module	293
<b>Chapter 53: Itertools Module</b>	294
Section 53.1: Combinations method in Itertools Module	294

第53.2节 : <a href="#">itertools.dropwhile</a>	294	<a href="#">Section 53.2: itertools.dropwhile</a>	294
第53.3节 : 将两个迭代器压缩, 直到两者都耗尽	295	<a href="#">Section 53.3: Zipping two iterators until they are both exhausted</a>	295
第53.4节 : 对生成器进行切片	295	<a href="#">Section 53.4: Take a slice of a generator</a>	295
第53.5节 : 使用函数对可迭代对象中的项目进行分组	296	<a href="#">Section 53.5: Grouping items from an iterable object using a function</a>	296
第53.6节 : <a href="#">itertools.takewhile</a>	297	<a href="#">Section 53.6: itertools.takewhile</a>	297
第53.7节 : <a href="#">itertools.permutations</a>	297	<a href="#">Section 53.7: itertools.permutations</a>	297
第53.8节 : <a href="#">itertools.repeat</a>	298	<a href="#">Section 53.8: itertools.repeat</a>	298
第53.9节 : 获取可迭代对象中数字的累积和	298	<a href="#">Section 53.9: Get an accumulated sum of numbers in an iterable</a>	298
第53.10节 : 循环遍历迭代器中的元素	298	<a href="#">Section 53.10: Cycle through elements in an iterator</a>	298
第53.11节 : <a href="#">itertools.product</a>	298	<a href="#">Section 53.11: itertools.product</a>	298
第53.12节 : <a href="#">itertools.count</a>	299	<a href="#">Section 53.12: itertools.count</a>	299
第53.13节 : 将多个迭代器连接在一起	300	<a href="#">Section 53.13: Chaining multiple iterators together</a>	300
<b>第54章 : Asyncio模块</b>	301	<b>Chapter 54: Asyncio Module</b>	301
第54.1节 : 协程和委托语法	301	<a href="#">Section 54.1: Coroutine and Delegation Syntax</a>	301
第54.2节 : 异步执行器	302	<a href="#">Section 54.2: Asynchronous Executors</a>	302
第54.3节 : 使用UVLoop	303	<a href="#">Section 54.3: Using UVLoop</a>	303
第54.4节 : 同步原语 : 事件	303	<a href="#">Section 54.4: Synchronization Primitive: Event</a>	303
第54.5节 : 一个简单的Websocket	304	<a href="#">Section 54.5: A Simple Websocket</a>	304
第54.6节 : 关于asyncio的常见误解	304	<a href="#">Section 54.6: Common Misconception about asyncio</a>	304
<b>第55章 : 随机模块</b>	307	<b>Chapter 55: Random module</b>	307
第55.1节 : 创建随机用户密码	307	<a href="#">Section 55.1: Creating a random user password</a>	307
第55.2节 : 创建密码学安全的随机数	307	<a href="#">Section 55.2: Create cryptographically secure random numbers</a>	307
第55.3节 : 随机与序列 : 洗牌、选择和抽样	308	<a href="#">Section 55.3: Random and sequences: shuffle, choice and sample</a>	308
第55.4节 : 创建随机整数和浮点数 : randint、randrange、random和uniform	309	<a href="#">Section 55.4: Creating random integers and floats: randint, randrange, random, and uniform</a>	309
第55.5节 : 可复现的随机数 : 种子和状态	310	<a href="#">Section 55.5: Reproducible random numbers: Seed and State</a>	310
第55.6节 : 随机二进制决策	311	<a href="#">Section 55.6: Random Binary Decision</a>	311
<b>第56章 : Functools模块</b>	312	<b>Chapter 56: Functools Module</b>	312
第56.1节 : <a href="#">partial</a>	312	<a href="#">Section 56.1: partial</a>	312
第56.2节 : <a href="#">cmp_to_key</a>	312	<a href="#">Section 56.2: cmp_to_key</a>	312
第56.3节 : <a href="#">lru_cache</a>	312	<a href="#">Section 56.3: lru_cache</a>	312
第56.4节 : <a href="#">total_ordering</a>	313	<a href="#">Section 56.4: total_ordering</a>	313
第56.5节 : <a href="#">reduce</a>	314	<a href="#">Section 56.5: reduce</a>	314
<b>第57章 : dis模块</b>	315	<b>Chapter 57: The dis module</b>	315
第57.1节 : 什么是Python字节码 ?	315	<a href="#">Section 57.1: What is Python bytecode?</a>	315
第57.2节 : dis模块中的常量	315	<a href="#">Section 57.2: Constants in the dis module</a>	315
第57.3节 : 反汇编模块	315	<a href="#">Section 57.3: Disassembling modules</a>	315
<b>第58章 : base64模块</b>	317	<b>Chapter 58: The base64 Module</b>	317
第58.1节 : Base64的编码与解码	318	<a href="#">Section 58.1: Encoding and Decoding Base64</a>	318
第58.2节 : Base32的编码与解码	319	<a href="#">Section 58.2: Encoding and Decoding Base32</a>	319
第58.3节 : Base16的编码与解码	320	<a href="#">Section 58.3: Encoding and Decoding Base16</a>	320
第58.4节 : ASCII85的编码与解码	320	<a href="#">Section 58.4: Encoding and Decoding ASCII85</a>	320
第58.5节 : Base85的编码与解码	321	<a href="#">Section 58.5: Encoding and Decoding Base85</a>	321
<b>第59章 : 队列模块</b>	322	<b>Chapter 59: Queue Module</b>	322
第59.1节 : 简单示例	322	<a href="#">Section 59.1: Simple example</a>	322
<b>第60章 : 双端队列模块</b>	324	<b>Chapter 60: Deque Module</b>	324
第60.1节 : 使用基本双端队列	324	<a href="#">Section 60.1: Basic deque using</a>	324
第60.2节 : 双端队列中可用的方法	324	<a href="#">Section 60.2: Available methods in deque</a>	324
第60.3节 : 限制双端队列大小	325	<a href="#">Section 60.3: limit deque size</a>	325
第60.4节 : 广度优先搜索	325	<a href="#">Section 60.4: Breadth First Search</a>	325

<b>第61章：网页浏览器模块</b>	326	<b>Chapter 61: Webbrowser Module</b>	326
第61.1节：使用默认浏览器打开URL	326	Section 61.1: Opening a URL with Default Browser	326
第61.2节：使用不同浏览器打开URL	327	Section 61.2: Opening a URL with Different Browsers	327
<b>第62章：tkinter</b>	328	<b>Chapter 62: tkinter</b>	328
第62.1节：几何管理器	328	Section 62.1: Geometry Managers	328
第62.2节：一个最小的tkinter应用程序	329	Section 62.2: A minimal tkinter Application	329
<b>第63章：pyautogui模块</b>	331	<b>Chapter 63: pyautoogui module</b>	331
第63.1节：鼠标功能	331	Section 63.1: Mouse Functions	331
第63.2节：键盘功能	331	Section 63.2: Keyboard Functions	331
第63.3节：截图与图像识别	331	Section 63.3: Screenshot And Image Recognition	331
<b>第64章：索引与切片</b>	332	<b>Chapter 64: Indexing and Slicing</b>	332
第64.1节：基本切片	332	Section 64.1: Basic Slicing	332
第64.2节：反转对象	333	Section 64.2: Reversing an object	333
第64.3节：切片赋值	333	Section 64.3: Slice assignment	333
第64.4节：制作数组的浅拷贝	333	Section 64.4: Making a shallow copy of an array	333
第64.5节：自定义类的索引： <code>getitem</code> 、 <code>setitem</code> 和 <code>delitem</code>	334	Section 64.5: Indexing custom classes: <code>getitem</code> ， <code>setitem</code> and <code>delitem</code>	334
第64.6节：基本索引	335	Section 64.6: Basic Indexing	335
<b>第65章：使用Matplotlib绘图</b>	337	<b>Chapter 65: Plotting with Matplotlib</b>	337
第65.1节：具有相同X轴但不同Y轴的图：使用 <code>twinx()</code>	337	Section 65.1: Plots with Common X-axis but different Y-axis : Using <code>twinx()</code>	337
第65.2节：具有相同Y轴但不同X轴的图：使用 <code>twiny()</code>	338	Section 65.2: Plots with common Y-axis and different X-axis using <code>twiny()</code>	338
第65.3节：Matplotlib中的简单绘图	340	Section 65.3: A Simple Plot in Matplotlib	340
第65.4节：为简单图添加更多功能：坐标轴标签、标题、坐标轴刻度、网格和图例	341	Section 65.4: Adding more features to a simple plot : axis labels, title, axis ticks, grid, and legend	341
第65.5节：通过叠加在同一图形中绘制多个图，类似于MATLAB	342	Section 65.5: Making multiple plots in the same figure by superimposition similar to MATLAB	342
第65.6节：使用带有单独绘图命令的叠加绘图在同一图形中绘制多个图命令	343	Section 65.6: Making multiple Plots in the same figure using plot superimposition with separate plot commands	343
<b>第66章：graph-tool</b>	345	<b>Chapter 66: graph-tool</b>	345
第66.1节：PyDotPlus	345	Section 66.1: PyDotPlus	345
第66.2节：PyGraphviz	345	Section 66.2: PyGraphviz	345
<b>第67章：生成器</b>	347	<b>Chapter 67: Generators</b>	347
第67.1节：介绍	347	Section 67.1: Introduction	347
第67.2节：无限序列	349	Section 67.2: Infinite sequences	349
第67.3节：向生成器发送对象	350	Section 67.3: Sending objects to a generator	350
第67.4节：从另一个可迭代对象生成所有值	351	Section 67.4: Yielding all values from another iterable	351
第67.5节：迭代	351	Section 67.5: Iteration	351
第67.6节： <code>next()</code> 函数	351	Section 67.6: The <code>next()</code> function	351
第67.7节：协程	352	Section 67.7: Coroutines	352
第67.8节：重构列表构建代码	352	Section 67.8: Refactoring list-building code	352
第67.9节：带递归的 <code>yield</code> ：递归列出目录中的所有文件	353	Section 67.9: Yield with recursion: recursively listing all files in a directory	353
第67.10节：生成器表达式	354	Section 67.10: Generator expressions	354
第67.11节：使用生成器查找斐波那契数列	354	Section 67.11: Using a generator to find Fibonacci Numbers	354
第67.12节：搜索	354	Section 67.12: Searching	354
第67.13节：并行迭代生成器	355	Section 67.13: Iterating over generators in parallel	355
<b>第68章：归约</b>	356	<b>Chapter 68: Reduce</b>	356
第68.1节：概述	356	Section 68.1: Overview	356
第68.2节：使用归约	356	Section 68.2: Using reduce	356
第68.3节：累积乘积	357	Section 68.3: Cumulative product	357
第68.4节：any/all的非短路变体	357	Section 68.4: Non short-circuit variant of any/all	357
<b>第69章：映射函数</b>	358	<b>Chapter 69: Map Function</b>	358
第69.1节： <code>map</code> 、 <code>itertools imap</code> 和 <code>future_builtins map</code> 的基本用法	358	Section 69.1: Basic use of <code>map</code> , <code>itertools imap</code> and <code>future_builtins.map</code>	358

第69.2节：映射可迭代对象中的每个值	358	<a href="#">Section 69.2: Mapping each value in an iterable</a>	358
第69.3节：映射不同可迭代对象的值	359	<a href="#">Section 69.3: Mapping values of different iterables</a>	359
第69.4节：使用Map进行转置：将“None”作为函数参数（仅限Python 2.x）	361	<a href="#">Section 69.4: Transposing with Map: Using "None" as function argument (python 2.x only)</a>	361
第69.5节：串行和并行映射	361	<a href="#">Section 69.5: Series and Parallel Mapping</a>	361
<b>第70章：指数运算</b>	365	<b>Chapter 70: Exponentiation</b>	365
第70.1节：使用内置函数进行指数运算：** 和 pow()	365	<a href="#">Section 70.1: Exponentiation using builtins: ** and pow()</a>	365
第70.2节：平方根：math.sqrt() 和 cmath.sqrt	365	<a href="#">Section 70.2: Square root: math.sqrt() and cmath.sqrt</a>	365
第70.3节：模幂运算：带三个参数的 pow()	366	<a href="#">Section 70.3: Modular exponentiation: pow() with 3 arguments</a>	366
第70.4节：计算大整数根	366	<a href="#">Section 70.4: Computing large integer roots</a>	366
第70.5节：使用 math 模块的幂运算：math.pow()	367	<a href="#">Section 70.5: Exponentiation using the math module: math.pow()</a>	367
第70.6节：指数函数：math.exp() 和 cmath.exp()	368	<a href="#">Section 70.6: Exponential function: math.exp() and cmath.exp()</a>	368
第70.7节：指数函数减1：math.expm1()	368	<a href="#">Section 70.7: Exponential function minus 1: math.expm1()</a>	368
第70.8节：魔术方法与指数运算：内置、math和cmath	369	<a href="#">Section 70.8: Magic methods and exponentiation: builtin, math and cmath</a>	369
第70.9节：根：带分数指数的n次根	370	<a href="#">Section 70.9: Roots: nth-root with fractional exponents</a>	370
<b>第71章：搜索</b>	371	<b>Chapter 71: Searching</b>	371
第71.1节：元素搜索	371	<a href="#">Section 71.1: Searching for an element</a>	371
第71.2节：自定义类中的搜索：__contains__ 和 __iter__	371	<a href="#">Section 71.2: Searching in custom classes: __contains__ and __iter__</a>	371
第71.3节：获取字符串索引：str.index()、str.rindex() 和 str.find()、str.rfind()	372	<a href="#">Section 71.3: Getting the index for strings: str.index(), str.rindex() and str.find(), str.rfind()</a>	372
第71.4节：获取索引列表和元组：list.index()，tuple.index()	373	<a href="#">Section 71.4: Getting the index list and tuples: list.index(), tuple.index()</a>	373
第71.5节：在字典中搜索值对应的键	373	<a href="#">Section 71.5: Searching key(s) for a value in dict</a>	373
第71.6节：获取排序序列的索引：bisect.bisect_left()	374	<a href="#">Section 71.6: Getting the index for sorted sequences: bisect.bisect_left()</a>	374
第71.7节：搜索嵌套序列	374	<a href="#">Section 71.7: Searching nested sequences</a>	374
<b>第72章：排序、最小值和最大值</b>	376	<b>Chapter 72: Sorting, Minimum and Maximum</b>	376
第72.1节：使自定义类可排序	376	<a href="#">Section 72.1: Make custom classes orderable</a>	376
第72.2节：特殊情况：字典	378	<a href="#">Section 72.2: Special case: dictionaries</a>	378
第72.3节：使用key参数	379	<a href="#">Section 72.3: Using the key argument</a>	379
第72.4节：max和min的默认参数	379	<a href="#">Section 72.4: Default Argument to max, min</a>	379
第72.5节：获取排序序列	380	<a href="#">Section 72.5: Getting a sorted sequence</a>	380
第72.6节：从可迭代对象中提取N个最大或最小的元素	380	<a href="#">Section 72.6: Extracting N largest or N smallest items from an iterable</a>	380
第72.7节：获取多个值的最小值或最大值	381	<a href="#">Section 72.7: Getting the minimum or maximum of several values</a>	381
第72.8节：序列的最小值和最大值	381	<a href="#">Section 72.8: Minimum and Maximum of a sequence</a>	381
<b>第73章：计数</b>	382	<b>Chapter 73: Counting</b>	382
第73.1节：统计可迭代对象中所有元素的出现次数：collections.Counter	382	<a href="#">Section 73.1: Counting all occurrence of all items in an iterable: collections.Counter</a>	382
第73.2节：获取最常见的值（多个）：collections.Counter.most_common()	382	<a href="#">Section 73.2: Getting the most common value(-s): collections.Counter.most_common()</a>	382
第73.3节：统计序列中某个元素的出现次数：list.count() 和 tuple.count()	382	<a href="#">Section 73.3: Counting the occurrences of one item in a sequence: list.count() and tuple.count()</a>	382
第73.4节：统计字符串中子串的出现次数：str.count()	383	<a href="#">Section 73.4: Counting the occurrences of a substring in a string: str.count()</a>	383
第73.5节：计算numpy数组中的出现次数	383	<a href="#">Section 73.5: Counting occurrences in numpy array</a>	383
<b>第74章：打印函数</b>	384	<b>Chapter 74: The Print Function</b>	384
第74.1节：打印基础	384	<a href="#">Section 74.1: Print basics</a>	384
第74.2节：打印参数	385	<a href="#">Section 74.2: Print parameters</a>	385
<b>第75章：正则表达式（Regex）</b>	388	<b>Chapter 75: Regular Expressions (Regex)</b>	388
第75.1节：匹配字符串开头	388	<a href="#">Section 75.1: Matching the beginning of a string</a>	388
第75.2节：搜索	389	<a href="#">Section 75.2: Searching</a>	389
第75.3节：预编译模式	389	<a href="#">Section 75.3: Precompiled patterns</a>	389
第75.4节：标志	390	<a href="#">Section 75.4: Flags</a>	390
第75.5节：替换	391	<a href="#">Section 75.5: Replacing</a>	391
第75.6节：查找所有不重叠的匹配项	391	<a href="#">Section 75.6: Find All Non-Overlapping Matches</a>	391
第75.7节：检查允许的字符	392	<a href="#">Section 75.7: Checking for allowed characters</a>	392
第75.8节：使用正则表达式拆分字符串	392	<a href="#">Section 75.8: Splitting a string using regular expressions</a>	392
第75.9节：分组	392	<a href="#">Section 75.9: Grouping</a>	392

第75.10节：转义特殊字符	393
第75.11节：仅在特定位置匹配表达式	394
第75.12节：使用`re.finditer`迭代匹配项	395
<b>第76章：复制数据</b>	396
第76.1节：复制字典	396
第76.2节：执行浅复制	396
第76.3节：执行深复制	396
第76.4节：执行列表的浅拷贝	396
第76.5节：拷贝集合	396
<b>第77章：上下文管理器（“with”语句）</b>	398
第77.1节：上下文管理器和with语句简介	398
第77.2节：编写你自己的上下文管理器	398
第77.3节：使用生成器语法编写你自己的上下文管理器	399
第77.4节：多个上下文管理器	400
第77.5节：分配给目标	400
第77.6节：管理资源	401
<b>第78章：name 特殊变量</b>	402
第78.1节：`name == 'main'`	402
第78.2节：在日志中的使用	402
第78.3节：`function class or module. name`	402
<b>第79章：检查路径存在性和权限</b>	404
第79.1节：使用`os.access`进行检查	404
<b>第80章：创建Python包</b>	406
第80.1节：介绍	406
第80.2节：上传到PyPI	406
第80.3节：使包可执行	408
<b>第81章：“pip”模块的使用：PyPI包管理器</b>	410
第81.1节：命令使用示例	410
第81.2节：处理`ImportError`异常	410
第81.3节：强制安装	411
<b>第82章：pip：PyPI包管理器</b>	412
第82.1节：安装包	412
第82.2节：列出所有使用`pip`安装的包	412
第82.3节：升级包	412
第82.4节：卸载包	413
第82.5节：在Linux上更新所有过时的包	413
第82.6节：更新Windows上所有过时的软件包	413
第82.7节：创建系统中所有软件包的`requirements.txt`文件	413
第82.8节：使用特定Python版本的`pip`	414
第82.9节：仅创建当前虚拟环境中软件包的`requirements.txt`文件	414
第82.10节：以wheel格式安装尚未在`pip`上的软件包	415
<b>第83章：解析命令行参数</b>	418
第83.1节：`argparse`中的Hello_world	418
第83.2节：使用`argv`传递命令行参数	418
第83.3节：使用`argparse`设置互斥参数	419
第83.4节：使用`docopt`的基本示例	420
第83.5节：使用`argparse`自定义解析器错误消息	420
第83.6节：使用`argparse.add_argument_group()`对参数进行概念分组	421
第83.7节：使用`docopt`和`docopt_dispatch`的高级示例	422

Section 75.10: Escaping Special Characters	393
Section 75.11: Match an expression only in specific locations	394
Section 75.12: Iterating over matches using `re.finditer`	395
<b>Chapter 76: Copying data</b>	396
Section 76.1: Copy a dictionary	396
Section 76.2: Performing a shallow copy	396
Section 76.3: Performing a deep copy	396
Section 76.4: Performing a shallow copy of a list	396
Section 76.5: Copy a set	396
<b>Chapter 77: Context Managers (“with” Statement)</b>	398
Section 77.1: Introduction to context managers and the with statement	398
Section 77.2: Writing your own context manager	398
Section 77.3: Writing your own contextmanager using generator syntax	399
Section 77.4: Multiple context managers	400
Section 77.5: Assigning to a target	400
Section 77.6: Manage Resources	401
<b>Chapter 78: The __name__ special variable</b>	402
Section 78.1: __name__ == '__main__'	402
Section 78.2: Use in logging	402
Section 78.3: function class or module. __name__	402
<b>Chapter 79: Checking Path Existence and Permissions</b>	404
Section 79.1: Perform checks using os.access	404
<b>Chapter 80: Creating Python packages</b>	406
Section 80.1: Introduction	406
Section 80.2: Uploading to PyPI	406
Section 80.3: Making package executable	408
<b>Chapter 81: Usage of "pip" module: PyPI Package Manager</b>	410
Section 81.1: Example use of commands	410
Section 81.2: Handling ImportError Exception	410
Section 81.3: Force install	411
<b>Chapter 82: pip: PyPI Package Manager</b>	412
Section 82.1: Install Packages	412
Section 82.2: To list all packages installed using `pip`	412
Section 82.3: Upgrade Packages	412
Section 82.4: Uninstall Packages	413
Section 82.5: Updating all outdated packages on Linux	413
Section 82.6: Updating all outdated packages on Windows	413
Section 82.7: Create a requirements.txt file of all packages on the system	413
Section 82.8: Using a certain Python version with pip	414
Section 82.9: Create a requirements.txt file of packages only in the current virtualenv	414
Section 82.10: Installing packages not yet on pip as wheels	415
<b>Chapter 83: Parsing Command Line arguments</b>	418
Section 83.1: Hello world in argparse	418
Section 83.2: Using command line arguments with argv	418
Section 83.3: Setting mutually exclusive arguments with argparse	419
Section 83.4: Basic example with docopt	420
Section 83.5: Custom parser error message with argparse	420
Section 83.6: Conceptual grouping of arguments with argparse.add_argument_group()	421
Section 83.7: Advanced example with docopt and docopt_dispatch	422

<b>第84章：子进程库</b>	424
第84.1节：使用Popen实现更多灵活性	424
第84.2节：调用外部命令	425
第84.3节：如何创建命令列表参数	425
<b>第85章：setup.py</b>	427
第85.1节：setup.py的目的	427
第85.2节：在setup.py中使用源代码管理元数据	427
第85.3节：向你的Python包添加命令行脚本	428
第85.4节：添加安装选项	428
<b>第86章：递归</b>	430
第86.1节：递归的是什么、如何以及何时使用	430
第86.2节：使用递归进行树的遍历	433
第86.3节：从1加到n的数字之和	434
第86.4节：增加最大递归深度	434
第86.5节：尾递归——不良实践	435
第86.6节：通过栈内省进行尾递归优化	435
<b>第87章：类型提示</b>	437
第87.1节：为函数添加类型	437
第87.2节：命名元组 (NamedTuple)	438
第87.3节：泛型类型	438
第87.4节：变量和属性	438
第87.5节：类成员和方法	439
第87.6节：关键字参数的类型提示	439
<b>第88章：异常</b>	440
第88.1节：捕获异常	440
第88.2节：不要捕获所有异常！	440
第88.3节：重新抛出异常	441
第88.4节：捕获多个异常	441
第88.5节：异常层次结构	442
第88.6节：Else	444
第88.7节：引发异常	444
第88.8节：创建自定义异常类型	445
第88.9节：异常处理的实际示例	445
第88.10节：异常也是对象	446
第88.11节：使用finally运行清理代码	446
第88.12节：使用raise from链式异常	447
<b>第89章：引发自定义错误/异常</b>	448
第89.1节：自定义异常	448
第89.2节：捕获自定义异常	448
<b>第90章：联邦异常</b>	450
第90.1节：其他错误	450
第90.2节：NameError：名称 '???' 未定义	451
第90.3节：类型错误	452
第90.4节：良好代码中的语法错误	453
第90.5节：缩进错误（或缩进语法错误）	454
<b>第91章：urllib</b>	456
第91.1节：HTTP GET	456
第91.2节：HTTP POST	456
第91.3节：根据内容类型编码解码接收的字节	457

<b>Chapter 84: Subprocess Library</b>	424
Section 84.1: More flexibility with Popen	424
Section 84.2: Calling External Commands	425
Section 84.3: How to create the command list argument	425
<b>Chapter 85: setup.py</b>	427
Section 85.1: Purpose of setup.py	427
Section 85.2: Using source control metadata in setup.py	427
Section 85.3: Adding command line scripts to your python package	428
Section 85.4: Adding installation options	428
<b>Chapter 86: Recursion</b>	430
Section 86.1: The What, How, and When of Recursion	430
Section 86.2: Tree exploration with recursion	433
Section 86.3: Sum of numbers from 1 to n	434
Section 86.4: Increasing the Maximum Recursion Depth	434
Section 86.5: Tail Recursion - Bad Practice	435
Section 86.6: Tail Recursion Optimization Through Stack Introspection	435
<b>Chapter 87: Type Hints</b>	437
Section 87.1: Adding types to a function	437
Section 87.2: NamedTuple	438
Section 87.3: Generic Types	438
Section 87.4: Variables and Attributes	438
Section 87.5: Class Members and Methods	439
Section 87.6: Type hints for keyword arguments	439
<b>Chapter 88: Exceptions</b>	440
Section 88.1: Catching Exceptions	440
Section 88.2: Do not catch everything!	440
Section 88.3: Re-raising exceptions	441
Section 88.4: Catching multiple exceptions	441
Section 88.5: Exception Hierarchy	442
Section 88.6: Else	444
Section 88.7: Raising Exceptions	444
Section 88.8: Creating custom exception types	445
Section 88.9: Practical examples of exception handling	445
Section 88.10: Exceptions are Objects too	446
Section 88.11: Running clean-up code with finally	446
Section 88.12: Chain exceptions with raise from	447
<b>Chapter 89: Raise Custom Errors / Exceptions</b>	448
Section 89.1: Custom Exception	448
Section 89.2: Catch custom Exception	448
<b>Chapter 90: Commonwealth Exceptions</b>	450
Section 90.1: Other Errors	450
Section 90.2: NameError: name '???' is not defined	451
Section 90.3: TypeErrors	452
Section 90.4: Syntax Error on good code	453
Section 90.5: IndentationErrors (or indentation SyntaxErrors)	454
<b>Chapter 91: urllib</b>	456
Section 91.1: HTTP GET	456
Section 91.2: HTTP POST	456
Section 91.3: Decode received bytes according to content type encoding	457

<b>第92章：使用Python进行网页爬取</b>	458
第92.1节：使用Scrapy框架进行爬取	458
第92.2节：使用Selenium WebDriver进行爬取	458
第92.3节：使用requests和lxml爬取数据的基本示例	459
第92.4节：使用requests维护网页爬取会话	459
第92.5节：使用BeautifulSoup4进行爬取	460
第92.6节：使用urllib.request简单下载网页内容	460
第92.7节：修改Scrapy用户代理	460
第92.8节：使用curl进行爬取	460
<b>第93章：HTML解析</b>	462
第93.1节：在BeautifulSoup中使用CSS选择器	462
第93.2节：PyQuery	462
第93.3节：在BeautifulSoup中定位元素后的文本	463
<b>第94章：操作XML</b>	464
第94.1节：使用ElementTree打开和读取	464
第94.2节：创建和构建XML文档	464
第94.3节：修改XML文件	465
第94.4节：使用XPath搜索XML	465
第94.5节：使用iterparse（增量解析）打开和读取大型XML文件	466
<b>第95章：Python Requests的Post方法</b>	468
第95.1节：简单表单提交	468
第95.2节：表单编码数据	469
第95.3节：文件上传	469
第95.4节：响应	470
第95.5节：认证	470
第95.6节：代理	471
<b>第96章：分布</b>	473
第96.1节：py2app	473
第96.2节：cx_Freeze	474
<b>第97章：属性对象</b>	475
第97.1节：使用@property装饰器实现可读写属性	475
第97.2节：使用@property装饰器	475
第97.3节：重写属性对象的getter、setter或deleter	476
第97.4节：使用无装饰器的属性	476
<b>第98章：重载</b>	479
第98.1节：运算符重载	479
第98.2节：魔法方法/双下划线方法	480
第98.3节：容器和序列类型	481
第98.4节：可调用类型	482
第98.5节：处理未实现的行为	482
<b>第99章：多态性</b>	484
第99.1节：鸭子类型	484
第99.2节：基本多态性	484
<b>第100章：方法重写</b>	488
第100.1节：基本方法重写	488
<b>第101章：用户自定义方法</b>	489
第101.1节：创建用户自定义方法对象	489
第101.2节：海龟示例	490
<b>第102章：类实例的字符串表示：str 和 repr 方法</b>	

<b>Chapter 92: Web scraping with Python</b>	458
Section 92.1: Scraping using the Scrapy framework	458
Section 92.2: Scraping using Selenium WebDriver	458
Section 92.3: Basic example of using requests and lxml to scrape some data	459
Section 92.4: Maintaining web-scraping session with requests	459
Section 92.5: Scraping using BeautifulSoup4	460
Section 92.6: Simple web content download with urllib.request	460
Section 92.7: Modify Scrapy user agent	460
Section 92.8: Scraping with curl	460
<b>Chapter 93: HTML Parsing</b>	462
Section 93.1: Using CSS selectors in BeautifulSoup	462
Section 93.2: PyQuery	462
Section 93.3: Locate a text after an element in BeautifulSoup	463
<b>Chapter 94: Manipulating XML</b>	464
Section 94.1: Opening and reading using an ElementTree	464
Section 94.2: Create and Build XML Documents	464
Section 94.3: Modifying an XML File	465
Section 94.4: Searching the XML with XPath	465
Section 94.5: Opening and reading large XML files using iterparse (incremental parsing)	466
<b>Chapter 95: Python Requests Post</b>	468
Section 95.1: Simple Post	468
Section 95.2: Form Encoded Data	469
Section 95.3: File Upload	469
Section 95.4: Responses	470
Section 95.5: Authentication	470
Section 95.6: Proxies	471
<b>Chapter 96: Distribution</b>	473
Section 96.1: py2app	473
Section 96.2: cx_Freeze	474
<b>Chapter 97: Property Objects</b>	475
Section 97.1: Using the @property decorator for read-write properties	475
Section 97.2: Using the @property decorator	475
Section 97.3: Overriding just a getter, setter or a deleter of a property object	476
Section 97.4: Using properties without decorators	476
<b>Chapter 98: Overloading</b>	479
Section 98.1: Operator overloading	479
Section 98.2: Magic/Dunder Methods	480
Section 98.3: Container and sequence types	481
Section 98.4: Callable types	482
Section 98.5: Handling unimplemented behaviour	482
<b>Chapter 99: Polymorphism</b>	484
Section 99.1: Duck Typing	484
Section 99.2: Basic Polymorphism	484
<b>Chapter 100: Method Overriding</b>	488
Section 100.1: Basic method overriding	488
<b>Chapter 101: User-Defined Methods</b>	489
Section 101.1: Creating user-defined method objects	489
Section 101.2: Turtle example	490
<b>Chapter 102: String representations of class instances: str and repr</b>	

<b>方法</b>	491
第102.1节：动机	491
第102.2节：两种方法的实现， <code>eval</code> -回环风格的 <code>repr()</code>	495
<b>第103章：调试</b>	496
第103.1节：通过 IPython 和 <code>ipdb</code>	496
第103.2节：Python 调试器：使用 <code>pdb</code> 逐步调试	496
第103.3节：远程调试器	498
<b>第104章：读取和写入 CSV</b>	499
第104.1节：使用 <code>pandas</code>	499
第104.2节：写入 TSV 文件	499
<b>第105章：从字符串或列表写入 CSV</b>	501
第105.1节：基本写入示例	501
第105.2节：在CSV文件中追加字符串作为换行符	501
<b>第106章：使用`exec`和`eval`的动态代码执行</b>	502
第106.1节：使用 <code>exec</code> 、 <code>eval</code> 或 <code>ast.literal_eval</code> 执行不受信任用户提供的代码	502
第106.2节：使用 <code>ast.literal_eval</code> 评估包含Python字面量的字符串	502
第106.3节：使用 <code>exec</code> 执行语句	502
第106.4节：使用 <code>eval</code> 评估表达式	503
第106.5节：预编译表达式以多次评估	503
第106.6节：使用自定义全局变量通过 <code>eval</code> 评估表达式	503
<b>第107章：PyInstaller——分发Python代码</b>	504
第107.1节：安装与设置	504
第107.2节：使用PyInstaller	504
第107.3节：打包到一个文件夹	505
第107.4节：打包成单个文件	505
<b>第108章：使用Python进行数据可视化</b>	506
第108.1节： <code>Seaborn</code>	506
第108.2节： <code>Matplotlib</code>	508
第108.3节： <code>Plotly</code>	509
第108.4节： <code>MayaVI</code>	511
<b>第109章：解释器（命令行控制台）</b>	513
第109.1节：获取一般帮助	513
第109.2节：引用最后一个表达式	513
第109.3节：打开Python控制台	514
第109.4节： <code>PYTHONSTARTUP</code> 变量	514
第109.5节：命令行参数	514
第109.6节：获取对象的帮助	515
<b>第110章：*args和**kwargs</b>	518
第110.1节：编写函数时使用 <code>**kwargs</code>	518
第110.2节：编写函数时使用 <code>*args</code>	518
第110.3节：用字典填充 <code>kwarg</code> 值	519
第110.4节：仅限关键字和必需关键字参数	519
第110.5节：调用函数时使用 <code>**kwargs</code>	519
第110.6节： <code>**kwargs</code> 与默认值	519
第110.7节：调用函数时使用 <code>*args</code>	520
<b>第111章：垃圾回收</b>	521
第111.1节：原始对象的重用	521
第111.2节： <code>del</code> 命令的影响	521
第111.3节：引用计数	522
<b>methods</b>	491
Section 102.1: Motivation	491
Section 102.2: Both methods implemented, eval-round-trip style <code>repr()</code>	495
<b>Chapter 103: Debugging</b>	496
Section 103.1: Via IPython and <code>ipdb</code>	496
Section 103.2: The Python Debugger: Step-through Debugging with <code>pdb</code>	496
Section 103.3: Remote debugger	498
<b>Chapter 104: Reading and Writing CSV</b>	499
Section 104.1: Using <code>pandas</code>	499
Section 104.2: Writing a TSV file	499
<b>Chapter 105: Writing to CSV from String or List</b>	501
Section 105.1: Basic Write Example	501
Section 105.2: Appending a String as a newline in a CSV file	501
<b>Chapter 106: Dynamic code execution with `exec` and `eval`</b>	502
Section 106.1: Executing code provided by untrusted user using <code>exec</code> , <code>eval</code> , or <code>ast.literal_eval</code>	502
Section 106.2: Evaluating a string containing a Python literal with <code>ast.literal_eval</code>	502
Section 106.3: Evaluating statements with <code>exec</code>	502
Section 106.4: Evaluating an expression with <code>eval</code>	503
Section 106.5: Precompiling an expression to evaluate it multiple times	503
Section 106.6: Evaluating an expression with <code>eval</code> using custom globals	503
<b>Chapter 107: PyInstaller - Distributing Python Code</b>	504
Section 107.1: Installation and Setup	504
Section 107.2: Using Pyinstaller	504
Section 107.3: Bundling to One Folder	505
Section 107.4: Bundling to a Single File	505
<b>Chapter 108: Data Visualization with Python</b>	506
Section 108.1: Seaborn	506
Section 108.2: Matplotlib	508
Section 108.3: Plotly	509
Section 108.4: MayaVI	511
<b>Chapter 109: The Interpreter (Command Line Console)</b>	513
Section 109.1: Getting general help	513
Section 109.2: Referring to the last expression	513
Section 109.3: Opening the Python console	514
Section 109.4: The <code>PYTHONSTARTUP</code> variable	514
Section 109.5: Command line arguments	514
Section 109.6: Getting help about an object	515
<b>Chapter 110: *args and **kwargs</b>	518
Section 110.1: Using <code>**kwargs</code> when writing functions	518
Section 110.2: Using <code>*args</code> when writing functions	518
Section 110.3: Populating kwarg values with a dictionary	519
Section 110.4: Keyword-only and Keyword-required arguments	519
Section 110.5: Using <code>**kwargs</code> when calling functions	519
Section 110.6: <code>**kwargs</code> and default values	519
Section 110.7: Using <code>*args</code> when calling functions	520
<b>Chapter 111: Garbage Collection</b>	521
Section 111.1: Reuse of primitive objects	521
Section 111.2: Effects of the <code>del</code> command	521
Section 111.3: Reference Counting	522

第111.4节：引用循环的垃圾回收器	522
第111.5节：强制释放对象	523
第111.6节：查看对象的引用计数	524
第111.7节：不要等待垃圾回收来清理	524
第111.8节：管理垃圾回收	524
<b>第112章：Pickle数据序列化</b>	526
第112.1节：使用Pickle序列化和反序列化对象	526
第112.2节：自定义Pickle数据	526
<b>第113章：二进制数据</b>	528
第113.1节：将值列表格式化为字节对象	528
第113.2节：根据格式字符串解包字节对象	528
第113.3节：打包结构	528
<b>第114章：习语</b>	530
第114.1节：字典键初始化	530
第114.2节：切换变量	530
第114.3节：使用真值测试	530
第114.4节：测试“ <code>main</code> ”以避免意外代码执行	531
<b>第115章：数据序列化</b>	533
第115.1节：使用JSON进行序列化	533
第115.2节：使用Pickle进行序列化	533
<b>第116章：多进程</b>	535
第116.1节：运行两个简单进程	535
第116.2节：使用Pool和Map	535
<b>第117章：多线程</b>	537
第117.1节：多线程基础	537
第117.2节：线程间通信	538
第117.3节：创建工作池	539
第117.4节：多线程的高级使用	539
第117.5节：带有while循环的可停止线程	541
<b>第118章：进程与线程</b>	542
第118.1节：全局解释器锁	542
第118.2节：多线程运行	543
第118.3节：多进程运行	544
第118.4节：线程间共享状态	544
第118.5节：进程间共享状态	545
<b>第119章：Python 并发</b>	546
第119.1节： <code>multiprocessing</code> 模块	546
第119.2节： <code>threading</code> 模块	547
第119.3节：多进程间传递数据	547
<b>第120章：并行计算</b>	550
第120.1节：使用multiprocessing模块实现任务并行	550
第120.2节：使用C扩展实现任务并行	550
第120.3节：使用父子脚本并行执行代码	550
第120.4节：使用PyPar模块实现并行	551
<b>第121章：套接字</b>	552
第121.1节：Linux上的原始套接字	552
第121.2节：通过UDP发送数据	552
第121.3节：通过UDP接收数据	553
第121.4节：通过TCP发送数据	553

Section 111.4: Garbage Collector for Reference Cycles	522
Section 111.5: Forcefully deallocating objects	523
Section 111.6: Viewing the refcount of an object	524
Section 111.7: Do not wait for the garbage collection to clean up	524
Section 111.8: Managing garbage collection	524
<b>Chapter 112: Pickle data serialisation</b>	526
Section 112.1: Using Pickle to serialize and deserialize an object	526
Section 112.2: Customize Pickled Data	526
<b>Chapter 113: Binary Data</b>	528
Section 113.1: Format a list of values into a byte object	528
Section 113.2: Unpack a byte object according to a format string	528
Section 113.3: Packing a structure	528
<b>Chapter 114: Idioms</b>	530
Section 114.1: Dictionary key initializations	530
Section 114.2: Switching variables	530
Section 114.3: Use truth value testing	530
Section 114.4: Test for “ <code>main</code> ” to avoid unexpected code execution	531
<b>Chapter 115: Data Serialization</b>	533
Section 115.1: Serialization using JSON	533
Section 115.2: Serialization using Pickle	533
<b>Chapter 116: Multiprocessing</b>	535
Section 116.1: Running Two Simple Processes	535
Section 116.2: Using Pool and Map	535
<b>Chapter 117: Multithreading</b>	537
Section 117.1: Basics of multithreading	537
Section 117.2: Communicating between threads	538
Section 117.3: Creating a worker pool	539
Section 117.4: Advanced use of multithreads	539
Section 117.5: Stoppable Thread with a while Loop	541
<b>Chapter 118: Processes and Threads</b>	542
Section 118.1: Global Interpreter Lock	542
Section 118.2: Running in Multiple Threads	543
Section 118.3: Running in Multiple Processes	544
Section 118.4: Sharing State Between Threads	544
Section 118.5: Sharing State Between Processes	545
<b>Chapter 119: Python concurrency</b>	546
Section 119.1: The <code>multiprocessing</code> module	546
Section 119.2: The <code>threading</code> module	547
Section 119.3: Passing data between multiprocessing processes	547
<b>Chapter 120: Parallel computation</b>	550
Section 120.1: Using the multiprocessing module to parallelise tasks	550
Section 120.2: Using a C-extension to parallelize tasks	550
Section 120.3: Using Parent and Children scripts to execute code in parallel	550
Section 120.4: Using PyPar module to parallelize	551
<b>Chapter 121: Sockets</b>	552
Section 121.1: Raw Sockets on Linux	552
Section 121.2: Sending data via UDP	552
Section 121.3: Receiving data via UDP	553
Section 121.4: Sending data via TCP	553

第121.5节：多线程TCP套接字服务器	553
<b>第122章：Websockets</b>	556
第122.1节：使用aiohttp的简单回声	556
第122.2节：使用aiohttp的包装类	556
第122.3节：使用Autobahn作为Websocket工厂	557
<b>第123章：客户端与服务器之间的套接字及消息加密/解密</b>	559
第123.1节：服务器端实现	559
第123.2节：客户端实现	561
<b>第124章：Python网络编程</b>	563
第124.1节：创建一个简单的Http服务器	563
第124.2节：创建一个TCP服务器	563
第124.3节：创建一个UDP服务器	564
第124.4节：在线程中启动简单的HttpServer并打开浏览器	564
第124.5节：最简单的Python套接字客户端-服务器示例	565
<b>第125章：Python HTTP服务器</b>	567
第125.1节：运行一个简单的HTTP服务器	567
第125.2节：提供文件服务	567
第125.3节：使用BaseHTTPRequestHandler对GET、POST、PUT的基本处理	568
第125.4节：SimpleHTTPServer的程序化API	569
<b>第126章：Flask</b>	571
第126.1节：文件和模板	571
第126.2节：基础知识	571
第126.3节：路由URL	572
第126.4节：HTTP方法	573
第126.5节：Jinja模板	573
第126.6节：请求对象	574
<b>第127章：使用AMQPStorm介绍RabbitMQ</b>	576
第127.1节：如何从RabbitMQ消费消息	576
第127.2节：如何向RabbitMQ发布消息	577
第127.3节：如何在RabbitMQ中创建延迟队列	577
<b>第128章：描述符</b>	580
第128.1节：简单描述符	580
第128.2节：双向转换	581
<b>第129章：临时文件 NamedTemporaryFile</b>	582
第129.1节：创建（并写入）已知的持久临时文件	582
<b>第130章：使用Pandas输入、子集和输出外部数据文件</b>	584
第130.1节：使用Pandas导入、子集和写入外部数据文件的基本代码	584
<b>第131章：解压文件</b>	586
第131.1节：使用Python ZipFile.extractall()解压ZIP文件	586
第131.2节：使用Python TarFile.extractall()解压tar包	586
<b>第132章：处理ZIP归档文件</b>	587
第132.1节：检查Zipfile内容	587
第132.2节：打开Zip文件	587
第132.3节：将zip文件内容解压到目录	588
第132.4节：创建新档案	588
<b>第133章：开始使用GZip</b>	589
第133.1节：读取和写入GNU压缩文件	589

Section 121.5: Multi-threaded TCP Socket Server	553
<b>Chapter 122: Websockets</b>	556
Section 122.1: Simple Echo with aiohttp	556
Section 122.2: Wrapper Class with aiohttp	556
Section 122.3: Using Autobahn as a Websocket Factory	557
<b>Chapter 123: Sockets And Message Encryption/Decryption Between Client and Server</b>	559
Section 123.1: Server side Implementation	559
Section 123.2: Client side Implementation	561
<b>Chapter 124: Python Networking</b>	563
Section 124.1: Creating a Simple Http Server	563
Section 124.2: Creating a TCP server	563
Section 124.3: Creating a UDP Server	564
Section 124.4: Start Simple HttpServer in a thread and open the browser	564
Section 124.5: The simplest Python socket client-server example	565
<b>Chapter 125: Python HTTP Server</b>	567
Section 125.1: Running a simple HTTP server	567
Section 125.2: Serving files	567
Section 125.3: Basic handling of GET, POST, PUT using BaseHTTPRequestHandler	568
Section 125.4: Programmatic API of SimpleHTTPServer	569
<b>Chapter 126: Flask</b>	571
Section 126.1: Files and Templates	571
Section 126.2: The basics	571
Section 126.3: Routing URLs	572
Section 126.4: HTTP Methods	573
Section 126.5: Jinja Templating	573
Section 126.6: The Request Object	574
<b>Chapter 127: Introduction to RabbitMQ using AMQPStorm</b>	576
Section 127.1: How to consume messages from RabbitMQ	576
Section 127.2: How to publish messages to RabbitMQ	577
Section 127.3: How to create a delayed queue in RabbitMQ	577
<b>Chapter 128: Descriptor</b>	580
Section 128.1: Simple descriptor	580
Section 128.2: Two-way conversions	581
<b>Chapter 129: tempfile NamedTemporaryFile</b>	582
Section 129.1: Create (and write to a) known, persistent temporary file	582
<b>Chapter 130: Input, Subset and Output External Data Files using Pandas</b>	584
Section 130.1: Basic Code to Import, Subset and Write External Data Files Using Pandas	584
<b>Chapter 131: Unzipping Files</b>	586
Section 131.1: Using Python ZipFile.extractall() to decompress a ZIP file	586
Section 131.2: Using Python TarFile.extractall() to decompress a tarball	586
<b>Chapter 132: Working with ZIP archives</b>	587
Section 132.1: Examining Zipfile Contents	587
Section 132.2: Opening Zip Files	587
Section 132.3: Extracting zip file contents to a directory	588
Section 132.4: Creating new archives	588
<b>Chapter 133: Getting start with GZip</b>	589
Section 133.1: Read and write GNU zip files	589

<b>第134章：栈</b>	590
第134.1节：使用列表对象创建栈类	590
第134.2节：解析括号	591
<b>第135章：绕过全局解释器锁（GIL）</b>	593
第135.1节：多处理.Pool	593
第135.2节：Cython nogil：	594
<b>第136章：部署</b>	595
第136.1节：上传Conda包	595
<b>第137章：日志记录</b>	597
第137.1节：Python日志记录简介	597
第137.2节：记录异常	598
<b>第138章：Web服务器网关接口（WSGI）</b>	601
第138.1节：服务器对象（方法）	601
<b>第139章：Python服务器发送事件</b>	602
第139.1节：Flask SSE	602
第139.2节：Asyncio SSE	602
<b>第140章：其他语言中switch语句的替代方案</b>	604
第140.1节：使用语言提供的功能：if/else结构	604
第140.2节：使用函数字典	604
第140.3节：使用类内省	605
第140.4节：使用上下文管理器	606
<b>第141章：列表解构（又称打包和解包）</b>	607
第141.1节：解构赋值	607
第141.2节：打包函数参数	608
第141.3节：解包函数参数	610
<b>第142章：访问Python源代码和字节码</b>	611
第142.1节：显示函数的字节码	611
第142.2节：显示对象的源代码	611
第142.3节：探索函数的代码对象	612
<b>第143章：混入（Mixins）</b>	613
第143.1节：Mixin（混入）	613
第143.2节：Mixin中的方法重写	614
<b>第144章：属性访问</b>	615
第144.1节：使用点符号的基本属性访问	615
第144.2节：设置器、获取器与属性	615
<b>第145章：ArcPy</b>	618
第145.1节：createDissolvedGDB 在工作空间中创建文件地理数据库	618
第145.2节：使用Search打印文件地理数据库中要素类所有行的一个字段值 游标	618
<b>第146章：抽象基类（abc）</b>	619
第146.1节：设置ABCMeta元类	619
第146.2节：为什么/如何使用ABCMeta和@abstractmethod	619
<b>第147章：插件和扩展类</b>	621
第147.1节：混入类	621
第147.2节：带有自定义类的插件	622
<b>第148章：不可变数据类型（int、float、str、tuple 和 frozenset）</b>	624
第148.1节：字符串的单个字符不可赋值	624
第148.2节：元组的单个成员不可赋值	624

<b>Chapter 134: Stack</b>	590
Section 134.1: Creating a Stack class with a List Object	590
Section 134.2: Parsing Parentheses	591
<b>Chapter 135: Working around the Global Interpreter Lock (GIL)</b>	593
Section 135.1: Multiprocessing.Pool	593
Section 135.2: Cython nogil:	594
<b>Chapter 136: Deployment</b>	595
Section 136.1: Uploading a Conda Package	595
<b>Chapter 137: Logging</b>	597
Section 137.1: Introduction to Python Logging	597
Section 137.2: Logging exceptions	598
<b>Chapter 138: Web Server Gateway Interface (WSGI)</b>	601
Section 138.1: Server Object (Method)	601
<b>Chapter 139: Python Server Sent Events</b>	602
Section 139.1: Flask SSE	602
Section 139.2: Asyncio SSE	602
<b>Chapter 140: Alternatives to switch statement from other languages</b>	604
Section 140.1: Use what the language offers: the if/else construct	604
Section 140.2: Use a dict of functions	604
Section 140.3: Use class introspection	605
Section 140.4: Using a context manager	606
<b>Chapter 141: List destructuring (aka packing and unpacking)</b>	607
Section 141.1: Destructuring assignment	607
Section 141.2: Packing function arguments	608
Section 141.3: Unpacking function arguments	610
<b>Chapter 142: Accessing Python source code and bytecode</b>	611
Section 142.1: Display the bytecode of a function	611
Section 142.2: Display the source code of an object	611
Section 142.3: Exploring the code object of a function	612
<b>Chapter 143: Mixins</b>	613
Section 143.1: Mixin	613
Section 143.2: Overriding Methods in Mixins	614
<b>Chapter 144: Attribute Access</b>	615
Section 144.1: Basic Attribute Access using the Dot Notation	615
Section 144.2: Setters, Getters & Properties	615
<b>Chapter 145: ArcPy</b>	618
Section 145.1: createDissolvedGDB to create a file gdb on the workspace	618
Section 145.2: Printing one field's value for all rows of feature class in file geodatabase using Search Cursor	618
<b>Chapter 146: Abstract Base Classes (abc)</b>	619
Section 146.1: Setting the ABCMeta metaclass	619
Section 146.2: Why/How to use ABCMeta and @abstractmethod	619
<b>Chapter 147: Plugin and Extension Classes</b>	621
Section 147.1: Mixins	621
Section 147.2: Plugins with Customized Classes	622
<b>Chapter 148: Immutable datatypes(int, float, str, tuple and frozensets)</b>	624
Section 148.1: Individual characters of strings are not assignable	624
Section 148.2: Tuple's individual members aren't assignable	624

<a href="#">第148.3节 : frozenset 是不可变的，且不可赋值</a>	624
<b>第149章：从 Python 2 迁移到 Python 3 的不兼容性</b>	625
<a href="#">第149.1节：整数除法</a>	625
<a href="#">第149.2节：解包可迭代对象</a>	626
<a href="#">第149.3节：字符串：字节与Unicode</a>	628
<a href="#">第149.4节：print语句与print函数</a>	630
<a href="#">第149.5节：range函数与xrange函数的区别</a>	631
<a href="#">第149.6节：引发和处理异常</a>	632
<a href="#">第149.7节：列表推导式中的变量泄漏</a>	634
<a href="#">第149.8节：True、False和None</a>	635
<a href="#">第149.9节：用户输入</a>	635
<a href="#">第149.10节：不同类型的比较</a>	636
<a href="#">第149.11节：迭代器上的.next()方法重命名</a>	636
<a href="#">第149.12节：filter()、map()和zip()返回迭代器而非序列</a>	637
<a href="#">第149.13节：重命名的模块</a>	637
<a href="#">第149.14节：移除了操作符&lt;&gt;和``，它们分别与!=和repr()同义</a>	638
<a href="#">第149.15节：long与int</a>	638
<a href="#">第149.16节：Python 3中的所有类都是“新式类”</a>	639
<a href="#">第149.17节：Reduce不再是内置函数</a>	640
<a href="#">第149.18节：绝对导入与相对导入</a>	640
<a href="#">第149.19节：map()函数</a>	642
<a href="#">第149.20节：round()函数的舍入规则及返回类型</a>	643
<a href="#">第149.21节：文件输入输出</a>	644
<a href="#">第149.22节：Python 3中移除了cmp函数</a>	644
<a href="#">第149.23节：八进制常量</a>	645
<a href="#">第149.24节：写入文件对象时的返回值</a>	645
<a href="#">第149.25节：exec语句在Python 3中变为函数</a>	645
<a href="#">第149.26节：不再支持encode/decode为十六进制</a>	646
<a href="#">第149.27节：字典方法的变更</a>	647
<a href="#">第149.28节：布尔值类</a>	647
<a href="#">第149.29节：Python 2中的hasattr函数错误</a>	648
<b>第150章：2to3工具</b>	650
<a href="#">第150.1节：基本用法</a>	650
<b>第151章：非官方Python实现</b>	652
<a href="#">第151.1节：IronPython</a>	652
<a href="#">第151.2节：Jython</a>	652
<a href="#">第151.3节：Transcrypt</a>	653
<b>第152章：抽象语法树</b>	656
<a href="#">第152.1节：分析Python脚本中的函数</a>	656
<b>第153章：Unicode与字节</b>	658
<a href="#">第153.1节：编码/解码错误处理</a>	658
<a href="#">第153.2节：文件输入输出</a>	658
<a href="#">第153.3节：基础知识</a>	659
<b>第154章：Python串口通信（pyserial）</b>	661
<a href="#">第154.1节：初始化串口设备</a>	661
<a href="#">第154.2节：从串口读取数据</a>	661
<a href="#">第154.3节：检查机器上可用的串口</a>	661
<b>第155章：使用Py2Neo的Neo4j和Cypher</b>	664
<a href="#">第155.1节：向Neo4j图添加节点</a>	664

<a href="#">Section 148.3: Frozenset's are immutable and not assignable</a>	624
<b>Chapter 149: Incompatibilities moving from Python 2 to Python 3</b>	625
<a href="#">Section 149.1: Integer Division</a>	625
<a href="#">Section 149.2: Unpacking Iterables</a>	626
<a href="#">Section 149.3: Strings: Bytes versus Unicode</a>	628
<a href="#">Section 149.4: Print statement vs. Print function</a>	630
<a href="#">Section 149.5: Differences between range and xrange functions</a>	631
<a href="#">Section 149.6: Raising and handling Exceptions</a>	632
<a href="#">Section 149.7: Leaked variables in list comprehension</a>	634
<a href="#">Section 149.8: True, False and None</a>	635
<a href="#">Section 149.9: User Input</a>	635
<a href="#">Section 149.10: Comparison of different types</a>	636
<a href="#">Section 149.11: .next() method on iterators renamed</a>	636
<a href="#">Section 149.12: filter(), map() and zip() return iterators instead of sequences</a>	637
<a href="#">Section 149.13: Renamed modules</a>	637
<a href="#">Section 149.14: Removed operators &lt;&gt; and ``, synonymous with != and repr()</a>	638
<a href="#">Section 149.15: long vs. int</a>	638
<a href="#">Section 149.16: All classes are "new-style classes" in Python 3</a>	639
<a href="#">Section 149.17: Reduce is no longer a built-in</a>	640
<a href="#">Section 149.18: Absolute/Relative Imports</a>	640
<a href="#">Section 149.19: map()</a>	642
<a href="#">Section 149.20: The round() function tie-breaking and return type</a>	643
<a href="#">Section 149.21: File I/O</a>	644
<a href="#">Section 149.22: cmp function removed in Python 3</a>	644
<a href="#">Section 149.23: Octal Constants</a>	645
<a href="#">Section 149.24: Return value when writing to a file object</a>	645
<a href="#">Section 149.25: exec statement is a function in Python 3</a>	645
<a href="#">Section 149.26: encode/decode to hex no longer available</a>	646
<a href="#">Section 149.27: Dictionary method changes</a>	647
<a href="#">Section 149.28: Class Boolean Value</a>	647
<a href="#">Section 149.29: hasattr function bug in Python 2</a>	648
<b>Chapter 150: 2to3 tool</b>	650
<a href="#">Section 150.1: Basic Usage</a>	650
<b>Chapter 151: Non-official Python implementations</b>	652
<a href="#">Section 151.1: IronPython</a>	652
<a href="#">Section 151.2: Jython</a>	652
<a href="#">Section 151.3: Transcrypt</a>	653
<b>Chapter 152: Abstract syntax tree</b>	656
<a href="#">Section 152.1: Analyze functions in a python script</a>	656
<b>Chapter 153: Unicode and bytes</b>	658
<a href="#">Section 153.1: Encoding/decoding error handling</a>	658
<a href="#">Section 153.2: File I/O</a>	658
<a href="#">Section 153.3: Basics</a>	659
<b>Chapter 154: Python Serial Communication (pyserial)</b>	661
<a href="#">Section 154.1: Initialize serial device</a>	661
<a href="#">Section 154.2: Read from serial port</a>	661
<a href="#">Section 154.3: Check what serial ports are available on your machine</a>	661
<b>Chapter 155: Neo4j and Cypher using Py2Neo</b>	664
<a href="#">Section 155.1: Adding Nodes to Neo4j Graph</a>	664

第155.2节：导入和认证	664
第155.3节：向Neo4j图添加关系	664
第155.4节：查询1：新闻标题自动完成	664
第155.5节：查询2：按特定日期和地点获取新闻文章	665
第155.6节：Cypher查询示例	665
<b>第156章：使用Python的基本Curses</b>	666
第156.1节：wrapper()辅助函数	666
第156.2节：基本调用示例	666
<b>第157章：Python中的模板</b>	667
第157.1节：使用模板的简单数据输出程序	667
第157.2节：更改分隔符	667
<b>第158章：Pillow</b>	668
第158.1节：读取图像文件	668
第158.2节：将文件转换为JPEG格式	668
<b>第159章：pass语句</b>	669
第159.1节：忽略异常	669
第159.2节：创建可捕获的新异常	669
<b>第160章：具有精确帮助输出的CLI子命令</b>	671
第160.1节：原生方式（无库）	671
第160.2节：argparse（默认帮助格式化器）	671
第160.3节：argparse（自定义帮助格式化器）	672
<b>第161章：数据库访问</b>	674
第161.1节：SQLite	674
第161.2节：使用MySQLdb访问MySQL数据库	679
第161.3节：连接	680
第161.4节：使用psycopg2访问PostgreSQL数据库	681
第161.5节：Oracle数据库	682
第161.6节：使用sqlalchemy	684
<b>第162章：将Python连接到SQL Server</b>	685
第162.1节：连接服务器，创建表，查询数据	685
<b>第163章：PostgreSQL</b>	686
第163.1节：入门	686
<b>第164章：Python与Excel</b>	687
第164.1节：使用xlrd模块读取Excel数据	687
第164.2节：使用xlsxwriter格式化Excel文件	687
第164.3节：将列表数据写入Excel文件	688
第164.4节：OpenPyXL	689
第164.5节：使用xlsxwriter创建Excel图表	689
<b>第165章：海龟绘图</b>	693
第165.1节：忍者扭转（海龟绘图）	693
<b>第166章：Python持久化</b>	694
第166.1节：Python持久化	694
第166.2节：保存和加载的函数工具	695
<b>第167章：设计模式</b>	696
第167.1节：设计模式简介与单例模式	696
第167.2节：策略模式	698
第167.3节：代理模式	699
<b>第168章：hashlib</b>	701

Section 155.2: Importing and Authenticating	664
Section 155.3: Adding Relationships to Neo4j Graph	664
Section 155.4: Query 1: Autocomplete on News Titles	664
Section 155.5: Query 2 : Get News Articles by Location on a particular date	665
Section 155.6: Cypher Query Samples	665
<b>Chapter 156: Basic Curses with Python</b>	666
Section 156.1: The wrapper() helper function	666
Section 156.2: Basic Invocation Example	666
<b>Chapter 157: Templates in python</b>	667
Section 157.1: Simple data output program using template	667
Section 157.2: Changing delimiter	667
<b>Chapter 158: Pillow</b>	668
Section 158.1: Read Image File	668
Section 158.2: Convert files to JPEG	668
<b>Chapter 159: The pass statement</b>	669
Section 159.1: Ignore an exception	669
Section 159.2: Create a new Exception that can be caught	669
<b>Chapter 160: CLI subcommands with precise help output</b>	671
Section 160.1: Native way (no libraries)	671
Section 160.2: argparse (default help formatter)	671
Section 160.3: argparse (custom help formatter)	672
<b>Chapter 161: Database Access</b>	674
Section 161.1: SQLite	674
Section 161.2: Accessing MySQL database using MySQLdb	679
Section 161.3: Connection	680
Section 161.4: PostgreSQL Database access using psycopg2	681
Section 161.5: Oracle database	682
Section 161.6: Using sqlalchemy	684
<b>Chapter 162: Connecting Python to SQL Server</b>	685
Section 162.1: Connect to Server, Create Table, Query Data	685
<b>Chapter 163: PostgreSQL</b>	686
Section 163.1: Getting Started	686
<b>Chapter 164: Python and Excel</b>	687
Section 164.1: Read the excel data using xlrd module	687
Section 164.2: Format Excel files with xlsxwriter	687
Section 164.3: Put list data into a Excel's file	688
Section 164.4: OpenPyXL	689
Section 164.5: Create excel charts with xlsxwriter	689
<b>Chapter 165: Turtle Graphics</b>	693
Section 165.1: Ninja Twist (Turtle Graphics)	693
<b>Chapter 166: Python Persistence</b>	694
Section 166.1: Python Persistence	694
Section 166.2: Function utility for save and load	695
<b>Chapter 167: Design Patterns</b>	696
Section 167.1: Introduction to design patterns and Singleton Pattern	696
Section 167.2: Strategy Pattern	698
Section 167.3: Proxy	699
<b>Chapter 168: hashlib</b>	701

第168.1节：字符串的MD5哈希	701
第168.2节：OpenSSL提供的算法	702
<b>第169章：使用Python创建Windows服务</b>	703
第169.1节：可以作为服务运行的Python脚本	703
第169.2节：将Flask Web应用作为服务运行	704
<b>第170章：Python中的可变与不可变（及可哈希）</b>	706
第170.1节：可变与不可变	706
第170.2节：作为参数的可变与不可变	708
<b>第171章：configparser</b>	710
第171.1节：以编程方式创建配置文件	710
第171.2节：基本用法	710
<b>第172章：光学字符识别</b>	711
第172.1节：PyTesseract	711
第172.2节：PyOCR	711
<b>第173章：虚拟环境</b>	713
第173.1节：创建和使用虚拟环境	713
第173.2节：在Unix/Linux脚本中指定使用的特定Python版本	715
第173.3节：为不同版本的Python创建虚拟环境	715
第173.4节：使用Anaconda创建虚拟环境	715
第173.5节：使用virtualenvwrapper管理多个虚拟环境	716
第173.6节：在虚拟环境中安装软件包	717
第173.7节：发现你正在使用哪个虚拟环境	718
第173.8节：检查是否在虚拟环境中运行	719
第173.9节：在fish shell中使用virtualenv	719
<b>第174章：Python虚拟环境 - virtualenv</b>	721
第174.1节：安装	721
第174.2节：使用	721
第174.3节：在你的虚拟环境中安装软件包	721
第174.4节：其他有用的virtualenv命令	722
<b>第175章：使用virtualenvwrapper的虚拟环境</b>	724
第175.1节：使用virtualenvwrapper创建虚拟环境	724
<b>第176章：在Windows中使用virtualenvwrapper创建虚拟环境</b>	726
第176.1节：Windows的virtualenvwrapper虚拟环境	726
<b>第177章：sys</b>	727
第177.1节：命令行参数	727
第177.2节：脚本名称	727
第177.3节：标准错误流	727
第177.4节：提前结束进程并返回退出代码	727
<b>第178章：ChemPy - Python包</b>	728
第178.1节：解析化学式	728
第178.2节：化学反应的计量平衡	728
第178.3节：平衡反应	728
第178.4节：化学平衡	729
第178.5节：离子强度	729
第178.6节：化学动力学（常微分方程系统）	729
<b>第179章：pygame</b>	731
第179.1节：Pygame的混音模块	731
第179.2节：安装pygame	732

Section 168.1: MD5 hash of a string	701
Section 168.2: algorithm provided by OpenSSL	702
<b>Chapter 169: Creating a Windows service using Python</b>	703
Section 169.1: A Python script that can be run as a service	703
Section 169.2: Running a Flask web application as a service	704
<b>Chapter 170: Mutable vs Immutable (and Hashable) in Python</b>	706
Section 170.1: Mutable vs Immutable	706
Section 170.2: Mutable and Immutable as Arguments	708
<b>Chapter 171: configparser</b>	710
Section 171.1: Creating configuration file programmatically	710
Section 171.2: Basic usage	710
<b>Chapter 172: Optical Character Recognition</b>	711
Section 172.1: PyTesseract	711
Section 172.2: PyOCR	711
<b>Chapter 173: Virtual environments</b>	713
Section 173.1: Creating and using a virtual environment	713
Section 173.2: Specifying specific python version to use in script on Unix/Linux	715
Section 173.3: Creating a virtual environment for a different version of python	715
Section 173.4: Making virtual environments using Anaconda	715
Section 173.5: Managing multiple virtual environments with virtualenvwrapper	716
Section 173.6: Installing packages in a virtual environment	717
Section 173.7: Discovering which virtual environment you are using	718
Section 173.8: Checking if running inside a virtual environment	719
Section 173.9: Using virtualenv with fish shell	719
<b>Chapter 174: Python Virtual Environment - virtualenv</b>	721
Section 174.1: Installation	721
Section 174.2: Usage	721
Section 174.3: Install a package in your Virtualenv	721
Section 174.4: Other useful virtualenv commands	722
<b>Chapter 175: Virtual environment with virtualenvwrapper</b>	724
Section 175.1: Create virtual environment with virtualenvwrapper	724
<b>Chapter 176: Create virtual environment with virtualenvwrapper in windows</b>	726
Section 176.1: Virtual environment with virtualenvwrapper for windows	726
<b>Chapter 177: sys</b>	727
Section 177.1: Command line arguments	727
Section 177.2: Script name	727
Section 177.3: Standard error stream	727
Section 177.4: Ending the process prematurely and returning an exit code	727
<b>Chapter 178: ChemPy - python package</b>	728
Section 178.1: Parsing formulae	728
Section 178.2: Balancing stoichiometry of a chemical reaction	728
Section 178.3: Balancing reactions	728
Section 178.4: Chemical equilibria	729
Section 178.5: Ionic strength	729
Section 178.6: Chemical kinetics (system of ordinary differential equations)	729
<b>Chapter 179: pygame</b>	731
Section 179.1: Pygame's mixer module	731
Section 179.2: Installing pygame	732

<b>第180章：Pyglet</b>	734
第180.1节：Pyglet的安装	734
第180.2节：Pyglet中的Hello World	734
第180.3节：在Pyglet中播放声音	734
第180.4节：使用Pyglet进行OpenGL编程	734
第180.5节：使用Pyglet和OpenGL绘制点	734
<b>第181章：音频</b>	736
第181.1节：处理WAV文件	736
第181.2节：使用Python和mpeg转换任意音频文件	736
第181.3节：播放Windows的提示音	736
第181.4节：使用Pyglet的音频	737
<b>第182章：pyaudio</b>	738
第182.1节：回调模式音频输入输出	738
第182.2节：阻塞模式音频输入输出	739
<b>第183章：shelve</b>	741
第183.1节：创建新的Shelf	741
第183.2节：货架示例代码	742
第183.3节：接口总结（键为字符串，数据为任意对象）：	742
第183.4节：写回	742
<b>第184章：使用Python和树莓派进行物联网编程</b>	744
第184.1节：示例 - 温度传感器	744
<b>第185章：kivy - 用于NUI开发的跨平台Python框架</b>	748
第185.1节：第一个应用程序	748
<b>第186章：Pandas转换：对分组执行操作并连接结果</b>	750
第186.1节：简单转换	750
第186.2节：每组多个结果	751
<b>第187章：语法相似，含义不同：Python与JavaScript</b>	752
第187.1节：列表中的`in`	752
<b>第188章：从C#调用Python</b>	753
第188.1节：由C#应用调用的Python脚本	753
第188.2节：调用Python脚本的C#代码	753
<b>第189章：ctypes</b>	755
第189.1节：ctypes数组	755
第189.2节：ctypes函数封装	755
第189.3节：基本用法	756
第189.4节：常见陷阱	756
第189.5节：基本的ctypes对象	757
第189.6节：复杂用法	758
<b>第190章：编写扩展</b>	760
第190.1节：使用C扩展的Hello World	760
第190.2节：使用C++和Boost的C扩展	760
第190.3节：向C扩展传递打开的文件	762
<b>第191章：Python Lex-Yacc</b>	763
第191.1节：PLY入门	763
第191.2节：PLY的“你好，世界！”——一个简单的计算器	763
第191.3节：第1部分：使用Lex对输入进行词法分析	765
第191.4节：第2部分：使用Yacc解析词法分析后的输入	768

<b>Chapter 180: Pyglet</b>	734
Section 180.1: Installation of Pyglet	734
Section 180.2: Hello World in Pyglet	734
Section 180.3: Playing Sound in Pyglet	734
Section 180.4: Using Pyglet for OpenGL	734
Section 180.5: Drawing Points Using Pyglet and OpenGL	734
<b>Chapter 181: Audio</b>	736
Section 181.1: Working with WAV files	736
Section 181.2: Convert any soundfile with python and ffmpeg	736
Section 181.3: Playing Windows' beeps	736
Section 181.4: Audio With Pyglet	737
<b>Chapter 182: pyaudio</b>	738
Section 182.1: Callback Mode Audio I/O	738
Section 182.2: Blocking Mode Audio I/O	739
<b>Chapter 183: shelve</b>	741
Section 183.1: Creating a new Shelf	741
Section 183.2: Sample code for shelve	742
Section 183.3: To summarize the interface (key is a string, data is an arbitrary object):	742
Section 183.4: Write-back	742
<b>Chapter 184: IoT Programming with Python and Raspberry PI</b>	744
Section 184.1: Example - Temperature sensor	744
<b>Chapter 185: kivy - Cross-platform Python Framework for NUI Development</b>	748
Section 185.1: First App	748
<b>Chapter 186: Pandas Transform: Preform operations on groups and concatenate the results</b>	750
Section 186.1: Simple transform	750
Section 186.2: Multiple results per group	751
<b>Chapter 187: Similarities in syntax, Differences in meaning: Python vs. JavaScript</b>	752
Section 187.1: `in` with lists	752
<b>Chapter 188: Call Python from C#</b>	753
Section 188.1: Python script to be called by C# application	753
Section 188.2: C# code calling Python script	753
<b>Chapter 189: ctypes</b>	755
Section 189.1: ctypes arrays	755
Section 189.2: Wrapping functions for ctypes	755
Section 189.3: Basic usage	756
Section 189.4: Common pitfalls	756
Section 189.5: Basic ctypes object	757
Section 189.6: Complex usage	758
<b>Chapter 190: Writing extensions</b>	760
Section 190.1: Hello World with C Extension	760
Section 190.2: C Extension Using c++ and Boost	760
Section 190.3: Passing an open file to C Extensions	762
<b>Chapter 191: Python Lex-Yacc</b>	763
Section 191.1: Getting Started with PLY	763
Section 191.2: The "Hello, World!" of PLY - A Simple Calculator	763
Section 191.3: Part 1: Tokenizing Input with Lex	765
Section 191.4: Parsing Tokenized Input with Yacc	768

<b>第192章：单元测试</b>	772	<b>Chapter 192: Unit Testing</b>	772
第192.1节：unittest.TestCase中的测试设置与拆卸	772	Section 192.1: Test Setup and Teardown within a unittest.TestCase	772
第192.2节：断言异常	772	Section 192.2: Asserting on Exceptions	772
第192.3节：测试异常	773	Section 192.3: Testing Exceptions	773
第192.4节：在单元测试中选择断言	774	Section 192.4: Choosing Assertions Within Unitests	774
第192.5节：使用pytest的单元测试	775	Section 192.5: Unit tests with pytest	775
第192.6节：使用unittest.mock.create_autospec模拟函数	778	Section 192.6: Mocking functions with unittest.mock.create_autospec	778
<b>第193章：py.test</b>	780	<b>Chapter 193: py.test</b>	780
第193.1节：设置py.test	780	Section 193.1: Setting up py.test	780
第193.2节：测试夹具简介	780	Section 193.2: Intro to Test Fixtures	780
第193.3节：失败的测试	783	Section 193.3: Failing Tests	783
<b>第194章：性能分析</b>	785	<b>Chapter 194: Profiling</b>	785
第194.1节：IPython中的%%timeit和%timeit	785	Section 194.1: %%timeit and %timeit in IPython	785
第194.2节：使用cProfile（首选性能分析器）	785	Section 194.2: Using cProfile (Preferred Profiler)	785
第194.3节：timeit()函数	785	Section 194.3: timeit() function	785
第194.4节：timeit命令行	786	Section 194.4: timeit command line	786
第194.5节：命令行中的line_profiler	786	Section 194.5: line_profiler in command line	786
<b>第195章：Python程序速度</b>	788	<b>Chapter 195: Python speed of program</b>	788
第195.1节：双端队列操作	788	Section 195.1: Deque operations	788
第195.2节：算法符号	788	Section 195.2: Algorithmic Notations	788
第195.3节：符号	789	Section 195.3: Notation	789
第195.4节：列表操作	790	Section 195.4: List operations	790
第195.5节：集合运算	790	Section 195.5: Set operations	790
<b>第196章：性能优化</b>	792	<b>Chapter 196: Performance optimization</b>	792
第196.1节：代码分析	792	Section 196.1: Code profiling	792
<b>第197章：安全与密码学</b>	794	<b>Chapter 197: Security and Cryptography</b>	794
第197.1节：安全密码哈希	794	Section 197.1: Secure Password Hashing	794
第197.2节：计算消息摘要	794	Section 197.2: Calculating a Message Digest	794
第197.3节：可用的哈希算法	794	Section 197.3: Available Hashing Algorithms	794
第197.4节：文件哈希	795	Section 197.4: File Hashing	795
第197.5节：使用pycrypto生成RSA签名	795	Section 197.5: Generating RSA signatures using pycrypto	795
第197.6节：使用pycrypto进行非对称RSA加密	796	Section 197.6: Asymmetric RSA encryption using pycrypto	796
第197.7节：使用pycrypto进行对称加密	797	Section 197.7: Symmetric encryption using pycrypto	797
<b>第198章：Python中的安全外壳连接</b>	798	<b>Chapter 198: Secure Shell Connection in Python</b>	798
第198.1节：ssh连接	798	Section 198.1: ssh connection	798
<b>第199章：Python反模式</b>	799	<b>Chapter 199: Python Anti-Patterns</b>	799
第199.1节：过度热心的except子句	799	Section 199.1: Overzealous except clause	799
第199.2节：在使用高处理器消耗函数时三思而后行	799	Section 199.2: Looking before you leap with processor-intensive function	799
<b>第200章：常见陷阱</b>	802	<b>Chapter 200: Common Pitfalls</b>	802
第200.1节：列表乘法与常见引用问题	802	Section 200.1: List multiplication and common references	802
第200.2节：可变的默认参数	805	Section 200.2: Mutable default argument	805
第200.3节：修改正在迭代的序列	806	Section 200.3: Changing the sequence you are iterating over	806
第200.4节：整数和字符串的标识	809	Section 200.4: Integer and String identity	809
第200.5节：字典是无序的	810	Section 200.5: Dictionaries are unordered	810
第200.6节：列表推导式和for循环中的变量泄漏	811	Section 200.6: Variable leaking in list comprehensions and for loops	811
第200.7节：或运算符的链式调用	811	Section 200.7: Chaining of or operator	811
第200.8节：sys.argv[0]是正在执行的文件名	812	Section 200.8: sys.argv[0] is the name of the file being executed	812
第200.9节：访问整数字面量的属性	812	Section 200.9: Accessing int literals' attributes	812
第200.10节：全局解释器锁（GIL）和阻塞线程	813	Section 200.10: Global Interpreter Lock (GIL) and blocking threads	813

<a href="#">第200.11节：多重返回</a>	814
<a href="#">第200.12节：Python风格的JSON键</a>	814
<b>第201章：隐藏功能</b>	816
<a href="#">第201.1节：运算符重载</a>	816
<a href="#">鸣谢</a>	817
<a href="#">你可能也喜欢</a>	831

<a href="#">Section 200.11: Multiple return</a>	814
<a href="#">Section 200.12: Pythonic JSON keys</a>	814
<b>Chapter 201: Hidden Features</b>	816
<a href="#">Section 201.1: Operator Overloading</a>	816
<a href="#">Credits</a>	817
<a href="#">You may also like</a>	831

欢迎随意免费分享此PDF，  
本书最新版本可从以下网址下载：

<https://GoalKicker.com/PythonBook>

本Python® 专业人士笔记一书汇编自[Stack Overflow Documentation](#)，内容由Stack Overflow的优秀贡献者撰写。  
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

本书为非官方免费教材，旨在教育用途，与官方Python®组织或公司及Stack Overflow无关。所有商标及注册商标均为其各自公司所有者所有。

本书所提供信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至[web@petercv.com](mailto:web@petercv.com)

Please feel free to share this PDF with anyone for free,  
latest version of this book can be downloaded from:

<https://GoalKicker.com/PythonBook>

This Python® Notes for Professionals book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow.  
Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Python® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to [web@petercv.com](mailto:web@petercv.com)

# 第1章：Python入门语言

## Python 3.x

### 版本 发布日期

<a href="#">3.8</a>	2020-04-29
<a href="#">3.7</a>	2018-06-27
<a href="#">3.6</a>	2016-12-23
<a href="#">3.5</a>	2015-09-13
<a href="#">3.4</a>	2014-03-17
<a href="#">3.3</a>	2012-09-29
<a href="#">3.2</a>	2011-02-20
<a href="#">3.1</a>	2009-06-26
<a href="#">3.0</a>	2008-12-03

## Python 2.x

### 版本 发布日期

<a href="#">2.7</a>	2010-07-03
<a href="#">2.6</a>	2008-10-02
<a href="#">2.5</a>	2006-09-19
<a href="#">2.4</a>	2004-11-30
<a href="#">2.3</a>	2003-07-29
<a href="#">2.2</a>	2001-12-21
<a href="#">2.1</a>	2001-04-15
<a href="#">2.0</a>	2000-10-16

## 第1.1节：入门

Python是一种广泛使用的高级通用编程语言，由吉多·范罗苏姆（Guido van Rossum）创建，首次发布于1991年。Python具有动态类型系统和自动内存管理，支持多种编程范式，包括面向对象、命令式、函数式和过程式风格。它拥有庞大且全面的标准库。

目前有两个主要版本的Python在积极使用中：

- Python 3.x是当前版本，正在积极开发中。
- Python 2.x 是遗留版本，直到2020年只会收到安全更新，不会实现任何新功能。请注意，虽然迁移到 Python 3 越来越容易，但许多项目仍在使用 Python 2。

您可以下载并安装任一版本的 Python [here](#)。请参见 [Python 3](#) 与 [Python 2](#) 的比较。此外，一些第三方提供了重新打包的 Python 版本，添加了常用库和其他功能，以简化数学、数据分析或科学用途等常见用例的设置。请参见官方站点上的 [the list at the official site](#)。

### 验证是否已安装 Python

要确认 Python 是否正确安装，您可以在喜欢的终端中运行以下命令进行验证（如果您使用的是 Windows 操作系统，需先将 Python 路径添加到环境变量，才能在命令提示符中使用）：

# Chapter 1: Getting started with Python Language

## Python 3.x

### Version Release Date

<a href="#">3.8</a>	2020-04-29
<a href="#">3.7</a>	2018-06-27
<a href="#">3.6</a>	2016-12-23
<a href="#">3.5</a>	2015-09-13
<a href="#">3.4</a>	2014-03-17
<a href="#">3.3</a>	2012-09-29
<a href="#">3.2</a>	2011-02-20
<a href="#">3.1</a>	2009-06-26
<a href="#">3.0</a>	2008-12-03

## Python 2.x

### Version Release Date

<a href="#">2.7</a>	2010-07-03
<a href="#">2.6</a>	2008-10-02
<a href="#">2.5</a>	2006-09-19
<a href="#">2.4</a>	2004-11-30
<a href="#">2.3</a>	2003-07-29
<a href="#">2.2</a>	2001-12-21
<a href="#">2.1</a>	2001-04-15
<a href="#">2.0</a>	2000-10-16

## Section 1.1: Getting Started

Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

Two major versions of Python are currently in active use:

- Python 3.x is the current version and is under active development.
- Python 2.x is the legacy version and will receive only security updates until 2020. No new features will be implemented. Note that many projects still use Python 2, although migrating to Python 3 is getting easier.

You can download and install either version of Python [here](#). See [Python 3 vs. Python 2](#) for a comparison between them. In addition, some third-parties offer re-packaged versions of Python that add commonly used libraries and other features to ease setup for common use cases, such as math, data analysis or scientific use. See [the list at the official site](#).

### Verify if Python is installed

To confirm that Python was installed correctly, you can verify that by running the following command in your favorite terminal (If you are using Windows OS, you need to add path of python to the environment variable before using it in command prompt):

```
$ python --version
```

Python 3.x 版本 ≥ 3.0

如果您已安装 Python 3，且它是您的默认版本（详情请参见 故障排除），您应该会看到类似如下内容：

```
$ python --version
```

Python 3.6.0

Python 2.x 版本 ≤ 2.7

如果你安装了Python 2，并且它是你的默认版本（详见故障排除），你应该会看到类似如下内容：

```
$ python --version
```

Python 2.7.13

如果你已经安装了 Python 3，但\$ python --version显示的是 Python 2 版本，说明你系统中也安装了 Python 2。这种情况在 MacOS 和许多 Linux 发行版中很常见。请使用\$ python3来明确调用 Python 3 解释器。

#### 使用 IDLE 编写的 Python 版本的“Hello, World”

IDLE是一个简单的 Python 编辑器，随 Python 一起捆绑提供。

#### 如何在 IDLE 中创建“Hello, World”程序

- 在你选择的系统上打开 IDLE。
  - 在较旧版本的 Windows 中，可以在 Windows 菜单下的所有程序中找到它。
  - 在 Windows 8 及以上版本中，搜索IDLE或在系统中已安装的应用中找到它。
  - 在基于 Unix 的系统（包括 Mac）上，可以通过在终端输入\$ idle python\_file. 来打开它。
- 它会打开一个顶部带有选项的交互式窗口。

在 shell 中，有一个由三个右尖括号组成的提示符：

```
>>>
```

现在在提示符中输入以下代码：

```
>>> print("Hello, World")
```

按下回车键 .

```
>>> print("Hello, World")
Hello, World
```

#### Hello World Python 文件

创建一个新文件 hello.py，内容包含以下代码行：

```
Python 3.x 版本 ≥ 3.0
```

```
print('Hello, World')
```

Python 2.x 版本 ≥ 2.6

```
$ python --version
```

Python 3.x Version ≥ 3.0

If you have Python 3 installed, and it is your default version (see **Troubleshooting** for more details) you should see something like this:

```
$ python --version
```

Python 3.6.0

Python 2.x Version ≤ 2.7

If you have Python 2 installed, and it is your default version (see **Troubleshooting** for more details) you should see something like this:

```
$ python --version
```

Python 2.7.13

If you have installed Python 3, but \$ python --version outputs a Python 2 version, you also have Python 2 installed. This is often the case on MacOS, and many Linux distributions. Use \$ python3 instead to explicitly use the Python 3 interpreter.

#### Hello, World in Python using IDLE

IDLE is a simple editor for Python, that comes bundled with Python.

#### How to create Hello, World program in IDLE

- Open IDLE on your system of choice.
  - In older versions of Windows, it can be found at All Programs under the Windows menu.
  - In Windows 8+, search for IDLE or find it in the apps that are present in your system.
  - On Unix-based (including Mac) systems you can open it from the shell by typing \$ idle python\_file.py.
- It will open a shell with options along the top.

In the shell, there is a prompt of three right angle brackets:

```
>>>
```

Now write the following code in the prompt:

```
>>> print("Hello, World")
```

Hit Enter .

```
>>> print("Hello, World")
Hello, World
```

#### Hello World Python file

Create a new file hello.py that contains the following line:

```
Python 3.x Version ≥ 3.0
```

```
print('Hello, World')
```

Python 2.x Version ≥ 2.6

你可以通过以下import语句在Python 2中使用Python 3的print函数：

```
from __future__ import print_function
```

Python 2有许多功能可以通过`__future__`

模块从Python 3中选择性导入，详见此处。

Python 2.x 版本 ≤ 2.7

如果使用Python 2，你也可以输入下面这行代码。注意这在Python 3中无效，因此不推荐使用，因为它降低了跨版本代码的兼容性。

```
print 'Hello, World'
```

在终端中，切换到包含文件hello.py的目录。

输入python hello.py，然后按下 **回车键** 键。

```
$ python hello.py  
Hello, World
```

你应该看到Hello, World打印到控制台。

你也可以用hello.py替换为你的文件路径。例如，如果你的文件在主目录中，且你的Linux用户名是"user"，你可以输入python /home/`user`/hello.py。

## 启动交互式Python shell

通过在终端执行（运行）python命令，你将进入一个交互式Python shell。这也被称为Python解释器或REPL（即“读取-求值-打印循环”）。

```
$ python  
Python 2.7.12 (default, 2016年6月28日, 08:46:01)  
[GCC 6.1.1 20160602] 在linux上  
输入"help"、"copyright"、"credits"或"license"获取更多信息。  
>>> print 'Hello, World'  
Hello, World  
>>>
```

如果你想从终端运行Python 3，执行命令python3。

```
$ python3  
Python 3.6.0 (default, 2017年1月13日, 00:00:00)  
[GCC 6.1.1 20160602] 在 linux 上  
输入"help"、"copyright"、"credits"或"license"获取更多信息。  
>>> print('Hello, World')  
Hello, World  
>>>
```

或者，启动交互式提示符并使用 `python -i <file.py>` 加载文件。

在命令行中运行：

```
$ python -i hello.py  
"Hello World"  
>>>
```

You can use the Python 3 `print` function in Python 2 with the following `import` statement:

```
from __future__ import print_function
```

Python 2 has a number of functionalities that can be optionally imported from Python 3 using the `__future__` module, as discussed here.

Python 2.x Version ≤ 2.7

If using Python 2, you may also type the line below. Note that this is not valid in Python 3 and thus not recommended because it reduces cross-version code compatibility.

```
print 'Hello, World'
```

In your terminal, navigate to the directory containing the file `hello.py`.

Type `python hello.py`, then hit the **Enter** key.

```
$ python hello.py  
Hello, World
```

You should see Hello, World printed to the console.

You can also substitute `hello.py` with the path to your file. For example, if you have the file in your home directory and your user is "user" on Linux, you can type `python /home/user/hello.py`.

## Launch an interactive Python shell

By executing (running) the `python` command in your terminal, you are presented with an interactive Python shell. This is also known as the [Python Interpreter](#) or a REPL (for 'Read Evaluate Print Loop').

```
$ python  
Python 2.7.12 (default, Jun 28 2016, 08:46:01)  
[GCC 6.1.1 20160602] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print 'Hello, World'  
Hello, World  
>>>
```

If you want to run Python 3 from your terminal, execute the command `python3`.

```
$ python3  
Python 3.6.0 (default, Jan 13 2017, 00:00:00)  
[GCC 6.1.1 20160602] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print('Hello, World')  
Hello, World  
>>>
```

Alternatively, start the interactive prompt and load file with `python -i <file.py>`.

In command line, run:

```
$ python -i hello.py  
"Hello World"  
>>>
```

关闭 Python 交互式环境有多种方法：

```
>>> exit()
```

或者

```
>>> quit()
```

或者，CTRL + D 将关闭 shell 并返回到终端的命令行。

如果你想取消正在输入的命令并返回到干净的命令提示符，同时保持在解释器 shell 内，请使用CTRL + C。

[试试在线交互式 Python shell。](#)

## 其他在线 Shell

各种网站提供在线访问 Python shell 的服务。

在线 shell 可能适用于以下用途：

- 在没有安装 Python 的设备（智能手机、平板等）上运行一小段代码。
- 学习或教授基础Python。
- 解决在线评测题目。

示例：

免责声明：文档作者与以下列出的任何资源无关。

- <https://www.python.org/shell/> - 官方Python网站托管的在线Python交互环境。
- <https://ideone.com/> - 网络上广泛使用的代码片段演示平台。
- <https://repl.it/languages/python3> - 功能强大且简单的在线编译器、集成开发环境和解释器。可编写、编译、并运行Python代码。
- [https://www.tutorialspoint.com/execute\\_python\\_online.php](https://www.tutorialspoint.com/execute_python_online.php) - 功能齐全的UNIX shell，且带有用户友好的项目浏览器。
- [http://rextester.com/l/python3\\_online\\_compiler](http://rextester.com/l/python3_online_compiler) - 简单易用的IDE，显示执行时间。

## 以字符串形式运行命令

Python可以在交互环境中以字符串形式传入任意代码：

```
$ python -c 'print("Hello, World")'  
Hello, World
```

这在将多个脚本的结果在 shell 中连接时非常有用。

## Shell 及其扩展

包管理 - PyPA 推荐的安装 Python 包的工具是 PIP。要安装，在你的命令行执行 `pip install <包名>`。例如，`pip install numpy`。（注意：在 Windows 上必须将 pip 添加到 PATH 环境变量。为避免此问题，可使用 `python -m pip install <包名>`）

Shell - 到目前为止，我们讨论了使用 Python 原生交互式 shell 运行代码的不同方式。Shell 利用

There are multiple ways to close the Python shell:

```
>>> exit()
```

or

```
>>> quit()
```

Alternatively, CTRL + D will close the shell and put you back on your terminal's command line.

If you want to cancel a command you're in the middle of typing and get back to a clean command prompt, while staying inside the Interpreter shell, use CTRL + C.

[Try an interactive Python shell online.](#)

## Other Online Shells

Various websites provide online access to Python shells.

Online shells may be useful for the following purposes:

- Run a small code snippet from a machine which lacks python installation(smartphones, tablets etc).
- Learn or teach basic Python.
- Solve online judge problems.

Examples:

Disclaimer: documentation author(s) are not affiliated with any resources listed below.

- <https://www.python.org/shell/> - The online Python shell hosted by the official Python website.
- <https://ideone.com/> - Widely used on the Net to illustrate code snippet behavior.
- <https://repl.it/languages/python3> - Powerful and simple online compiler, IDE and interpreter. Code, compile, and run code in Python.
- [https://www.tutorialspoint.com/execute\\_python\\_online.php](https://www.tutorialspoint.com/execute_python_online.php) - Full-featured UNIX shell, and a user-friendly project explorer.
- [http://rextester.com/l/python3\\_online\\_compiler](http://rextester.com/l/python3_online_compiler) - Simple and easy to use IDE which shows execution time

## Run commands as a string

Python can be passed arbitrary code as a string in the shell:

```
$ python -c 'print("Hello, World")'  
Hello, World
```

This can be useful when concatenating the results of scripts together in the shell.

## Shells and Beyond

Package Management - The PyPA recommended tool for installing Python packages is [PIP](#). To install, on your command line execute `pip install <the package name>`. For instance, `pip install numpy`. (Note: On windows you must add pip to your PATH environment variables. To avoid this, use `python -m pip install <the package name>`)

Shells - So far, we have discussed different ways to run code using Python's native interactive shell. Shells use

Python 的解释能力实时实验代码。其他 shell 包括 IDLE - 一个预装的图形界面，IPython - 以扩展交互体验著称，等等。

程序 - 为了长期保存，你可以将内容保存为 .py 文件，并使用外部工具如 shell、IDE（例如 PyCharm）、Jupyter 笔记本等编辑/执行它们。中级用户可能会使用这些工具；不过这里讨论的方法足以入门。

[Python 教程](#) 允许你逐步执行 Python 代码，从而可视化程序的执行流程，帮助你理解程序出错的原因。

[PEP8](#) 定义了 Python 代码格式的指导原则。良好的代码格式有助于你快速理解代码的功能。

## 第1.2节：创建变量并赋值

在Python中创建变量，只需指定变量名，然后赋值即可。

```
<variable name> = <value>
```

Python 使用 = 来给变量赋值。无需提前声明变量（或为其指定数据类型），给变量赋值的同时即声明并初始化该变量。无法声明一个没有初始值的变量。

```
# 整数
a = 2
print(a)
# 输出: 2

# 整数
b = 9223372036854775807
print(b)
# 输出: 9223372036854775807
```

```
# 浮点数
pi = 3.14
print(pi)
# 输出: 3.14
```

```
# 字符串
c = 'A'
print(c)
# 输出: A
```

```
# 字符串
name = 'John Doe'
print(name)
# 输出: John Doe
```

```
# 布尔值
q = True
print(q)
# 输出: True
```

```
# 空值或空数据类型
x = None
print(x)
# 输出: None
```

Python's interpretive power for experimenting with code real-time. Alternative shells include [IDLE](#) - a pre-bundled GUI, [IPython](#) - known for extending the interactive experience, etc.

Programs - For long-term storage you can save content to .py files and edit/execute them as scripts or programs with external tools e.g. shell, IDEs (such as [PyCharm](#), [Jupyter notebooks](#), etc. Intermediate users may use these tools; however, the methods discussed here are sufficient for getting started.

[Python tutor](#) allows you to step through Python code so you can visualize how the program will flow, and helps you to understand where your program went wrong.

[PEP8](#) defines guidelines for formatting Python code. Formatting code well is important so you can quickly read what the code does.

## Section 1.2: Creating variables and assigning values

To create a variable in Python, all you need to do is specify the variable name, and then assign a value to it.

```
<variable name> = <value>
```

Python uses = to assign values to variables. There's no need to declare a variable in advance (or to assign a data type to it), assigning a value to a variable itself declares and initializes the variable with that value. There's no way to declare a variable without assigning it an initial value.

```
# Integer
a = 2
print(a)
# Output: 2

# Integer
b = 9223372036854775807
print(b)
# Output: 9223372036854775807
```

```
# Floating point
pi = 3.14
print(pi)
# Output: 3.14
```

```
# String
c = 'A'
print(c)
# Output: A
```

```
# String
name = 'John Doe'
print(name)
# Output: John Doe
```

```
# Boolean
q = True
print(q)
# Output: True
```

```
# Empty value or null data type
x = None
print(x)
# Output: None
```

变量赋值是从左到右进行的。因此，以下代码会给你一个语法错误。

```
0 = x  
=> 输出: 语法错误: 不能将值赋给字面量
```

你不能使用 Python 的关键字作为有效的变量名。你可以通过以下方式查看关键字列表：

```
import keyword  
print(keyword.kwlist)
```

变量命名规则：

1. 变量名必须以字母或下划线开头。

```
x = True # 有效  
_y = True # 有效  
  
9x = False # 以数字开头  
=> 语法错误: 无效语法  
  
$y = False # 以符号开头  
=> 语法错误: 无效语法
```

2. 变量名的其余部分可以由字母、数字和下划线组成。

```
has_0_in_it = "仍然有效"
```

3. 名称区分大小写。

```
x = 9  
y = X*5  
=>NameError: name 'X' is not defined
```

尽管在Python中声明变量时无需指定数据类型，但在为变量分配内存所需空间时，Python解释器会自动选择最合适的内置类型：

```
a = 2  
print(type(a))  
# 输出: <type 'int'>  
  
b = 9223372036854775807  
print(type(b))  
# 输出: <type 'int'>  
  
pi = 3.14  
print(type(pi))  
# 输出: <type 'float'>  
  
c = 'A'  
print(type(c))  
# 输出: <type 'str'>  
  
name = 'John Doe'  
print(type(name))  
# 输出: <type 'str'>  
  
q = True  
print(type(q))
```

Variable assignment works from left to right. So the following will give you an syntax error.

```
0 = x  
=> Output: SyntaxError: can't assign to literal
```

You can not use python's keywords as a valid variable name. You can see the list of keyword by:

```
import keyword  
print(keyword.kwlist)
```

Rules for variable naming:

1. Variables names must start with a letter or an underscore.

```
x = True # valid  
_y = True # valid  
  
9x = False # starts with numeral  
=> SyntaxError: invalid syntax  
  
$y = False # starts with symbol  
=> SyntaxError: invalid syntax
```

2. The remainder of your variable name may consist of letters, numbers and underscores.

```
has_0_in_it = "Still Valid"
```

3. Names are case sensitive.

```
x = 9  
y = X*5  
=>NameError: name 'X' is not defined
```

Even though there's no need to specify a data type when declaring a variable in Python, while allocating the necessary area in memory for the variable, the Python interpreter automatically picks the most suitable built-in type for it:

```
a = 2  
print(type(a))  
# Output: <type 'int'>  
  
b = 9223372036854775807  
print(type(b))  
# Output: <type 'int'>  
  
pi = 3.14  
print(type(pi))  
# Output: <type 'float'>  
  
c = 'A'  
print(type(c))  
# Output: <type 'str'>  
  
name = 'John Doe'  
print(type(name))  
# Output: <type 'str'>  
  
q = True  
print(type(q))
```

```
# 输出: <type 'bool'>
```

```
x = None  
print(type(x))  
# 输出: <type 'NoneType'>
```

现在你已经了解了赋值的基础知识，让我们来解决一下Python赋值中的这个细微差别。

当你使用=进行赋值操作时，=左边的是右边**对象的一个名称**。最终，=的作用是将右边对象的**引用**赋给左边的**名称**。

也就是说：

```
a_name = an_object # "a_name"现在是对象"an_object"引用的名称
```

所以，从上面许多赋值的例子中，如果我们选择pi = 3.14，那么pi是对象3.14的一个名称（不是唯一名称，因为一个对象可以有多个名称）。如果你下面有不理解的地方，可以回到这里再读一遍！另外，你也可以看看this以获得更好的理解。

你可以在一行中给多个变量赋多个值。注意，=操作符左右两边的参数数量必须相同：

```
a, b, c = 1, 2, 3  
print(a, b, c)  
# 输出: 1 2 3
```

```
a, b, c = 1, 2  
=> 追踪 (最近一次调用最后) :  
=> 文件 "name.py", 第 N 行, 在 <模块>  
=> a, b, c = 1, 2  
=> ValueError: 需要超过 2 个值来解包
```

```
a, b = 1, 2, 3  
=> 追踪 (最近一次调用最后) :  
=> 文件 "name.py", 第 N 行, 在 <模块>  
=> a, b = 1, 2, 3  
=> ValueError: 解包的值太多
```

最后一个示例中的错误可以通过将剩余的值赋给相同数量的任意变量来避免。

这个哑变量可以有任何名称，但通常使用下划线（\_）来赋值不需要的值：

```
a, b, _ = 1, 2, 3  
print(a, b)  
# 输出: 1, 2
```

请注意，\_的数量和剩余值的数量必须相等。否则会抛出如上所示的“too many values to unpack error”错误：

```
a, b, _ = 1, 2, 3, 4  
=>Traceback (most recent call last):  
=>File "name.py", line N, in <module>  
=>a, b, _ = 1, 2, 3, 4  
=>ValueError: too many values to unpack (expected 3)
```

你也可以同时将一个值赋给多个变量。

```
# Output: <type 'bool'>
```

```
x = None  
print(type(x))  
# Output: <type 'NoneType'>
```

Now you know the basics of assignment, let's get this subtlety about assignment in python out of the way.

When you use = to do an assignment operation, what's on the left of = is a **name** for the **object** on the right. Finally, what = does is assign the **reference** of the object on the right to the **name** on the left.

That is:

```
a_name = an_object # "a_name" is now a name for the reference to the object "an_object"
```

So, from many assignment examples above, if we pick pi = 3.14, then pi is a name (not the name, since an object can have multiple names) for the object 3.14. If you don't understand something below, come back to this point and read this again! Also, you can take a look at [this](#) for a better understanding.

You can assign multiple values to multiple variables in one line. Note that there must be the same number of arguments on the right and left sides of the = operator:

```
a, b, c = 1, 2, 3  
print(a, b, c)  
# Output: 1 2 3
```

```
a, b, c = 1, 2  
=> Traceback (most recent call last):  
=> File "name.py", line N, in <module>  
=> a, b, c = 1, 2  
=> ValueError: need more than 2 values to unpack
```

```
a, b = 1, 2, 3  
=> Traceback (most recent call last):  
=> File "name.py", line N, in <module>  
=> a, b = 1, 2, 3  
=> ValueError: too many values to unpack
```

The error in last example can be obviated by assigning remaining values to equal number of arbitrary variables. This dummy variable can have any name, but it is conventional to use the underscore (\_) for assigning unwanted values:

```
a, b, _ = 1, 2, 3  
print(a, b)  
# Output: 1, 2
```

Note that the number of \_ and number of remaining values must be equal. Otherwise 'too many values to unpack error' is thrown as above:

```
a, b, _ = 1, 2, 3, 4  
=>Traceback (most recent call last):  
=>File "name.py", line N, in <module>  
=>a, b, _ = 1, 2, 3, 4  
=>ValueError: too many values to unpack (expected 3)
```

You can also assign a single value to several variables simultaneously.

```
a = b = c = 1  
print(a, b, c)  
# 输出: 1 1 1
```

使用这种级联赋值时，重要的是要注意变量 `a`、`b` 和 `c` 都引用内存中同一个值为 1 的 `int` 对象。换句话说，`a`、`b` 和 `c` 是同一个 `int` 对象的三个不同名称。随后给其中一个赋予不同的对象不会改变其他变量，这正是预期的行为：

```
a = b = c = 1    # 三个名字 a, b 和 c 都指向值为 1 的同一个整数对象  
print(a, b, c)  
# 输出: 1 1 1  
b = 2            # b 现在指向另一个整数对象，值为 2  
print(a, b, c)  
# 输出: 1 2 1 # 因此输出符合预期。
```

上述情况对于可变类型（如 `list`、`dict` 等）和不可变类型（如 `int`、`string`、`tuple` 等）同样适用：

```
x = y = [7, 8, 9]    # x 和 y 指向刚创建的同一个列表对象 [7, 8, 9]  
x = [13, 8, 9]        # x 现在指向另一个新创建的列表对象 [13, 8, 9]  
print(y)                # y 仍然指向最初赋值的列表  
# 输出: [7, 8, 9]
```

到目前为止一切正常。当使用级联赋值处理可变类型时，修改对象（与将名字赋值给不同对象不同，我们上面做的是赋值）情况会有所不同。请看下面，你会亲眼见到：

```
x = y = [7, 8, 9]    # x 和 y 是同一个刚创建的列表对象 [7, 8, 9] 的两个不同名字  
x[0] = 13            # 我们通过其中一个名字 x 更新列表 [7, 8, 9] 的值  
print(y)                # 使用列表的另一个名称打印其值  
# 输出: [13, 8, 9] # 因此，变化自然得以反映
```

嵌套列表在 Python 中也是有效的。这意味着一个列表可以包含另一个列表作为元素。

```
x = [1, 2, [3, 4, 5], 6, 7] # 这是一个嵌套列表  
print x[2]  
# 输出: [3, 4, 5]  
print x[2][1]  
# 输出: 4
```

最后，Python 中的变量类型不必保持最初定义时的类型——你可以简单地使用 `=` 来给变量赋新值，即使该值是不同类型的。

```
a = 2  
print(a)  
# 输出: 2  
  
a = "New value"  
print(a)  
# 输出: New value
```

如果这让你感到困惑，想想看，`=` 左边的只是对象的一个名字。起初你用值为 2 的 `int` 对象命名为 `a`，后来你改变主意，决定把名字 `a` 赋给一个值为 'New value' 的 `string` 对象。很简单，对吧？

```
a = b = c = 1  
print(a, b, c)  
# Output: 1 1 1
```

When using such cascading assignment, it is important to note that all three variables `a`, `b` and `c` refer to the same object in memory, an `int` object with the value of 1. In other words, `a`, `b` and `c` are three different names given to the same `int` object. Assigning a different object to one of them afterwards doesn't change the others, just as expected:

```
a = b = c = 1    # all three names a, b and c refer to same int object with value 1  
print(a, b, c)  
# Output: 1 1 1  
b = 2            # b now refers to another int object, one with a value of 2  
print(a, b, c)  
# Output: 1 2 1 # so output is as expected.
```

The above is also true for mutable types (like `list`, `dict`, etc.) just as it is true for immutable types (like `int`, `string`, `tuple`, etc.):

```
x = y = [7, 8, 9]    # x and y refer to the same list object just created, [7, 8, 9]  
x = [13, 8, 9]        # x now refers to a different list object just created, [13, 8, 9]  
print(y)                # y still refers to the list it was first assigned  
# Output: [7, 8, 9]
```

So far so good. Things are a bit different when it comes to modifying the object (in contrast to assigning the name to a different object, which we did above) when the cascading assignment is used for mutable types. Take a look below, and you will see it first hand:

```
x = y = [7, 8, 9]    # x and y are two different names for the same list object just created, [7,  
8, 9]  
x[0] = 13            # we are updating the value of the list [7, 8, 9] through one of its names, x  
in this case  
print(y)                # printing the value of the list using its other name  
# Output: [13, 8, 9] # hence, naturally the change is reflected
```

Nested lists are also valid in python. This means that a list can contain another list as an element.

```
x = [1, 2, [3, 4, 5], 6, 7] # this is nested list  
print x[2]  
# Output: [3, 4, 5]  
print x[2][1]  
# Output: 4
```

Lastly, variables in Python do not have to stay the same type as which they were first defined -- you can simply use `=` to assign a new value to a variable, even if that value is of a different type.

```
a = 2  
print(a)  
# Output: 2  
  
a = "New value"  
print(a)  
# Output: New value
```

If this bothers you, think about the fact that what's on the left of `=` is just a name for an object. First you call the `int` object with value 2 `a`, then you change your mind and decide to give the name `a` to a `string` object, having value 'New value'. Simple, right?

## 第1.3节：块缩进

Python 使用缩进来定义控制和循环结构。这有助于提高 Python 的可读性，然而，它要求程序员密切注意空白的使用。因此，编辑器校准错误可能导致代码表现出意想不到的行为。

Python 使用冒号符号 (:) 和缩进来表示代码块的开始和结束（如果你来自其他语言，不要将此与三元运算符混淆）。也就是说，Python 中的代码块，如函数、循环、if 条件语句及其他结构，没有结束标识符。所有代码块以冒号开始，随后包含其下方缩进的代码行。

例如：

```
def my_function():    # 这是一个函数定义。注意冒号 (:)
    a = 2            # 这行属于函数，因为它是缩进的
    return a          # 这行也属于同一个函数
print(my_function()) # 这行在函数块之外
```

或者

```
if a > b:           # if 块从这里开始
    print(a)          # 这是 if 块的一部分
else:                # else 必须与 if 保持同一级别
    print(b)          # 这行是 else 块的一部分
```

包含恰好一条单行语句的代码块可以写在同一行，尽管这种写法通常不被认为是良好风格：

```
if a > b: print(a)
else: print(b)
```

尝试用多条语句写在同一行将无法实现：

```
if x > y: y = x
    print(y) # IndentationError: unexpected indent

if x > y: while y != z: y -= 1 # SyntaxError: invalid syntax
```

空代码块会导致IndentationError。当代码块没有内容时，使用pass（一个不执行任何操作的命令）：

```
def will_be_implemented_later():
    pass
```

### 空格与制表符

简而言之：始终使用4个空格进行缩进。

完全使用制表符也是可能的，但PEP 8 ([Python代码的风格指南](#)) 指出推荐使用空格。

Python 3.x 版本 ≥ 3.0

Python 3 不允许混合使用制表符和空格进行缩进。若出现这种情况，会产生编译时错误：Inconsistent use of tabs and spaces in indentation，程序将无法运行。

Python 2.x 版本 ≤ 2.7

## Section 1.3: Block Indentation

Python uses indentation to define control and loop constructs. This contributes to Python's readability, however, it requires the programmer to pay close attention to the use of whitespace. Thus, editor miscalibration could result in code that behaves in unexpected ways.

Python uses the colon symbol (:) and indentation for showing where blocks of code begin and end (If you come from another language, do not confuse this with somehow being related to the [ternary operator](#)). That is, blocks in Python, such as functions, loops, if clauses and other constructs, have no ending identifiers. All blocks start with a colon and then contain the indented lines below it.

For example:

```
def my_function():    # This is a function definition. Note the colon (:)
    a = 2            # This line belongs to the function because it's indented
    return a          # This line also belongs to the same function
print(my_function()) # This line is OUTSIDE the function block
```

or

```
if a > b:           # If block starts here
    print(a)          # This is part of the if block
else:                # else must be at the same level as if
    print(b)          # This line is part of the else block
```

Blocks that contain exactly one single-line statement may be put on the same line, though this form is generally not considered good style:

```
if a > b: print(a)
else: print(b)
```

Attempting to do this with more than a single statement will *not* work:

```
if x > y: y = x
    print(y) # IndentationError: unexpected indent

if x > y: while y != z: y -= 1 # SyntaxError: invalid syntax
```

An empty block causes an IndentationError. Use `pass` (a command that does nothing) when you have a block with no content:

```
def will_be_implemented_later():
    pass
```

### Spaces vs. Tabs

In short: **always** use 4 spaces for indentation.

Using tabs exclusively is possible but [PEP 8](#), the style guide for Python code, states that spaces are preferred.

Python 3.x Version ≥ 3.0

Python 3 disallows mixing the use of tabs and spaces for indentation. In such case a compile-time error is generated: Inconsistent use of tabs **and** spaces **in** indentation and the program will not run.

Python 2.x Version ≤ 2.7

Python 2 允许在缩进中混合使用制表符和空格；但强烈不建议这样做。制表符字符会将之前的缩进补足为8个空格的倍数。由于编辑器通常配置为将制表符显示为4个空格，这可能导致细微的错误。

引用PEP 8：

当使用-t选项调用Python 2命令行解释器时，会对非法混合使用制表符和空格的代码发出警告。使用-tt时，这些警告会变成错误。强烈推荐使用这些选项！

许多编辑器都有“制表符转空格”的配置。在配置编辑器时，应区分制表符字符 ('') 和 制表符键。

- 制表符字符应配置为显示8个空格，以符合语言语义——至少在可能出现（意外）混合缩进的情况下。编辑器也可以自动将制表符字符转换为空格。
- 但是，将编辑器配置为按下 制表符键时插入4个空格，而不是插入制表符字符，可能会更有帮助。

用混合制表符和空格编写的Python源代码，或使用非标准缩进空格数的代码，可以使用autopep8使其符合pep8规范。（大多数Python安装中还附带一个功能较弱的替代工具：[reindent.py](#)）

## 第1.4节：数据类型

内置类型

布尔值

bool：布尔值，取值为True或False。可以对布尔值执行逻辑运算，如and、or、not。

```
x or y    # 如果 x 为假，则为 y, 否则为 x  
x and y   # 如果 x 为假，则为 x, 否则为 y  
not x     # 如果 x 为真，则为假, 否则为真
```

在 Python 2.x 和 Python 3.x 中，布尔值也是一个int。类型bool是int类型的子类，True和False是它的唯一实例：

```
issubclass(bool, int) # True  
  
isinstance(True, bool) # True  
isinstance(False, bool) # True
```

如果布尔值用于算术运算，将使用它们的整数值（True为1，False为0）来返回整数结果：

```
True + False == 1 # 1 + 0 == 1  
True * True == 1 # 1 * 1 == 1
```

数字

- int：整数

Python 2 allows mixing tabs and spaces in indentation; this is strongly discouraged. The tab character completes the previous indentation to be a [multiple of 8 spaces](#). Since it is common that editors are configured to show tabs as multiple of 4 spaces, this can cause subtle bugs.

Citing [PEP 8](#):

When invoking the Python 2 command line interpreter with the -t option, it issues warnings about code that illegally mixes tabs and spaces. When using -tt these warnings become errors. These options are highly recommended!

Many editors have "tabs to spaces" configuration. When configuring the editor, one should differentiate between the tab *character* ('\t') and the `Tab` key.

- The tab *character* should be configured to show 8 spaces, to match the language semantics - at least in cases when (accidental) mixed indentation is possible. Editors can also automatically convert the tab character to spaces.
- However, it might be helpful to configure the editor so that pressing the `Tab` key will insert 4 spaces, instead of inserting a tab character.

Python source code written with a mix of tabs and spaces, or with non-standard number of indentation spaces can be made pep8-conformant using [autopep8](#). (A less powerful alternative comes with most Python installations: [reindent.py](#))

## Section 1.4: Datatypes

Built-in Types

Booleans

`bool`: A boolean value of either `True` or `False`. Logical operations like `and`, `or`, `not` can be performed on booleans.

```
x or y    # if x is False then y otherwise x  
x and y   # if x is False then x otherwise y  
not x     # if x is True then False, otherwise True
```

In Python 2.x and in Python 3.x, a boolean is also an `int`. The `bool` type is a subclass of the `int` type and `True` and `False` are its only instances:

```
issubclass(bool, int) # True  
  
isinstance(True, bool) # True  
isinstance(False, bool) # True
```

If boolean values are used in arithmetic operations, their integer values (1 and 0 for `True` and `False`) will be used to return an integer result:

```
True + False == 1 # 1 + 0 == 1  
True * True == 1 # 1 * 1 == 1
```

Numbers

- `int`: Integer number

```
a = 2
b = 100
c = 123456789
d = 38563846326424324
```

Python 中的整数大小是任意的。

注意：在较早版本的Python中，存在一种long类型，它与int不同。两者已被统一。

- float：浮点数；精度取决于实现和系统架构，对于CPython，float数据类型对应于C语言的double类型。

```
a = 2.0
b = 100.e0
c = 123456789.e1
```

- complex：复数

```
a = 2 + 1j
b = 100 + 10j
```

当任一操作数为复数时，<、<=、>和>=运算符将引发TypeError异常。

## 字符串

Python 3.x 版本  $\geq$  3.0

- str：一个**Unicode字符串**，类型为 'hello'
- bytes：一个字节字符串。类型为 b'hello'

Python 2.x 版本  $\leq$  2.7

- str：一个**字节字符串**。类型为 'hello'
- bytes：str的同义词
- unicode：a **unicode** 字符串。类型为 u'hello'

## 序列和集合

Python 区分有序序列和无序集合（例如set和dict）。

- 字符串 (str、bytes、unicode) 是序列
- reversed：使用reversed函数对str进行反转顺序

```
a = reversed('hello')
```

- 元组：一个有序的 n 个任意类型值的集合 ( $n \geq 0$ )。

```
a = (1, 2, 3)
b = ('a', 1, 'python', (1, 2))
b[2] = 'something else' # 返回 TypeError
```

支持索引；不可变；如果所有成员都是可哈希的，则自身可哈希

```
a = 2
b = 100
c = 123456789
d = 38563846326424324
```

Integers in Python are of arbitrary sizes.

Note: in older versions of Python, a **long** type was available and this was distinct from **int**. The two have been unified.

- float：Floating point number; precision depends on the implementation and system architecture, for CPython the **float** datatype corresponds to a C double.

```
a = 2.0
b = 100.e0
c = 123456789.e1
```

- complex：Complex numbers

```
a = 2 + 1j
b = 100 + 10j
```

The <、<=、> 和 >= 运算符将引发TypeError异常 when any operand is a complex number.

## Strings

Python 3.x Version  $\geq$  3.0

- str：a **unicode string**. The type of 'hello'
- bytes：a **byte string**. The type of b'hello'

Python 2.x Version  $\leq$  2.7

- str：a **byte string**. The type of 'hello'
- bytes：synonym for str
- unicode：a **unicode string**. The type of u'hello'

## Sequences and collections

Python differentiates between ordered sequences and unordered collections (such as set and dict).

- strings (str, bytes, unicode) are sequences
- reversed：A reversed order of str with reversed function

```
a = reversed('hello')
```

- tuple：An ordered collection of n values of any type ( $n \geq 0$ ).

```
a = (1, 2, 3)
b = ('a', 1, 'python', (1, 2))
b[2] = 'something else' # returns a TypeError
```

Supports indexing; immutable; hashable if all its members are hashable

- 列表：一个有序的 n 个值的集合 (n >= 0)

```
a = [1, 2, 3]
b = ['a', 1, 'python', (1, 2), [1, 2]]
b[2] = 'something else' # 允许
```

不可哈希；可变的。

- set：一个无序的唯一值集合。元素必须是可哈希的。

```
a = {1, 2, 'a'}
```

- dict：一个无序的唯一键值对集合；键必须是可哈希的。

```
a = {1: 'one',
      2: 'two'}
b = {'a': [1, 2, 3],
      'b': 'a string'}
```

如果一个对象在其生命周期内具有不变的哈希值（需要有`__hash__()`方法），并且可以与其他对象比较（需要有`__eq__()`方法），则该对象是可哈希的。可哈希对象在比较相等时必须具有相同的哈希值。

## 内置常量

内置数据类型之外，内置命名空间中还有少量内置常量：

- `True`：内置类型`bool`的真值
- `False`：内置类型`bool`的假值
- `None`：用于表示值缺失的单例对象。
- `Ellipsis`或...：在核心Python3+中任何地方使用，在Python2.7+中作为数组表示法的一部分有限使用。`numpy`及相关包将其用作数组中的“包含所有”引用。
- `NotImplemented`：一个单例，用于向Python指示某个特殊方法不支持特定参数，Python会尝试其他可用的替代方案。

```
a = None # 不会赋值。以后可以赋予任何有效的数据类型
```

Python 3.x 版本 ≥ 3.0

`None`没有自然的顺序。使用顺序比较运算符 (<、<=、>=、>) 不再支持，并会引发`TypeError`。

Python 2.x 版本 ≤ 2.7

`None`总是小于任何数字 (`None < -32` 结果为`True`)。

## 变量类型测试

在Python中，我们可以使用内置函数`type`来检查对象的数据类型。

```
a = '123'
print(type(a))
```

- `list`: An ordered collection of n values (n >= 0)

```
a = [1, 2, 3]
b = ['a', 1, 'python', (1, 2), [1, 2]]
b[2] = 'something else' # allowed
```

Not hashable; mutable.

- `set`: An unordered collection of unique values. Items must be [hashable](#).

```
a = {1, 2, 'a'}
```

- `dict`: An unordered collection of unique key-value pairs; keys must be [hashable](#).

```
a = {1: 'one',
      2: 'two'}
```

```
b = {'a': [1, 2, 3],
      'b': 'a string'}
```

An object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equality must have the same hash value.

## Built-in constants

In conjunction with the built-in datatypes there are a small number of built-in constants in the built-in namespace:

- `True`: The true value of the built-in type `bool`
- `False`: The false value of the built-in type `bool`
- `None`: A singleton object used to signal that a value is absent.
- `Ellipsis` or ...: used in core Python3+ anywhere and limited usage in Python2.7+ as part of array notation. `numpy` and related packages use this as a 'include everything' reference in arrays.
- `NotImplemented`: a singleton used to indicate to Python that a special method doesn't support the specific arguments, and Python will try alternatives if available.

```
a = None # No value will be assigned. Any valid datatype can be assigned later
```

Python 3.x Version ≥ 3.0

`None` doesn't have any natural ordering. Using ordering comparison operators (<、<=、>=、>) isn't supported anymore and will raise a `TypeError`.

Python 2.x Version ≤ 2.7

`None` is always less than any number (`None < -32` evaluates to `True`).

## Testing the type of variables

In python, we can check the datatype of an object using the built-in function `type`.

```
a = '123'
print(type(a))
```

```
# Out: <class 'str'>
b = 123
print(type(b))
# Out: <class 'int'>
```

在条件语句中，可以使用`isinstance`来测试数据类型。然而，通常不鼓励依赖变量的类型。

```
i = 7
if isinstance(i, int):
    i += 1
elif isinstance(i, str):
    i = int(i)
    i += 1
```

有关`type()`和`isinstance()`之间差异的信息，请阅读：[Python 中`isinstance`和`type`的区别](#)

测试某个对象是否为`NoneType`：

```
x = None
if x is None:
    print('不意外，我刚刚将 x 定义为 None。')
```

## 数据类型转换

你可以执行显式的数据类型转换。

例如，'123' 是`str`类型，可以使用`int`函数将其转换为整数。

```
a = '123'
b = int(a)
```

可以使用`float`函数将类似'123.456'的浮点数字符串转换为浮点数。

```
a = '123.456'
b = float(a)
c = int(a)      # ValueError: invalid literal for int() with base 10: '123.456'
d = int(b)      # 123
```

你也可以转换序列或集合类型

```
a = 'hello'
list(a)  # ['h', 'e', 'l', 'l', 'o']
set(a)  # {'o', 'e', 'l', 'h'}
tuple(a) # ('h', 'e', 'l', 'l', 'o')
```

## 字面量定义时的显式字符串类型

在引号前加一个字母标签，可以告诉你想定义哪种类型的字符串。

- `b'foo bar'`: 在 Python 3 中结果为`bytes`，在 Python 2 中为`str`
- `u'foo bar'`: 在 Python 3 中结果为`str`，在 Python 2 中为`unicode`
- `'foo bar'`: 结果为`str`
- `r'foo bar'`: 结果为所谓的原始字符串，不需要转义特殊字符，所有内容都按你输入的字面意思处理

```
normal = 'foobar' # foo
```

```
# Out: <class 'str'>
b = 123
print(type(b))
# Out: <class 'int'>
```

In conditional statements it is possible to test the datatype with`isinstance`. However, it is usually not encouraged to rely on the type of the variable.

```
i = 7
if isinstance(i, int):
    i += 1
elif isinstance(i, str):
    i = int(i)
    i += 1
```

For information on the differences between`type()` and`isinstance()` read: [Differences between`isinstance` and`type` in Python](#)

To test if something is of`NoneType`:

```
x = None
if x is None:
    print('Not a surprise, I just defined x as None.')
```

## Converting between datatypes

You can perform explicit datatype conversion.

For example, '123' is of`str` type and it can be converted to integer using`int`function.

```
a = '123'
b = int(a)
```

Converting from a float string such as '123.456' can be done using`float`function.

```
a = '123.456'
b = float(a)
c = int(a)      # ValueError: invalid literal for int() with base 10: '123.456'
d = int(b)      # 123
```

You can also convert sequence or collection types

```
a = 'hello'
list(a)  # ['h', 'e', 'l', 'l', 'o']
set(a)  # {'o', 'e', 'l', 'h'}
tuple(a) # ('h', 'e', 'l', 'l', 'o')
```

## Explicit string type at definition of literals

With one letter labels just in front of the quotes you can tell what type of string you want to define.

- `b'foo bar'`: results`bytes`in Python 3,`str`in Python 2
- `u'foo bar'`: results`str`in Python 3,`unicode`in Python 2
- `'foo bar'`: results`str`
- `r'foo bar'`: results so called raw string, where escaping special characters is not necessary, everything is taken verbatim as you typed

```
normal = 'foo\nbar' # foo
```

```
# bar  
escaped = 'foo\nbar' # foobar raw  
= r'foobar' # foobar
```

## 可变和不可变数据类型

如果一个对象可以被更改，则称其为可变的。例如，当你将一个列表传递给某个函数时，列表可以被更改：

```
def f(m):  
    m.append(3) # 向列表添加一个数字。这是一次变更。  
  
x = [1, 2]  
f(x)  
x == [1, 2] # 现在为False, 因为列表中添加了一个元素
```

如果一个对象在任何情况下都不能被更改，则称其为不可变的。例如，整数是不可变的，因为没有办法更改它们：

```
def bar():  
    x = (1, 2)  
    g(x)  
x == (1, 2) # 总是为真, 因为没有函数能改变对象 (1, 2)
```

注意，变量本身是可变的，所以我们可以重新赋值变量 `x`，但这并不会改变 `x`之前指向的对象。它只是让 `x` 指向了一个新对象。

其实例是可变的类型称为可变数据类型，类似地，不可变对象和数据类型也是如此。

### 不可变数据类型示例：

- `int, long, float, complex`
- `str`
- `bytes`
- `tuple`
- `frozenset`

### 可变数据类型示例：

- `bytearray`
- `list`
- `set`
- 字典

## 第1.5节：集合类型

Python中有多种集合类型。虽然像`int`和`str`这样的类型只包含单个值，集合类型则包含多个值。

### 列表

`list`类型可能是Python中最常用的集合类型。尽管名字叫列表，但列表更像是其他语言中（主要是JavaScript）的数组。在Python中，列表只是一个有序的有效Python值集合。列表可以通过用方括号括起用逗号分隔的值来创建：

```
# bar  
escaped = 'foo\nbar' # foobar raw  
raw      = r'foo\nbar' # foobar
```

## Mutable and Immutable Data Types

An object is called *mutable* if it can be changed. For example, when you pass a list to some function, the list can be changed:

```
def f(m):  
    m.append(3) # adds a number to the list. This is a mutation.  
  
x = [1, 2]  
f(x)  
x == [1, 2] # False now, since an item was added to the list
```

An object is called *immutable* if it cannot be changed in any way. For example, integers are immutable, since there's no way to change them:

```
def bar():  
    x = (1, 2)  
    g(x)  
x == (1, 2) # Will always be True, since no function can change the object (1, 2)
```

Note that **variables** themselves are mutable, so we can reassign the *variable* `x`, but this does not change the object that `x` had previously pointed to. It only made `x` point to a new object.

Data types whose instances are mutable are called *mutable data types*, and similarly for immutable objects and datatypes.

### Examples of immutable Data Types:

- `int, long, float, complex`
- `str`
- `bytes`
- `tuple`
- `frozenset`

### Examples of mutable Data Types:

- `bytearray`
- `list`
- `set`
- `dict`

## Section 1.5: Collection Types

There are a number of collection types in Python. While types such as `int` and `str` hold a single value, collection types hold multiple values.

### Lists

The `list` type is probably the most commonly used collection type in Python. Despite its name, a list is more like an array in other languages, mostly JavaScript. In Python, a list is merely an ordered collection of valid Python values. A list can be created by enclosing values, separated by commas, in square brackets:

```
int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
```

列表可以是空的：

```
empty_list = []
```

列表的元素不限于单一数据类型，这符合Python作为动态语言的特点：

```
mixed_list = [1, 'abc', True, 2.34, None]
```

列表可以包含另一个列表作为其元素：

```
nested_list = [['a', 'b', 'c'], [1, 2, 3]]
```

列表的元素可以通过索引访问，索引是它们位置的数字表示。Python中的列表是从零开始索引，意味着列表中的第一个元素索引为0，第二个元素索引为1，依此类推：

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
print(names[0]) # Alice
print(names[2]) # Craig
```

索引也可以是负数，表示从列表末尾开始计数（-1 是最后一个元素的索引）。所以，使用上面例子中的列表：

```
print(names[-1]) # Eric
print(names[-4]) # Bob
```

列表是可变的，因此你可以更改列表中的值：

```
names[0] = 'Ann'
print(names)
# 输出 ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
```

此外，可以向列表中添加和/或移除元素：

使用`L.append(object)`将对象追加到列表末尾，返回值为`None`。

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names)
# 输出 ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

在列表的指定索引处添加新元素。`L.insert(index, object)`

```
names.insert(1, "Nikki")
print(names)
# 输出 ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

使用`L.remove(value)`移除第一个出现的指定值，返回值为`None`

```
names.remove("Bob")
print(names) # 输出 ['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
```

```
int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
```

A list can be empty:

```
empty_list = []
```

The elements of a list are not restricted to a single data type, which makes sense given that Python is a dynamic language:

```
mixed_list = [1, 'abc', True, 2.34, None]
```

A list can contain another list as its element:

```
nested_list = [['a', 'b', 'c'], [1, 2, 3]]
```

The elements of a list can be accessed via an *index*, or numeric representation of their position. Lists in Python are *zero-indexed* meaning that the first element in the list is at index 0, the second element is at index 1 and so on:

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
print(names[0]) # Alice
print(names[2]) # Craig
```

Indices can also be negative which means counting from the end of the list (-1 being the index of the last element). So, using the list from the above example:

```
print(names[-1]) # Eric
print(names[-4]) # Bob
```

Lists are mutable, so you can change the values in a list:

```
names[0] = 'Ann'
print(names)
# Outputs ['Ann', 'Bob', 'Craig', 'Diana', 'Eric']
```

Besides, it is possible to add and/or remove elements from a list:

Append object to end of list with `L.append(object)`, returns `None`.

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names)
# Outputs ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Add a new element to list at a specific index. `L.insert(index, object)`

```
names.insert(1, "Nikki")
print(names)
# Outputs ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia']
```

Remove the first occurrence of a value with `L.remove(value)`, returns `None`

```
names.remove("Bob")
print(names) # Outputs ['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia']
```

获取列表中第一个值为 `x` 的元素的索引。如果没有该元素，将显示错误。

```
name.index("Alice")  
0
```

计算列表长度

```
len(names)  
6
```

统计列表中某个元素的出现次数

```
a = [1, 1, 1, 2, 3, 4]  
a.count(1)  
3
```

反转列表

```
a.reverse()  
[4, 3, 2, 1, 1, 1]  
# 或者  
a[::-1]  
[4, 3, 2, 1, 1, 1]
```

使用 `L.pop([index])` 移除并返回指定索引处的元素（默认为最后一个元素），返回该元素

```
names.pop() # 输出 'Sia'
```

你可以像下面这样遍历列表元素：

```
for element in my_list:  
    print (element)
```

元组

元组 (tuple) 类似于列表，但它是固定长度且不可变的。因此，元组中的值不能被更改，也不能向元组中添加或删除值。元组通常用于不需要更改的小型值集合，例如IP地址和端口。元组用圆括号表示，而不是方括号：

```
ip_address = ('10.20.30.40', 8080)
```

列表的索引规则同样适用于元组。元组也可以嵌套，且值可以是任何有效的Python值。

只有一个成员的元组必须这样定义（注意逗号）：

```
one_member_tuple = ('Only member',)
```

或者

```
one_member_tuple = 'Only member', # 无括号
```

或者直接使用 `tuple` 语法

Get the index in the list of the first item whose value is `x`. It will show an error if there is no such item.

```
name.index("Alice")  
0
```

Count length of list

```
len(names)  
6
```

count occurrence of any item in list

```
a = [1, 1, 1, 2, 3, 4]  
a.count(1)  
3
```

Reverse the list

```
a.reverse()  
[4, 3, 2, 1, 1, 1]  
# or  
a[::-1]  
[4, 3, 2, 1, 1, 1]
```

Remove and return item at index (defaults to the last item) with `L.pop([index])`, returns the item

```
names.pop() # Outputs 'Sia'
```

You can iterate over the list elements like below:

```
for element in my_list:  
    print (element)
```

Tuples

A `tuple` is similar to a list except that it is fixed-length and immutable. So the values in the tuple cannot be changed nor the values be added to or removed from the tuple. Tuples are commonly used for small collections of values that will not need to change, such as an IP address and port. Tuples are represented with parentheses instead of square brackets:

```
ip_address = ('10.20.30.40', 8080)
```

The same indexing rules for lists also apply to tuples. Tuples can also be nested and the values can be any valid Python valid.

A tuple with only one member must be defined (note the comma) this way:

```
one_member_tuple = ('Only member',)
```

or

```
one_member_tuple = 'Only member', # No brackets
```

or just using `tuple` syntax

```
one_member_tuple = tuple(['Only member'])
```

## 字典

Python中的字典是一组键值对的集合。字典用花括号括起来。每对键值之间用逗号分隔，键和值之间用冒号分隔。下面是一个例子：

```
state_capitals = {  
    '阿肯色州': '小石城',  
    '科罗拉多州': '丹佛',  
    '加利福尼亚州': '萨克拉门托',  
    '乔治亚州': '亚特兰大'  
}
```

要获取一个值，可以通过其键来引用：

```
ca_capital = state_capitals['加利福尼亚州']
```

你也可以获取字典中的所有键，然后对它们进行迭代：

```
for k in state_capitals.keys():  
    print('{} 是 {} 的首府'.format(state_capitals[k], k))
```

字典与 JSON 语法非常相似。Python 标准库中的原生 json 模块可以用来在 JSON 和字典之间进行转换。

## set

set 是一个不包含重复元素且无插入顺序但有排序顺序的元素集合。它们用于只需将某些元素分组而不关心它们的插入顺序的场景。对于大量数据，检查一个元素是否在 set 中的速度远快于在 list 中进行相同操作。

定义一个 set 与定义一个 dictionary 非常相似：

```
first_names = {'亚当', '贝丝', '查理'}
```

或者你可以使用现有的列表来构建一个集合：

```
my_list = [1, 2, 3]  
my_set = set(my_list)
```

使用in检查集合中的成员资格：

```
if name in first_names:  
    print(name)
```

你可以像遍历列表一样遍历集合，但请记住：值的顺序是任意的，由实现决定的。

## defaultdict

defaultdict 是一种带有默认值的字典，因此对于没有显式定义值的键也可以访问而不会报错。 defaultdict 特别适用于字典中的值是集合（列表、字典等）的情况，因为在使用新键时不需要每次都初始化。

```
one_member_tuple = tuple(['Only member'])
```

## Dictionaries

A dictionary in Python is a collection of key-value pairs. The dictionary is surrounded by curly braces. Each pair is separated by a comma and the key and value are separated by a colon. Here is an example:

```
state_capitals = {  
    'Arkansas': 'Little Rock',  
    'Colorado': 'Denver',  
    'California': 'Sacramento',  
    'Georgia': 'Atlanta'  
}
```

To get a value, refer to it by its key:

```
ca_capital = state_capitals['California']
```

You can also get all of the keys in a dictionary and then iterate over them:

```
for k in state_capitals.keys():  
    print('{} is the capital of {}'.format(state_capitals[k], k))
```

Dictionaries strongly resemble JSON syntax. The native json module in the Python standard library can be used to convert between JSON and dictionaries.

## set

A set is a collection of elements with no repeats and without insertion order but sorted order. They are used in situations where it is only important that some things are grouped together, and not what order they were included. For large groups of data, it is much faster to check whether or not an element is in a set than it is to do the same for a list.

Defining a set is very similar to defining a dictionary:

```
first_names = {'Adam', 'Beth', 'Charlie'}
```

Or you can build a set using an existing list:

```
my_list = [1, 2, 3]  
my_set = set(my_list)
```

Check membership of the set using in:

```
if name in first_names:  
    print(name)
```

You can iterate over a set exactly like a list, but remember: the values will be in an arbitrary, implementation-defined order.

## defaultdict

A defaultdict is a dictionary with a default value for keys, so that keys for which no value has been explicitly defined can be accessed without errors. defaultdict is especially useful when the values in the dictionary are collections (lists, dicts, etc) in the sense that it does not need to be initialized every time when a new key is used.

defaultdict 永远不会引发 KeyError。任何不存在的键都会返回默认值。

例如，考虑以下字典

```
>>> state_capitals = {  
    '阿肯色州': '小石城',  
    '科罗拉多州': '丹佛',  
    '加利福尼亚州': '萨克拉门托',  
    '乔治亚州': '亚特兰大'  
}
```

如果我们尝试访问一个不存在的键，Python 会返回如下错误

```
>>> state_capitals['阿拉巴马州']  
追溯 (最近一次调用最后) :  
  
文件 "<ipython-input-61-236329695e6f>", 第 1 行, 在 <模块>  
state_capitals['阿拉巴马州']  
  
KeyError: '阿拉巴马州'
```

让我们尝试使用 defaultdict。它位于 collections 模块中。

```
>>> from collections import defaultdict  
>>> state_capitals = defaultdict(lambda: '波士顿')
```

我们这里所做的设置一个默认值（波士顿），以防给定的键不存在。现在像之前一样填充字典：

```
>>> state_capitals['阿肯色州'] = '小石城'  
>>> state_capitals['加利福尼亚州'] = '萨克拉门托'  
>>> state_capitals['科罗拉多州'] = '丹佛'  
>>> state_capitals['乔治亚州'] = '亚特兰大'
```

如果我们尝试使用一个不存在的键访问字典，Python 会返回默认值，即波士顿

```
>>> state_capitals['Alabama']  
'Boston'
```

并且对于存在的键，会像普通字典一样返回对应的值

```
>>> state_capitals['Arkansas']  
'Little Rock'
```

## 第1.6节：IDLE - Python图形用户界面

IDLE 是 Python 的集成开发与学习环境，是命令行的替代方案。顾名思义，IDLE 非常适合开发新代码或学习 Python。在 Windows 系统中，它随 Python 解释器一起提供，但在其他操作系统中，可能需要通过包管理器安装。

IDLE 的主要用途包括：

- 带有语法高亮、自动补全和智能缩进的多窗口文本编辑器
- 带有语法高亮的 Python 交互式命令行
- 集成调试器，支持单步执行、持久断点和调用栈可视化
- 自动缩进（对初学者学习 Python 缩进非常有用）

A defaultdict will never raise a KeyError. Any key that does not exist gets the default value returned.

For example, consider the following dictionary

```
>>> state_capitals = {  
    'Arkansas': 'Little Rock',  
    'Colorado': 'Denver',  
    'California': 'Sacramento',  
    'Georgia': 'Atlanta'  
}
```

If we try to access a non-existent key, python returns us an error as follows

```
>>> state_capitals['Alabama']  
Traceback (most recent call last):  
  
File "<ipython-input-61-236329695e6f>", line 1, in <module>  
    state_capitals['Alabama']  
  
KeyError: 'Alabama'
```

Let us try with a defaultdict. It can be found in the collections module.

```
>>> from collections import defaultdict  
>>> state_capitals = defaultdict(lambda: 'Boston')
```

What we did here is to set a default value (**Boston**) in case the give key does not exist. Now populate the dict as before:

```
>>> state_capitals['Arkansas'] = 'Little Rock'  
>>> state_capitals['California'] = 'Sacramento'  
>>> state_capitals['Colorado'] = 'Denver'  
>>> state_capitals['Georgia'] = 'Atlanta'
```

If we try to access the dict with a non-existent key, python will return us the default value i.e. Boston

```
>>> state_capitals['Alabama']  
'Boston'
```

and returns the created values for existing key just like a normal dictionary

```
>>> state_capitals['Arkansas']  
'Little Rock'
```

## Section 1.6: IDLE - Python GUI

IDLE is Python's Integrated Development and Learning Environment and is an alternative to the command line. As the name may imply, IDLE is very useful for developing new code or learning python. On Windows this comes with the Python interpreter, but in other operating systems you may need to install it through your package manager.

The main purposes of IDLE are:

- Multi-window text editor with syntax highlighting, autocompletion, and smart indent
- Python shell with syntax highlighting
- Integrated debugger with stepping, persistent breakpoints, and call stack visibility
- Automatic indentation (useful for beginners learning about Python's indentation)

- 将Python程序保存为.py文件，并使用IDLE运行和编辑它们。

在 IDLE 中，按F5或运行 Python Shell以启动解释器。使用 IDLE 对新用户来说可能是更好的学习体验，因为代码会随着用户的编写而被解释执行。

请注意，有许多替代方案，例如参见此讨论或此列表。

## 故障排除

### • Windows

如果你使用的是 Windows，默认命令是python。如果收到“python' 不是内部或外部命令，也不是可运行的程序或批处理文件”错误，最可能的原因是 Python 的位置未添加到系统的PATH环境变量中。你可以通过右键点击“我的电脑”并选择“属性”，或者通过“控制面板”进入“系统”来访问该设置。点击“高级系统设置”，然后点击“环境变量...”。编辑PATH变量，添加你的 Python 安装目录以及 Scripts 文件夹（通常是

C:\Python27;C:\Python27\Scripts）。这需要管理员权限，可能还需要重启。

当在同一台机器上使用多个版本的 Python 时，一种可能的解决方案是重命名其中一个 python.exe文件。例如，将一个版本命名为python27.exe，则python27将成为该版本的 Python 命令。

你也可以使用 Windows 的 Python 启动器，该启动器随安装程序默认安装。它允许你通过使用py -[x.y]而不是python[x.y]来选择运行的 Python 版本。你可以通过py -2运行脚本来使用最新的 Python 2 版本，通过py -3运行脚本来使用最新的 Python 3 版本。

### • Debian/Ubuntu/MacOS

本节假设python可执行文件的位置已添加到PATH环境变量中。

如果你使用的是Debian/Ubuntu/MacOS，打开终端并输入python（Python 2.x）或python3（Python 3.x）。

输入which python查看将使用哪个Python解释器。

### • Arch Linux

Arch Linux（及其衍生版）上的默认Python是Python 3，因此使用python或python3来运行Python 3.x，使用python2来运行Python 2.x。

### • 其他系统

Python 3有时绑定为python而不是python3。要在这些系统上使用已安装的Python 2，可以使用python2。

- Saving the Python program as .py files and run them and edit them later at any them using IDLE.

In IDLE, hit F5 or run Python Shell to launch an interpreter. Using IDLE can be a better learning experience for new users because code is interpreted as the user writes.

Note that there are lots of alternatives, see for example [this discussion](#) or [this list](#).

## Troubleshooting

### • Windows

If you're on Windows, the default command is python. If you receive a " 'python' is not recognized" error, the most likely cause is that Python's location is not in your system's PATH environment variable. This can be accessed by right-clicking on 'My Computer' and selecting 'Properties' or by navigating to 'System' through 'Control Panel'. Click on 'Advanced system settings' and then 'Environment Variables...'. Edit the PATH variable to include the directory of your Python installation, as well as the Script folder (usually C:\Python27;C:\Python27\Scripts). This requires administrative privileges and may require a restart.

When using multiple versions of Python on the same machine, a possible solution is to rename one of the python.exe files. For example, naming one version python27.exe would cause python27 to become the Python command for that version.

You can also use the Python Launcher for Windows, which is available through the installer and comes by default. It allows you to select the version of Python to run by using py -[x.y] instead of python[x.y]. You can use the latest version of Python 2 by running scripts with py -2 and the latest version of Python 3 by running scripts with py -3.

### • Debian/Ubuntu/MacOS

This section assumes that the location of the python executable has been added to the PATH environment variable.

If you're on Debian/Ubuntu/MacOS, open the terminal and type python for Python 2.x or python3 for Python 3.x.

Type which python to see which Python interpreter will be used.

### • Arch Linux

The default Python on Arch Linux (and descendants) is Python 3, so use python or python3 for Python 3.x and python2 for Python 2.x.

### • Other systems

Python 3 sometimes binds to python instead of python3. To use Python 2 on these systems where it is installed, you can use python2.

## 第1.7节：用户输入

### 交互式输入

要从用户获取输入，使用`input`函数（note：在Python 2.x中，该函数称为`raw_input`，尽管Python 2.x有自己版本的`input`，且完全不同）：

Python 2.x 版本 ≥ 2.3

```
name = raw_input("你叫什么名字？")  
# 输出：你叫什么名字？
```

**安全提示 不要在 Python2 中使用 `input()`，因为输入的文本会被当作**

Python 表达式进行求值（相当于 Python3 中的 `eval(input())`），这很容易导致安全漏洞。  
有关使用此函数风险的更多信息，请参见这篇文章。

Python 3.x 版本 ≥ 3.0

```
name = input("你叫什么名字？")  
# 输出：你叫什么名字？
```

以下示例将使用 Python 3 语法。

该函数接受一个字符串参数，将其显示为提示，并返回一个字符串。上述代码提供了一个提示，等待用户输入。

```
name = input("你叫什么名字？")  
# 输出：你叫什么名字？
```

如果用户输入“Bob”并按回车，变量 `name` 将被赋值为字符串 “Bob”：

```
name = input("你叫什么名字？")  
# 输出：你叫什么名字？ Bob  
print(name)  
# 输出：Bob
```

请注意，`input` 始终是 `str` 类型，如果你希望用户输入数字，这一点非常重要。因此，在尝试将其作为数字使用之前，你需要先将 `str` 转换为数字：

```
x = input("请输入一个数字:")  
# 输出：请输入一个数字：10  
x / 2  
# 输出：TypeError: unsupported operand type(s) for /: 'str' and 'int'  
float(x) / 2  
# 输出：5.0
```

注意：建议使用 `try/except` 代码块来捕获处理用户输入时的异常。例如，如果你的代码想将 `raw_input` 转换为 `int`，而用户输入的内容无法转换，则会引发 `ValueError`。

## 第1.8节：内置模块和函数

模块是包含Python定义和语句的文件。函数是一段执行某些逻辑的代码。

```
>>> pow(2,3) #8
```

## Section 1.7: User Input

### Interactive input

To get input from the user, use the `input` function (note: in Python 2.x, the function is called `raw_input` instead, although Python 2.x has its own version of `input` that is completely different):

Python 2.x Version ≥ 2.3

```
name = raw_input("What is your name? ")  
# Out: What is your name? _
```

**Security Remark** Do not use `input()` in Python2 - the entered text will be evaluated as if it were a Python expression (equivalent to `eval(input())` in Python3), which might easily become a vulnerability. See [this article](#) for further information on the risks of using this function.

Python 3.x Version ≥ 3.0

```
name = input("What is your name? ")  
# Out: What is your name? _
```

The remainder of this example will be using Python 3 syntax.

The function takes a string argument, which displays it as a prompt and returns a string. The above code provides a prompt, waiting for the user to input.

```
name = input("What is your name? ")  
# Out: What is your name?
```

If the user types "Bob" and hits enter, the variable `name` will be assigned to the string "Bob":

```
name = input("What is your name? ")  
# Out: What is your name? Bob  
print(name)  
# Out: Bob
```

Note that the `input` is always of type `str`, which is important if you want the user to enter numbers. Therefore, you need to convert the `str` before trying to use it as a number:

```
x = input("Write a number:")  
# Out: Write a number: 10  
x / 2  
# Out: TypeError: unsupported operand type(s) for /: 'str' and 'int'  
float(x) / 2  
# Out: 5.0
```

NB: It's recommended to use `try/except` blocks to catch exceptions when dealing with user inputs. For instance, if your code wants to cast a `raw_input` into an `int`, and what the user writes is uncastable, it raises a `ValueError`.

## Section 1.8: Built in Modules and Functions

A module is a file containing Python definitions and statements. Function is a piece of code which execute some logic.

```
>>> pow(2,3) #8
```

要检查Python中的内置函数，我们可以使用 `dir()`。如果不带参数调用，则返回当前作用域中的名称。否则，返回一个按字母顺序排列的名称列表，该列表包含给定对象的（部分）属性以及从中可访问的属性。

```
>>> dir(__builtins__)
[
    'ArithmetError',
    'AssertionError',
    'AttributeError',
    'BaseException',
    'BufferError',
    'BytesWarning',
    'DeprecationWarning',
    'EOFError',
    'Ellipsis',
    'EnvironmentError',
    'Exception',
    'False',
    'FloatingPointError',
    'FutureWarning',
    'GeneratorExit',
    'IOError',
    'ImportError',
    'ImportWarning',
    'IndentationError',
    'IndexError',
    'KeyError',
    'KeyboardInterrupt',
    'LookupError',
    'MemoryError',
    'NameError',
    'None',
    'NotImplemented',
    'NotImplementedError',
    'OSError',
    'OverflowError',
    'PendingDeprecationWarning',
    'ReferenceError',
    'RuntimeError',
    'RuntimeWarning',
    'StandardError',
    'StopIteration',
    'SyntaxError',
    'SyntaxWarning',
    'SystemError',
    'SystemExit',
    'TabError',
    'True',
    'TypeError',
    'UnboundLocalError',
    'UnicodeDecodeError',
    'UnicodeEncodeError',
    'UnicodeError',
    'UnicodeTranslateError',
    'UnicodeWarning',
    'UserWarning',
    'ValueError',
    'Warning',
    'ZeroDivisionError',
    '__debug__',
    '__doc__'
```

To check the built in function in python we can use `dir()` . If called without an argument, return the names in the current scope. Else, return an alphabetized list of names comprising (some of) the attribute of the given object, and of attributes reachable from it.

```
>>> dir(__builtins__)
[
    'ArithmetError',
    'AssertionError',
    'AttributeError',
    'BaseException',
    'BufferError',
    'BytesWarning',
    'DeprecationWarning',
    'EOFError',
    'Ellipsis',
    'EnvironmentError',
    'Exception',
    'False',
    'FloatingPointError',
    'FutureWarning',
    'GeneratorExit',
    'IOError',
    'ImportError',
    'ImportWarning',
    'IndentationError',
    'IndexError',
    'KeyError',
    'KeyboardInterrupt',
    'LookupError',
    'MemoryError',
    'NameError',
    'None',
    'NotImplemented',
    'NotImplementedError',
    'OSError',
    'OverflowError',
    'PendingDeprecationWarning',
    'ReferenceError',
    'RuntimeError',
    'RuntimeWarning',
    'StandardError',
    'StopIteration',
    'SyntaxError',
    'SyntaxWarning',
    'SystemError',
    'SystemExit',
    'TabError',
    'True',
    'TypeError',
    'UnboundLocalError',
    'UnicodeDecodeError',
    'UnicodeEncodeError',
    'UnicodeError',
    'UnicodeTranslateError',
    'UnicodeWarning',
    'UserWarning',
    'ValueError',
    'Warning',
    'ZeroDivisionError',
    '__debug__',
    '__doc__'
```

```
'__import__',  
'__name__',  
'__package__',  
'abs',  
'all',  
'any',  
'apply',  
'basestring',  
'bin',  
'bool',  
'buffer',  
'bytearray',  
'bytes',  
'callable',  
'chr',  
'classmethod',  
'cmp',  
'coerce',  
'compile',  
'complex',  
'copyright',  
'credits',  
'delattr',  
'dict',  
'dir',  
'divmod',  
'enumerate',  
'eval',  
'execfile',  
'exit',  
'file',  
'filter',  
'float',  
'format',  
'frozenset',  
'getattr',  
'globals',  
'hasattr',  
'hash',  
'help',  
'hex',  
'id',  
'input',  
'int',  
'intern',  
'isinstance',  
'issubclass',  
'iter',  
'len',  
'license',  
'list',  
'locals',  
'long',  
'map',  
'max',  
'memoryview',  
'min',  
'next',  
'object',  
'oct',  
'open',  
'ord',
```

```
'__import__',  
'__name__',  
'__package__',  
'abs',  
'all',  
'any',  
'apply',  
'basestring',  
'bin',  
'bool',  
'buffer',  
'bytearray',  
'bytes',  
'callable',  
'chr',  
'classmethod',  
'cmp',  
'coerce',  
'compile',  
'complex',  
'copyright',  
'credits',  
'delattr',  
'dict',  
'dir',  
'divmod',  
'enumerate',  
'eval',  
'execfile',  
'exit',  
'file',  
'filter',  
'float',  
'format',  
'frozenset',  
'getattr',  
'globals',  
'hasattr',  
'hash',  
'help',  
'hex',  
'id',  
'input',  
'int',  
'intern',  
'isinstance',  
'issubclass',  
'iter',  
'len',  
'license',  
'list',  
'locals',  
'long',  
'map',  
'max',  
'memoryview',  
'min',  
'next',  
'object',  
'oct',  
'open',  
'ord',
```

```
'pow',
'print',
'property',
'quit',
'range',
'raw_input',
'reduce',
'reload',
'repr',
'reversed',
'round',
'set',
'setattr',
'slice',
'sorted',
'staticmethod',
'str',
'sum',
'super',
'tuple',
'type',
'unichr',
'unicode',
'vars',
'xrange',
'zip'
]
```

要了解任何函数的功能，我们可以使用内置函数help。

```
>>> help(max)
内置函数 max 的帮助信息，位于模块 __builtin__ 中：
max(...)
    max(iterable[, key=func]) -> 值
    max(a, b, c, ...[, key=func]) -> 值
当只有一个可迭代参数时，返回其中最大的元素。
当有两个或更多参数时，返回最大的参数。
```

内置模块包含额外的功能。例如，要获取一个数的平方根，我们需要导入math模块。

```
>>> import math
>>> math.sqrt(16) # 4.0
```

要了解模块中的所有函数，我们可以将函数列表赋值给一个变量，然后打印该变量。

```
>>> import math
>>> dir(math)

['__doc__', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'trunc']
```

看起来\_\_doc\_\_对于在函数中提供一些文档说明很有用

```
'pow',
'print',
'property',
'quit',
'range',
'raw_input',
'reduce',
'reload',
'repr',
'reversed',
'round',
'set',
'setattr',
'slice',
'sorted',
'staticmethod',
'str',
'sum',
'super',
'tuple',
'type',
'unichr',
'unicode',
'vars',
'xrange',
'zip'
]
```

To know the functionality of any function, we can use built in function `help`.

```
>>> help(max)
Help on built-in function max in module __builtin__:
max(...)
    max(iterable[, key=func]) -> value
    max(a, b, c, ...[, key=func]) -> value
With a single iterable argument, return its largest item.
With two or more arguments, return the largest argument.
```

Built in modules contains extra functionalities. For example to get square root of a number we need to include `math` module.

```
>>> import math
>>> math.sqrt(16) # 4.0
```

To know all the functions in a module we can assign the functions list to a variable, and then print the variable.

```
>>> import math
>>> dir(math)

['__doc__', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'trunc']
```

it seems \_\_doc\_\_ is useful to provide some documentation in, say, functions

```
>>> math.__doc__  
'该模块始终可用。它提供了访问C标准定义的数学函数的接口。'
```

除了函数，文档也可以在模块中提供。因此，如果你有一个名为helloWorld.py的文件，内容如下：

```
"""这是模块的文档字符串。  
  
def sayHello():  
    """这是函数的文档字符串。  
    return 'Hello World'
```

你可以这样访问它的文档字符串：

```
>>> import helloWorld  
>>> helloWorld.__doc__  
'这是模块的文档字符串。  
>>> helloWorld.sayHello.__doc__  
'这是函数的文档字符串。'
```

- 对于任何用户定义的类型，其属性、其类的属性，以及递归地其类的基类的属性，都可以使用 `dir()` 函数获取

```
>>> class MyClassObject(object):  
...     pass  
  
...  
>>> dir(MyClassObject)  
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__',  
'__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',  
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

任何数据类型都可以使用内置函数 `str` 简单地转换为字符串。当数据类型传递给 `print` 时，默认会调用此函数

```
>>> str(123) # "123"
```

## 第1.9节：创建模块

模块是一个可导入的文件，包含定义和语句。

可以通过创建一个.py文件来创建一个模块。

```
# hello.py  
def say_hello():  
    print("Hello!")
```

模块中的函数可以通过导入该模块来使用。

对于你自己制作的模块，它们需要与导入它们的文件位于同一目录下。（不过，你也可以将它们放入包含预装模块的Python库目录中，但应尽量避免这样做。）

```
$ python  
>>> import hello  
>>> hello.say_hello()
```

```
>>> math.__doc__  
'This module is always available. It provides access to the mathematical  
functions defined by the C standard.'
```

In addition to functions, documentation can also be provided in modules. So, if you have a file named `helloWorld.py` like this:

```
"""This is the module docstring.  
  
def sayHello():  
    """This is the function docstring."  
    return 'Hello World'
```

You can access its docstrings like this:

```
>>> import helloWorld  
>>> helloWorld.__doc__  
'This is the module docstring.'  
>>> helloWorld.sayHello.__doc__  
'This is the function docstring.'
```

- For any user defined type, its attributes, its class's attributes, and recursively the attributes of its class's base classes can be retrieved using `dir()`

```
>>> class MyClassObject(object):  
...     pass  
  
...  
>>> dir(MyClassObject)  
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__',  
'__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',  
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Any data type can be simply converted to string using a builtin function called `str`. This function is called by default when a data type is passed to `print`

```
>>> str(123) # "123"
```

## Section 1.9: Creating a module

A module is an importable file containing definitions and statements.

A module can be created by creating a .py file.

```
# hello.py  
def say_hello():  
    print("Hello!")
```

Functions in a module can be used by importing the module.

For modules that you have made, they will need to be in the same directory as the file that you are importing them into. (However, you can also put them into the Python lib directory with the pre-included modules, but should be avoided if possible.)

```
$ python  
>>> import hello  
>>> hello.say_hello()
```

```
=> "Hello!"
```

模块可以被其他模块导入。

```
# greet.py  
import hello  
hello.say_hello()
```

可以导入模块的特定功能。

```
# greet.py  
from hello import say_hello  
say_hello()
```

模块可以设置别名。

```
# greet.py  
import hello as ai  
ai.say_hello()
```

模块可以作为独立可运行的脚本。

```
# run_hello.py  
if __name__ == '__main__':  
    from hello import say_hello  
    say_hello()
```

运行它！

```
$ python run_hello.py  
=> "Hello!"
```

如果模块位于某个目录内并且需要被 Python 识别，该目录应包含一个名为 `__init__.py` 的文件。

## 第1.10节：Python 2.7.x 和 3.x 的安装

**注意：**以下说明针对 Python 2.7 编写（除非另有说明）：Python 3.x 的安装方法类似。

### Windows

首先，从官方网站 (<https://www.python.org/downloads/>) 下载最新版本的 Python 2.7。该版本以 MSI 安装包形式提供。要手动安装，只需双击该文件。

默认情况下，Python 安装在以下目录：

```
C:\Python27\
```

警告：安装程序不会自动修改 PATH 环境变量。

假设您的 Python 安装在 C:\Python27，将此添加到您的 PATH 中：

```
=> "Hello!"
```

Modules can be imported by other modules.

```
# greet.py  
import hello  
hello.say_hello()
```

Specific functions of a module can be imported.

```
# greet.py  
from hello import say_hello  
say_hello()
```

Modules can be aliased.

```
# greet.py  
import hello as ai  
ai.say_hello()
```

A module can be stand-alone runnable script.

```
# run_hello.py  
if __name__ == '__main__':  
    from hello import say_hello  
    say_hello()
```

Run it!

```
$ python run_hello.py  
=> "Hello!"
```

If the module is inside a directory and needs to be detected by python, the directory should contain a file named `__init__.py`.

## Section 1.10: Installation of Python 2.7.x and 3.x

**Note:** Following instructions are written for Python 2.7 (unless specified): instructions for Python 3.x are similar.

### Windows

First, download the latest version of Python 2.7 from the official Website (<https://www.python.org/downloads/>). Version is provided as an MSI package. To install it manually, just double-click the file.

By default, Python installs to a directory:

```
C:\Python27\
```

Warning: installation does not automatically modify the PATH environment variable.

Assuming that your Python installation is in C:\Python27, add this to your PATH:

C:\Python27\;C:\Python27\Scripts\

现在在命令提示符中输入以下命令以检查Python安装是否有效：

```
python --version
```

## Python 2.x 和 3.x 并存

在Windows机器上安装并同时使用Python 2.x和3.x：

### 1. 使用MSI安装程序安装Python 2.x。

- 确保Python为所有用户安装。
- 可选：将Python添加到PATH，以便通过命令行使用python调用Python 2.x。

### 2. 使用相应的安装程序安装Python 3.x。

- 再次确保为所有用户安装了Python。
- 可选：将Python添加到PATH，以便通过命令行使用python调用Python 3.x。这样可能会覆盖Python 2.x的PATH设置，因此请仔细检查你的PATH并确保其配置符合你的偏好。
- 确保为所有用户安装了py launcher。

Python 3将安装Python启动器，可用于在命令行中交替启动Python 2.x和Python 3.x：

```
P:\>py -3
Python 3.6.1 (v3.6.1:69c0db5, 2017年3月21日, 17:54:52) [MSC v.1900 32 位 (Intel)] 在win32上
输入"help"、"copyright"、"credits"或"license"获取更多信息。
>>>
```

```
C:\>py -2
Python 2.7.13 (v2.7.13:a06454b1afa1, 2016年12月17日, 20:42:59) [MSC v.1500 32 Intel] 在win32上
输入"help"、"copyright"、"credits"或"license"获取更多信息。
>>>
```

要使用特定Python版本对应的pip，请使用：

```
C:\>py -3 -m pip -V
pip 9.0.1 来自 C:\Python36\lib\site-packages (python 3.6)
```

```
C:\>py -2 -m pip -V
pip 9.0.1 来自 C:\Python27\lib\site-packages (python 2.7)
```

## Linux

最新版本的 CentOS、Fedora、红帽企业版 (RHEL) 和 Ubuntu 都自带 Python 2.7。

要在 Linux 上手动安装 Python 2.7，只需在终端执行以下操作：

```
wget --no-check-certificate https://www.python.org/ftp/python/2.7.X/Python-2.7.X.tgz
tar -xzf Python-2.7.X.tgz
cd Python-2.7.X
./configure
make
```

C:\Python27\;C:\Python27\Scripts\

Now to check if Python installation is valid write in cmd:

```
python --version
```

## Python 2.x and 3.x Side-By-Side

To install and use both Python 2.x and 3.x side-by-side on a Windows machine:

### 1. Install Python 2.x using the MSI installer.

- Ensure Python is installed for all users.
- Optional: add Python to PATH to make Python 2.x callable from the command-line using python.

### 2. Install Python 3.x using its respective installer.

- Again, ensure Python is installed for all users.
- Optional: add Python to PATH to make Python 3.x callable from the command-line using python. This may override Python 2.x PATH settings, so double-check your PATH and ensure it's configured to your preferences.
- Make sure to install the py launcher for all users.

Python 3 will install the Python launcher which can be used to launch Python 2.x and Python 3.x interchangeably from the command-line:

```
P:\>py -3
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
C:\>py -2
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 Intel] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To use the corresponding version of pip for a specific Python version, use:

```
C:\>py -3 -m pip -V
pip 9.0.1 from C:\Python36\lib\site-packages (python 3.6)
```

```
C:\>py -2 -m pip -V
pip 9.0.1 from C:\Python27\lib\site-packages (python 2.7)
```

## Linux

The latest versions of CentOS, Fedora, Red Hat Enterprise (RHEL) and Ubuntu come with Python 2.7.

To install Python 2.7 on linux manually, just do the following in terminal:

```
wget --no-check-certificate https://www.python.org/ftp/python/2.7.X/Python-2.7.X.tgz
tar -xzf Python-2.7.X.tgz
cd Python-2.7.X
./configure
make
```

```
sudo make install
```

还需将新 Python 的路径添加到 PATH 环境变量中。如果新 Python 位于 /root/python-2.7.X，则运行  
export PATH = \$PATH:/root/python-2.7.X

现在要检查 Python 安装是否有效，请在终端输入：

```
python --version
```

Ubuntu (从源码安装)

如果你需要 Python 3.6，可以按照下面的方法从源码安装（Ubuntu 16.10 和 17.04 的通用仓库中已有 3.6 版本）。以下步骤适用于 Ubuntu 16.04 及更低版本：

```
sudo apt install build-essential checkinstall  
sudo apt install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev  
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tar.xz  
tar xvf Python-3.6.1.tar.xz  
cd Python-3.6.1/  
.configure --enable-optimizations  
sudo make altinstall
```

## macOS

目前，macOS 自带 Python 2.7.10，但该版本已经过时，并且与常规 Python 有所不同。

OS X 自带的 Python 版本适合学习，但不适合开发。OS X 自带的版本可能落后于官方当前的 Python 版本，而官方版本被视为稳定的生产版本。[\(source\)](#)

安装 Homebrew：

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

安装 Python 2.7：

```
brew install python
```

对于 Python 3.x，请使用命令 brew install python3。

## 第1.11节：字符串函数 - str() 和 repr()

有两个函数可以用来获取对象的可读表示。

`repr(x)` 调用 `x.__repr__()`：返回 `x` 的表示。`eval` 通常会将此函数的结果转换回原始对象。

`str(x)` 调用 `x.__str__()`：返回描述对象的可读字符串。该字符串可能省略一些技术细节。

### repr()

```
sudo make install
```

Also add the path of new python in PATH environment variable. If new python is in /root/python-2.7.X then run  
export PATH = \$PATH:/root/python-2.7.X

Now to check if Python installation is valid write in terminal:

```
python --version
```

Ubuntu (From Source)

If you need Python 3.6 you can install it from source as shown below (Ubuntu 16.10 and 17.04 have 3.6 version in the universal repository). Below steps have to be followed for Ubuntu 16.04 and lower versions:

```
sudo apt install build-essential checkinstall  
sudo apt install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev  
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tar.xz  
tar xvf Python-3.6.1.tar.xz  
cd Python-3.6.1/  
.configure --enable-optimizations  
sudo make altinstall
```

## macOS

As we speak, macOS comes installed with Python 2.7.10, but this version is outdated and slightly modified from the regular Python.

The version of Python that ships with OS X is great for learning but it's not good for development. The version shipped with OS X may be out of date from the official current Python release, which is considered the stable production version. [\(source\)](#)

Install [Homebrew](#):

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Install Python 2.7:

```
brew install python
```

For Python 3.x, use the command `brew install python3` instead.

## Section 1.11: String function - str() and repr()

There are two functions that can be used to obtain a readable representation of an object.

`repr(x)` calls `x.__repr__()`: a representation of `x`. `eval` will usually convert the result of this function back to the original object.

`str(x)` calls `x.__str__()`: a human-readable string that describes the object. This may elide some technical detail.

### repr()

对于许多类型，此函数尝试返回一个字符串，当传递给 eval() 时能生成具有相同值的对象。否则，表示形式是一个用尖括号括起来的字符串，包含对象类型的名称及其他信息。通常包括对象的名称和地址。

### str()

对于字符串，该函数返回字符串本身。它与repr(object)的区别在于，str(object)并不总是尝试返回一个可被eval()接受的字符串。相反，它的目标是返回一个可打印的或“人类可读”的字符串。如果没有给出参数，则返回空字符串”。

示例 1：

```
s = """w'o"w"""
repr(s) # 输出: '\w\\\'o"w\
str(s) # 输出: 'w'o"w'
eval(str(s)) == s # 会抛出 SyntaxError
eval(repr(s)) == s # 输出: True
```

示例 2：

```
import datetime
today = datetime.datetime.now()
str(today) # 输出: '2016-09-15 06:58:46.915000'
repr(today) # 输出: 'datetime.datetime(2016, 9, 15, 6, 58, 46, 915000)'
```

编写类时，您可以重写这些方法来实现您想要的功能：

```
class Represent(object):

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "Represent(x={},y={})".format(self.x, self.y)

    def __str__(self):
        return "Representing x as {} and y as {}".format(self.x, self.y)
```

使用上述类我们可以看到结果：

```
r = Represent(1, "Hopper")
print(r) # 打印 __str__
print(r.__repr__) # 打印 __repr__: '<bound method Represent.__repr__ of
Represent(x=1,y="Hopper")>'
rep = r.__repr__() # 将 __repr__ 的执行结果赋值给一个新变量
print(rep) # 打印 'Represent(x=1,y="Hopper")'
r2 = eval(rep) # 计算 rep
print(r2) # 打印新对象的 __str__
print(r2 == r) # 打印 'False'，因为它们是不同的对象
```

## 第1.12节：使用 pip 安装外部模块

当你需要从 Python 包索引 (PyPI) 中众多可选包中安装任何包时，pip 是你的好帮手。如果你使用的是 Python 2 >= 2.7.9 或 Python 3 >= 3.4 (从 python.org 下载)，pip 已经预装。对于运行 Linux 或其他带有本地包管理器的 \*nix 系统的计算机，pip 通常需要手动安装。

For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval(). Otherwise, the representation is a string enclosed in angle brackets that contains the name of the type of the object along with additional information. This often includes the name and address of the object.

### str()

For strings, this returns the string itself. The difference between this and repr(object) is that str(object) does not always attempt to return a string that is acceptable to eval(). Rather, its goal is to return a printable or 'human readable' string. If no argument is given, this returns the empty string, ''.

Example 1:

```
s = """w'o"w"""
repr(s) # Output: '\w\\\'o"w\
str(s) # Output: 'w'o"w'
eval(str(s)) == s # Gives a SyntaxError
eval(repr(s)) == s # Output: True
```

Example 2:

```
import datetime
today = datetime.datetime.now()
str(today) # Output: '2016-09-15 06:58:46.915000'
repr(today) # Output: 'datetime.datetime(2016, 9, 15, 6, 58, 46, 915000)'
```

When writing a class, you can override these methods to do whatever you want:

```
class Represent(object):

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "Represent(x={},y={})".format(self.x, self.y)

    def __str__(self):
        return "Representing x as {} and y as {}".format(self.x, self.y)
```

Using the above class we can see the results:

```
r = Represent(1, "Hopper")
print(r) # prints __str__
print(r.__repr__) # prints __repr__: '<bound method Represent.__repr__ of
Represent(x=1,y="Hopper")>'
rep = r.__repr__() # sets the execution of __repr__ to a new variable
print(rep) # prints 'Represent(x=1,y="Hopper")'
r2 = eval(rep) # evaluates rep
print(r2) # prints __str__ from new object
print(r2 == r) # prints 'False' because they are different objects
```

## Section 1.12: Installing external modules using pip

pip is your friend when you need to install any package from the plethora of choices available at the python package index (PyPI). pip is already installed if you're using Python 2 >= 2.7.9 or Python 3 >= 3.4 downloaded from python.org. For computers running Linux or another \*nix with a native package manager, pip must often be [manually installed](#).

在同时安装了 Python 2 和 Python 3 的环境中，pip 通常指向 Python 2，而 pip3 指向 Python 3。使用 pip 只会为 Python 2 安装包，使用 pip3 只会为 Python 3 安装包。

## 查找 / 安装包

搜索包就像输入一样简单

```
$ pip search <query>
# 搜索名称或摘要中包含 <query> 的软件包
```

安装软件包就像输入命令一样简单（在终端/命令提示符中，而不是在 Python 解释器中）

```
$ pip install [package_name]          # 安装软件包的最新版本
$ pip install [package_name]==x.x.x    # 安装软件包的指定版本
$ pip install '[package_name]>=x.x.x'  # 安装软件包的最低版本
```

其中 x.x.x 是你想安装的软件包的版本号。

当你的服务器处于代理后面时，可以使用以下命令安装软件包：

```
$ pip --proxy http://<服务器地址>:<端口> install
```

## 升级已安装的软件包

当已安装的软件包有新版本时，它们不会自动安装到你的系统。要查看哪些已安装的软件包已过时，请运行：

```
$ pip list --outdated
```

要升级特定的软件包，请使用

```
$ pip install [package_name] --upgrade
```

更新所有过时的软件包不是pip的标准功能。

## 升级 pip

您可以使用以下命令升级现有的 pip 安装

- 在 Linux 或 macOS X 上：

```
$ pip install -U pip
```

在某些 Linux 系统上，您可能需要使用 sudo 来运行 pip

- 在 Windows 上：

```
py -m pip install -U pip
```

或者

```
python -m pip install -U pip
```

On instances with both Python 2 and Python 3 installed, pip often refers to Python 2 and pip3 to Python 3. Using pip will only install packages for Python 2 and pip3 will only install packages for Python 3.

## Finding / installing a package

Searching for a package is as simple as typing

```
$ pip search <query>
# Searches for packages whose name or summary contains <query>
```

Installing a package is as simple as typing (*in a terminal / command-prompt, not in the Python interpreter*)

```
$ pip install [package_name]          # latest version of the package
$ pip install [package_name]==x.x.x    # specific version of the package
$ pip install '[package_name]>=x.x.x'  # minimum version of the package
```

where x.x.x is the version number of the package you want to install.

When your server is behind proxy, you can install package by using below command:

```
$ pip --proxy http://<server address>:<port> install
```

## Upgrading installed packages

When new versions of installed packages appear they are not automatically installed to your system. To get an overview of which of your installed packages have become outdated, run:

```
$ pip list --outdated
```

To upgrade a specific package use

```
$ pip install [package_name] --upgrade
```

Updating all outdated packages is not a standard functionality of pip.

## Upgrading pip

You can upgrade your existing pip installation by using the following commands

- On Linux or macOS X:

```
$ pip install -U pip
```

You may need to use sudo with pip on some Linux Systems

- On Windows:

```
py -m pip install -U pip
```

or

```
python -m pip install -U pip
```

有关 pip 的更多信息, 请[点击这里阅读](#)。

## 第1.13节：帮助工具

Python解释器内置了多个函数。如果你想获取关键字、内置函数、模块或主题的信息, 可以打开Python控制台并输入:

```
>>> help()
```

你可以通过直接输入关键字来获取信息:

```
>>> help(help)
```

或者在工具内输入:

```
help> help
```

这将显示一个说明:

模块\_sitebuiltins对象中的\_Helper的帮助:

```
class _Helper(builtins.object)
| 定义内置的'help'.
|
| 这是 pydoc.help 的一个包装, 当在 Python 交互提示符输入 'help' 时, 会提供一条有用的信息
|.
|
| 在 Python 提示符调用 help() 会启动一个交互式帮助会话。
| 调用 help(thing) 会打印 Python 对象 'thing' 的帮助信息。
|
| 此处定义的方法:
|
| __call__(self, *args, **kwds)
|
| __repr__(self)
|
| -----
|
| 此处定义的数据描述符:
|
| __dict__
| 实例变量的字典 (如果定义了)
|
| __weakref__
| 对象的弱引用列表 (如果已定义)
```

您也可以请求模块的子类:

```
help(pymysql.connections)
```

您可以使用 help 来访问已导入的不同模块的文档字符串, 例如, 尝试以下操作:

```
>>> help(math)
```

然后您会收到一个错误

```
>>> import math
```

For more information regarding pip do [read here](#).

## Section 1.13: Help Utility

Python has several functions built into the interpreter. If you want to get information of keywords, built-in functions, modules or topics open a Python console and enter:

```
>>> help()
```

You will receive information by entering keywords directly:

```
>>> help(help)
```

or within the utility:

```
help> help
```

which will show an explanation:

Help on \_Helper in module \_sitebuiltins object:

```
class _Helper(builtins.object)
| Define the builtin 'help'.
|
| This is a wrapper around pydoc.help that provides a helpful message
| when 'help' is typed at the Python interactive prompt.
|
| Calling help() at the Python prompt starts an interactive help session.
| Calling help(thing) prints help for the python object 'thing'.
|
| Methods defined here:
|
| __call__(self, *args, **kwds)
|
| __repr__(self)
|
| -----
|
| Data descriptors defined here:
|
| __dict__
| dictionary for instance variables (if defined)
|
| __weakref__
| list of weak references to the object (if defined)
```

You can also request subclasses of modules:

```
help(pymysql.connections)
```

You can use help to access the docstrings of the different modules you have imported, e.g., try the following:

```
>>> help(math)
```

and you'll get an error

```
>>> import math
```

```
>>> help(math)
```

现在您将在导入模块之后，获得该模块中可用方法的列表。

使用 `quit` 关闭帮助程序

```
>>> help(math)
```

And now you will get a list of the available methods in the module, but only AFTER you have imported it.

Close the helper with `quit`

# 第2章：Python数据类型

数据类型就是你用来在内存中保留空间的变量。Python变量不需要显式声明来保留内存空间。当你给变量赋值时，声明会自动发生。

## 第2.1节：字符串数据类型

字符串被识别为用引号表示的一组连续字符。Python允许使用单引号或双引号成对表示。字符串是不可变的序列数据类型，即每次对字符串进行任何修改时，都会创建一个全新的字符串对象。

```
a_str = 'Hello World'  
print(a_str)      #输出将是整个字符串。Hello World  
print(a_str[0])    #输出将是第一个字符。H  
print(a_str[0:5])  #输出将是前五个字符。Hello
```

## 第2.2节：集合数据类型

集合是无序的唯一对象集合，集合有两种类型：

1.集合 - 它们是可变的，定义集合后可以添加新元素

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}  
print(basket)          # 重复项将被移除  
> {'orange', 'banana', 'pear', 'apple'}  
a = set('abracadabra')  
print(a)                # a 中的唯一字母  
> {'a', 'r', 'b', 'c', 'd'}  
a.add('z')  
print(a)  
> {'a', 'c', 'r', 'b', 'z', 'd'}
```

2.冻结集合 - 它们是不可变的，定义后不能添加新元素。

```
b = frozenset('asdfagsa')  
print(b)  
> frozenset({'f', 'g', 'd', 'a', 's'})  
cities = frozenset(["Frankfurt", "Basel", "Freiburg"])  
print(cities)  
> frozenset({'Frankfurt', 'Basel', 'Freiburg'})
```

## 第2.3节：数字数据类型

Python中有四种数字类型。整数 (int)、浮点数 (float)、复数 (complex) 和长整数 (long)。

```
int_num = 10      #整数值  
float_num = 10.2   #浮点数值  
complex_num = 3.14j  #复数值  
long_num = 1234567L #长整数值
```

# Chapter 2: Python Data Types

Data types are nothing but variables you use to reserve some space in memory. Python variables do not need an explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable.

## Section 2.1: String Data Type

String are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Strings are immutable sequence data type, i.e each time one makes any changes to a string, completely new string object is created.

```
a_str = 'Hello World'  
print(a_str)      #output will be whole string. Hello World  
print(a_str[0])    #output will be first character. H  
print(a_str[0:5])  #output will be first five characters. Hello
```

## Section 2.2: Set Data Types

Sets are unordered collections of unique objects, there are two types of set:

1. Sets - They are mutable and new elements can be added once sets are defined

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}  
print(basket)          # duplicates will be removed  
> {'orange', 'banana', 'pear', 'apple'}  
a = set('abracadabra')  
print(a)                # unique letters in a  
> {'a', 'r', 'b', 'c', 'd'}  
a.add('z')  
print(a)  
> {'a', 'c', 'r', 'b', 'z', 'd'}
```

2. Frozen Sets - They are immutable and new elements cannot be added after its defined.

```
b = frozenset('asdfagsa')  
print(b)  
> frozenset({'f', 'g', 'd', 'a', 's'})  
cities = frozenset(["Frankfurt", "Basel", "Freiburg"])  
print(cities)  
> frozenset({'Frankfurt', 'Basel', 'Freiburg'})
```

## Section 2.3: Numbers data type

Numbers have four types in Python. Int, float, complex, and long.

```
int_num = 10      #int value  
float_num = 10.2   #float value  
complex_num = 3.14j  #complex value  
long_num = 1234567L #long value
```

## 第2.4节：列表数据类型

列表包含由逗号分隔并用方括号[]括起来的项目。列表与C语言中的数组非常相似。一个区别是列表中的所有项目可以是不同的数据类型。

```
list = [123,'abcd',10.2,'d']    #可以是任何数据类型或单一数据类型的数组。  
list1 = ['hello','world']  
print(list)      #将输出整个列表。[123,'abcd',10.2,'d']  
print(list[0:2])    #将输出列表的前两个元素。[123,'abcd']  
print(list1 * 2)    #将输出list1两次。['hello','world','hello','world']  
print(list + list1)   #将输出两个列表的连接。  
[123, 'abcd', 10.2, 'd', 'hello', 'world']
```

## Section 2.4: List Data Type

A list contains items separated by commas and enclosed within square brackets []. lists are almost similar to arrays in C. One difference is that all the items belonging to a list can be of different data type.

```
list = [123, 'abcd', 10.2, 'd']    #can be an array of any data type or single data type.  
list1 = ['hello', 'world']  
print(list)      #will output whole list. [123, 'abcd', 10.2, 'd']  
print(list[0:2])    #will output first two element of list. [123, 'abcd']  
print(list1 * 2)    #will gave list1 two times. ['hello', 'world', 'hello', 'world']  
print(list + list1)   #will gave concatenation of both the lists.  
[123, 'abcd', 10.2, 'd', 'hello', 'world']
```

## 第2.5节：字典数据类型

字典由键值对组成。它用花括号{}括起来，值可以通过方括号[]进行赋值和访问。

```
dic={'name':'red','age':10}  
print(dic)      #将输出所有键值对。{'name':'red','age':10}  
print(dic['name'])    #将只输出键为'name'的值。'red'  
print(dic.values())  #将输出dic中值的列表。['red',10]  
print(dic.keys())   #将输出键的列表。['name','age']
```

## Section 2.5: Dictionary Data Type

Dictionary consists of key-value pairs. It is enclosed by curly braces {} and values can be assigned and accessed using square brackets[].

```
dic={'name':'red', 'age':10}  
print(dic)      #will output all the key-value pairs. {'name':'red', 'age':10}  
print(dic['name'])    #will output only value with 'name' key. 'red'  
print(dic.values())  #will output list of values in dic. ['red', 10]  
print(dic.keys())   #will output list of keys. ['name', 'age']
```

## 第2.6节：元组数据类型

列表用方括号 [ ] 括起来，其元素和大小可以改变，而元组用圆括号 ( ) 括起来，且不能被更新。元组是不可变的。

```
tuple = (123, 'hello')  
tuple1 = ('world')  
print(tuple)      #将输出整个元组。(123,'hello')  
print(tuple[0])    #将输出第一个值。(123)  
print(tuple + tuple1)  #将输出(123,'hello','world')  
tuple[1]='update'   #这将导致错误。
```

## Section 2.6: Tuple Data Type

Lists are enclosed in brackets [] and their elements and size can be changed, while tuples are enclosed in parentheses () and cannot be updated. Tuples are immutable.

```
tuple = (123, 'hello')  
tuple1 = ('world')  
print(tuple)      #will output whole tuple. (123, 'hello')  
print(tuple[0])    #will output first value. (123)  
print(tuple + tuple1)  #will output (123, 'hello', 'world')  
tuple[1]='update'   #this will give you error.
```

# 第3章：缩进

## 第3.1节：简单示例

对于Python，Guido van Rossum基于缩进来分组语句。其原因在“设计与历史Python FAQ”的第一节中进行了说明。冒号 : 用于声明一个缩进代码块，例如以下示例：

```
class 示例类:  
    #属于类的每个函数必须缩进相同  
    def __init__(self):  
        name = "example"  
  
    def 某函数(self, a):  
        #注意属于函数的所有内容必须缩进  
        if a > 5:  
            return True  
        else:  
            return False  
  
        #如果函数没有缩进到相同级别，则不会被视为父类的一部分  
def separateFunction(b):  
    for i in b:  
        #循环也会缩进，嵌套条件会开始新的缩进  
        if i == 1:  
            return True  
    return False  
  
separateFunction([2,3,5,6,1])
```

### 空格还是制表符？

推荐的缩进是4个空格，但只要保持一致，可以使用制表符或空格。**不要在Python中混用制表符和空格** 因为这会在Python 3中导致错误，并且可能在Python 2中引发错误。

## 第3.2节：缩进是如何解析的

空白符由词法分析器在解析前处理。

词法分析器使用栈来存储缩进级别。开始时，栈中只包含值0，即最左边的位置。每当一个嵌套块开始时，新的缩进级别会被压入栈中，并且一个“INDENT”标记会插入到传递给解析器的标记流中。连续出现多个“INDENT”标记是绝不允许的（IndentationError）。

当遇到缩进级别较小的行时，会从栈中弹出值，直到栈顶的值等于新的缩进级别（如果找不到，则发生语法错误）。每弹出一个值，都会生成一个“DEDENT”标记。显然，可能会连续出现多个“DEDENT”标记。

词法分析器会跳过空行（仅包含空白字符和可能的注释），并且永远不会为它们生成“INDENT”或“DEDENT”标记。

在源代码末尾，会为栈中剩余的每个缩进级别生成“DEDENT”标记，直到只剩下0为止。

例如：

# Chapter 3: Indentation

## Section 3.1: Simple example

For Python，Guido van Rossum based the grouping of statements on indentation. The reasons for this are explained in [the first section of the "Design and History Python FAQ"](#). Colons, :, are used to [declare an indented code block](#), such as the following example:

```
class ExampleClass:  
    #Every function belonging to a class must be indented equally  
    def __init__(self):  
        name = "example"  
  
    def someFunction(self, a):  
        #Notice everything belonging to a function must be indented  
        if a > 5:  
            return True  
        else:  
            return False  
  
    #If a function is not indented to the same level it will not be considered as part of the parent class  
def separateFunction(b):  
    for i in b:  
        #Loops are also indented and nested conditions start a new indentation  
        if i == 1:  
            return True  
    return False  
  
separateFunction([2, 3, 5, 6, 1])
```

### Spaces or Tabs?

The recommended [indentation is 4 spaces](#) but tabs or spaces can be used so long as they are consistent. **Do not mix tabs and spaces in Python** as this will cause an error in Python 3 and can cause errors in Python 2.

## Section 3.2: How Indentation is Parsed

Whitespace is handled by the lexical analyzer before being parsed.

The lexical analyzer uses a stack to store indentation levels. At the beginning, the stack contains just the value 0, which is the leftmost position. Whenever a nested block begins, the new indentation level is pushed onto the stack, and an “INDENT” token is inserted into the token stream which is passed to the parser. There can never be more than one “INDENT” token in a row (IndentationError).

When a line is encountered with a smaller indentation level, values are popped from the stack until a value is on top which is equal to the new indentation level (if none is found, a syntax error occurs). For each value popped, a “DEDENT” token is generated. Obviously, there can be multiple “DEDENT” tokens in a row.

The lexical analyzer skips empty lines (those containing only whitespace and possibly comments), and will never generate either “INDENT” or “DEDENT” tokens for them.

At the end of the source code, “DEDENT” tokens are generated for each indentation level left on the stack, until just the 0 is left.

For example:

```
if foo:  
    if bar:  
x = 42  
else:  
    print foo
```

被解析为：

```
<if> <foo> <:> [0]  
<INDENT> <if> <bar> <:> [0, 4]  
<INDENT> <x> <=> <42> [0, 4, 8]  
<DEDENT> <DEDENT> <else> <:> [0]  
<INDENT> <print> <foo> [0, 2]  
<DEDENT>
```

解析器随后将“INDENT”和“DEDENT”标记作为代码块的分隔符处理。

### 第3.3节：缩进错误

整个代码中的空格应均匀一致。不正确的缩进可能导致IndentationError错误，或导致程序执行意外的操作。以下示例会引发IndentationError错误：

```
a = 7  
if a > 5:  
    print "foo"  
else:  
    print "bar"  
print "done"
```

或者如果冒号后面的行没有缩进，也会引发IndentationError错误：

```
if True:  
print "true"
```

如果你在不该缩进的地方添加缩进，将会引发IndentationError错误：

```
if True:  
a = 6  
    b = 5
```

如果你忘记取消缩进，功能可能会丢失。在这个例子中，返回的是None而不是预期的False：

```
def isEven(a):  
    if a%2 ==0:  
        return True  
    #下一行应该与if对齐  
    return False  
print isEven(7)
```

```
if foo:  
    if bar:  
x = 42  
else:  
    print foo
```

is analyzed as:

```
<if> <foo> <:> [0]  
<INDENT> <if> <bar> <:> [0, 4]  
<INDENT> <x> <=> <42> [0, 4, 8]  
<DEDENT> <DEDENT> <else> <:> [0]  
<INDENT> <print> <foo> [0, 2]  
<DEDENT>
```

The parser then handles the "INDENT" and "DEDENT" tokens as block delimiters.

### Section 3.3: Indentation Errors

The spacing should be even and uniform throughout. Improper indentation can cause an IndentationError or cause the program to do something unexpected. The following example raises an IndentationError:

```
a = 7  
if a > 5:  
    print "foo"  
else:  
    print "bar"  
print "done"
```

Or if the line following a colon is not indented, an IndentationError will also be raised:

```
if True:  
print "true"
```

If you add indentation where it doesn't belong, an IndentationError will be raised:

```
if True:  
a = 6  
    b = 5
```

If you forget to un-indent functionality could be lost. In this example None is returned instead of the expected False:

```
def isEven(a):  
    if a%2 ==0:  
        return True  
    #this next line should be even with the if  
    return False  
print isEven(7)
```

# 第4章：注释和文档

## 第4.1节：单行、内联和多行注释

注释用于解释代码，当基本代码本身不够清晰时。

Python 会忽略注释，因此不会执行注释中的代码，也不会因为纯英文句子而引发语法错误。

单行注释以井号字符 (#) 开头，并以行尾结束。

- 单行注释：

```
# 这是 Python 中的一行注释
```

- 行内注释：

```
print("Hello World") # 这一行打印"Hello World"
```

- 跨多行的注释两端使用""或''. 这与多行字符串相同，但它们可以用作注释：

```
"""
```

这种类型的注释跨多行。

这些主要用于函数、类和模块的文档说明。

```
"""
```

## 第4.2节：以编程方式访问文档字符串

文档字符串与普通注释不同，它们作为被记录函数的属性存储，这意味着你可以通过编程方式访问它们。

### 一个示例函数

```
def func():
    """这是一个什么都不做的函数"""
    return
```

可以使用`_doc_`属性访问文档字符串：

```
print(func.__doc__)
```

这是一个什么都不做的函数

```
help(func)
```

关于模块`__main__`中函数`func`的帮助：

```
func()
```

这是一个完全不执行任何操作的函数

### 另一个示例函数

# Chapter 4: Comments and Documentation

## Section 4.1: Single line, inline and multiline comments

Comments are used to explain code when the basic code itself isn't clear.

Python ignores comments, and so will not execute code in there, or raise syntax errors for plain English sentences.

Single-line comments begin with the hash character (#) and are terminated by the end of line.

- Single line comment:

```
# This is a single line comment in Python
```

- Inline comment:

```
print("Hello World") # This line prints "Hello World"
```

- Comments spanning multiple lines have ""'' or '''' on either end. This is the same as a multiline string, but they can be used as comments:

```
"""
```

This type of comment spans multiple lines.

These are mostly used for documentation of functions, classes and modules.

```
"""
```

## Section 4.2: Programmatically accessing docstrings

Docstrings are - unlike regular comments - stored as an attribute of the function they document, meaning that you can access them programmatically.

### An example function

```
def func():
    """This is a function that does nothing at all"""
    return
```

The docstring can be accessed using the `__doc__` attribute:

```
print(func.__doc__)
```

This is a function that does nothing at all

```
help(func)
```

Help on function func in module `__main__`:

```
func()
```

This is a function that does nothing at all

### Another example function

`function.__doc__` 只是作为字符串的实际文档字符串，而 `help` 函数提供关于函数的一般信息，包括文档字符串。这里有一个更有用的示例：

```
def greet(name, greeting="Hello"):
    """打印对用户`name`的问候
可选参数`greeting`可以改变问候语。"""

    print("{} {}".format(greeting, name))

help(greet)
```

关于模块`_main_`中函数`greet`的帮助：

```
greet(name, greeting='Hello')
```

向用户`name`打印问候语

可选参数`greeting`可以改变问候语的内容。

## 文档字符串相较于普通注释的优势

仅仅在函数中没有文档字符串或只有普通注释，会使函数的帮助性大大降低。

```
def greet(name, greeting="Hello"):
    # 向用户 `name` 打印问候语
    # 可选参数 `greeting` 可以改变问候语的内容。

    print("{} {}".format(greeting, name))

print(greet.__doc__)
```

None

```
help(greet)
```

模块 `main` 中函数 `greet` 的帮助信息：

```
greet(name, greeting='Hello')
```

## 第4.3节：使用文档字符串编写文档

文档字符串（`docstring`）是一种多行注释，用于记录模块、类、函数和方法。它必须是所描述组件的第一个语句。

```
def hello(name):
    """向某人问好。

打印对指定姓名的人的问候语 ("Hello")。
"""

    print("Hello "+name)

class Greeter:
    """用于向人们问好的对象。
```

`function.__doc__` is just the actual docstring as a string, while the `help` function provides general information about a function, including the docstring. Here's a more helpful example:

```
def greet(name, greeting="Hello"):
    """Print a greeting to the user `name`

Optional parameter `greeting` can change what they're greeted with."""

    print("{} {}".format(greeting, name))

help(greet)
```

Help on function `greet` in module `__main__`:

```
greet(name, greeting='Hello')
```

Print a greeting to the user name

Optional parameter `greeting` can change what they're greeted with.

## Advantages of docstrings over regular comments

Just putting no docstring or a regular comment in a function makes it a lot less helpful.

```
def greet(name, greeting="Hello"):
    # Print a greeting to the user `name`
    # Optional parameter `greeting` can change what they're greeted with.

    print("{} {}".format(greeting, name))

print(greet.__doc__)
```

None

```
help(greet)
```

Help on function `greet` in module `main`:

```
greet(name, greeting='Hello')
```

## Section 4.3: Write documentation using docstrings

A `docstring` is a multi-line comment used to document modules, classes, functions and methods. It has to be the first statement of the component it describes.

```
def hello(name):
    """Greet someone.

Print a greeting ("Hello") for the person with the given name.
"""

    print("Hello "+name)

class Greeter:
    """An object used to greet people.
```

它包含多种语言和一天中不同时间的多种问候函数。

"""

文档字符串的值可以在程序中访问，例如被help命令使用。

## 语法约定

### PEP 257

[PEP 257](#) 定义了文档字符串注释的语法规则。它基本上允许两种类型：

- 单行文档字符串：

根据PEP 257，单行文档字符串应当用于简短且简单的函数。所有内容放在一行，例如：

```
def hello():
    """向你的朋友们问好。””
    print("Hello my friends!")
```

文档字符串应以句号结尾，动词应使用祈使语气。

- 多行文档字符串：

多行文档字符串应当用于较长、更复杂的函数、模块或类。

```
def hello(name, language="en"):
    向某人打招呼。
```

参数：

name：该人的名字 language：用于打  
招呼的语言

```
print(greeting[language]+" "+name)
```

它们以简短的摘要开始（相当于一行文档字符串的内容），该摘要可以与引号在同一行  
也可以在下一行，接着给出详细说明并列出参数和返回值。

注意PEP 257定义了[文档字符串中应包含的信息](#)，但并未规定其  
应采用何种格式。这就是其他方和文档解析工具制定自己  
文档标准的原因，下面及[本问题](#)中列出了一些示例。

## Sphinx

[Sphinx](#) 是一个基于文档字符串为Python项目生成HTML文档的工具。它使用的标记语言是reStructuredText。它  
们定义了自己的文档标准，[pythonhosted.org](#)上有非常好的描述。Sphinx格式例如被pyCharm集成开发环境使用。

使用Sphinx/reStructuredText格式，函数的文档示例如下：

```
def hello(name, language="en"):
    向某人打招呼。

:param name: 该人的名字
:type name: str
:param language: 需要问候的语言
:type language: str
```

It contains multiple greeting functions for several languages  
and times of the day.

"""

The value of the docstring can be accessed within the program and is - for example - used by the [help](#) command.

## Syntax conventions

### PEP 257

[PEP 257](#) 定义了一个语法标准来处理文档字符串注释。它基本上允许两种类型：

- One-line Docstrings:

根据PEP 257，它们应该用于简短且简单的函数。所有内容放在一行，例如：

```
def hello():
    """Say hello to your friends."""
    print("Hello my friends!")
```

文档字符串应以句号结尾，动词应使用祈使语气。

- Multi-line Docstrings:

多行文档字符串应用于较长、更复杂的函数、模块或类。

```
def hello(name, language="en"):
    """Say hello to a person.
```

Arguments:

name: the name of the person  
language: the language in which the person should be greeted  
"""

```
print(greeting[language]+" "+name)
```

它们以简短的摘要开始（相当于一行文档字符串的内容），该摘要可以与引号在同一行  
也可以在下一行，接着给出详细说明并列出参数和返回值。

Note PEP 257 defines [what information should be given](#) within a docstring, it doesn't define in which format it  
should be given. This was the reason for other parties and documentation parsing tools to specify their own  
standards for documentation, some of which are listed below and in [this question](#).

## Sphinx

[Sphinx](#) 是一个工具，用于根据文档字符串为Python项目生成HTML文档。其标记语言是reStructuredText。它们  
定义了自己的文档标准，[pythonhosted.org](#)上有非常好的描述。Sphinx格式例如被pyCharm集成开发环境使用。  
[very good description of them](#). The Sphinx format is for example used by the [pyCharm IDE](#).

A function would be documented like this using the Sphinx/reStructuredText format:

```
def hello(name, language="en"):
    """Say hello to a person.

:param name: the name of the person
:type name: str
:param language: the language in which the person should be greeted
:type language: str
```

```
:return: 一个数字
:rtype: int
"""


```

```
print(greeting[language]+" "+name)
return 4
```

## Google Python 风格指南

Google 发布了[Google Python 风格指南](#), 定义了 Python 的编码规范, 包括文档注释。与 Sphinx/reST 相比, 许多人认为按照 Google 指南编写的文档更易于人类阅读。

上述提到的[pythonhosted.org 页面](#)也提供了一些符合 Google 风格指南的良好文档示例。

使用[Napoleon插件](#), Sphinx 也可以解析符合 Google 风格指南格式的文档。

使用 Google 风格指南格式, 函数的文档会这样写 :

```
def hello(name, language="en"):
    向某人打招呼。
```

参数 :

name : 作为字符串的人名  
language : 语言代码字符串

返回值 :

一个数字。

```
"""
print(greeting[language]+" "+name)
return 4
```

```
:return: a number
:rtype: int
"""


```

```
print(greeting[language]+" "+name)
return 4
```

## Google Python Style Guide

Google has published [Google Python Style Guide](#) which defines coding conventions for Python, including documentation comments. In comparison to the Sphinx/reST many people say that documentation according to Google's guidelines is better human-readable.

The [pythonhosted.org page mentioned above](#) also provides some examples for good documentation according to the Google Style Guide.

Using the [Napoleon](#) plugin, Sphinx can also parse documentation in the Google Style Guide-compliant format.

A function would be documented like this using the Google Style Guide format:

```
def hello(name, language="en"):
    """Say hello to a person.
```

Args:

name: the name of the person as string  
language: the language code string

Returns:

A number.

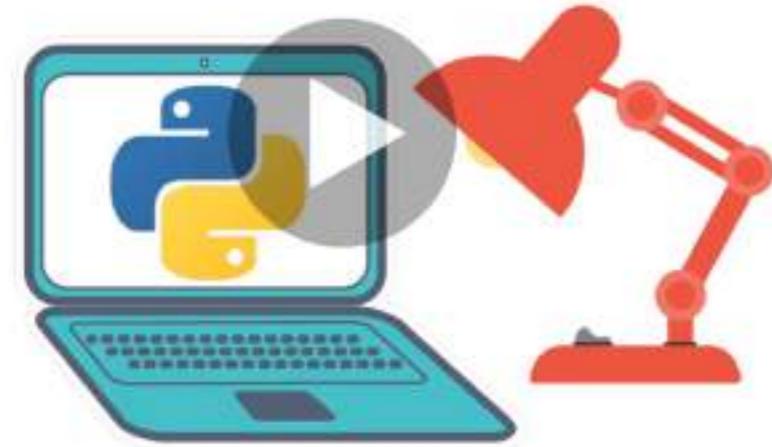
"""

```
print(greeting[language]+" "+name)
return 4
```

# 视频：完整的Python 训练营：从零开始 成为Python 3高手

像专业人士一样学习Python！从基础开始，直到  
创建你自己的应用程序和游戏！

## COMPLETE PYTHON 3 BOOTCAMP



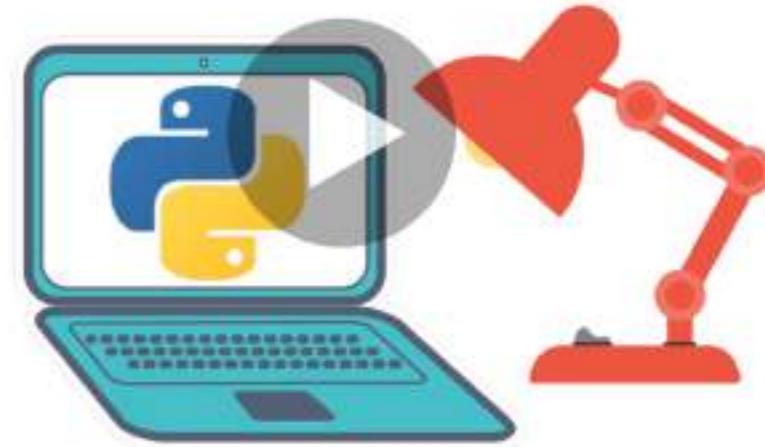
- ✓ 学习专业使用Python，掌握Python 2和Python 3！
- ✓ 用Python创建游戏，比如井字棋和二十一点！
- ✓ 学习高级Python功能，如collections模块和时间戳的使用！
- ✓ 学习使用面向对象编程和类！
- ✓ 理解复杂主题，如装饰器。
- ✓ 理解如何使用Jupyter Notebook并创建.py文件
- ✓ 了解如何在Jupyter Notebook系统中创建图形用户界面（GUI）！
- ✓ 从零开始全面理解Python！

今天观看 →

# VIDEO: Complete Python Bootcamp: Go from zero to hero in Python 3

Learn Python like a Professional! Start from the basics and go all the way to creating your own applications and games!

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ Learn to use Python professionally, learning both Python 2 and Python 3!
- ✓ Create games with Python, like Tic Tac Toe and Blackjack!
- ✓ Learn advanced Python features, like the collections module and how to work with timestamps!
- ✓ Learn to use Object Oriented Programming with classes!
- ✓ Understand complex topics, like decorators.
- ✓ Understand how to use both the Jupyter Notebook and create .py files
- ✓ Get an understanding of how to create GUIs in the Jupyter Notebook system!
- ✓ Build a complete understanding of Python from the ground up!

Watch Today →

# 第5章：日期和时间

## 第5.1节：将字符串解析为带时区的datetime对象

Python 3.2及以上版本支持%z格式，用于将字符串解析为datetime对象。

UTC偏移量格式为+HHMM或-HHMM（如果对象是天真的，则为空字符串）。

Python 3.x 版本 ≥ 3.2

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

对于其他版本的Python，可以使用外部库如dateutil，它可以快速将带时区的字符串解析为datetime对象。

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

变量 dt 现在是一个 datetime 对象，值如下：

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

## 第 5.2 节：构造带时区的日期时间

默认情况下，所有 datetime 对象都是天真的。要使它们具有时区感知，必须附加一个 tzinfo 对象，该对象根据日期和时间提供 UTC 偏移量和时区缩写。

### 固定偏移时区

对于与 UTC 有固定偏移的时区，在 Python 3.2 及以上版本中，datetime 模块提供了 timezone 类，这是 tzinfo 的具体实现，接受一个 timedelta 和一个（可选的）名称参数：

Python 3.x 版本 ≥ 3.2

```
from datetime import datetime, timedelta, timezone
JST = timezone(timedelta(hours=+9))

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00

print(dt.tzname())
# UTC+09:00

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=timezone(timedelta(hours=9), 'JST'))
print(dt.tzname())
# 'JST'
```

对于 Python 3.2 之前的版本，需要使用第三方库，例如dateutil。dateutil提供了一个等效的类 tzoffset，（截至版本 2.5.3）其参数形式为dateutil.tz.tzoffset(tzname,offset)，其中offset以秒为单位指定：

Python 3.x 版本 < 3.2

# Chapter 5: Date and Time

## Section 5.1: Parsing a string into a timezone aware datetime object

Python 3.2+ has support for %z format when [parsing a string](#) into a `datetime` object.

UTC offset in the form +HHMM or -HHMM (empty string if the object is naive).

Python 3.x Version ≥ 3.2

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

For other versions of Python, you can use an external library such as [dateutil](#), which makes parsing a string with timezone into a `datetime` object is quick.

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

The dt variable is now a `datetime` object with the following value:

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

## Section 5.2: Constructing timezone-aware datetimes

By default all `datetime` objects are naive. To make them timezone-aware, you must attach a `tzinfo` object, which provides the UTC offset and timezone abbreviation as a function of date and time.

### Fixed Offset Time Zones

For time zones that are a fixed offset from UTC, in Python 3.2+, the `datetime` module provides the `timezone` class, a concrete implementation of `tzinfo`, which takes a `timedelta` and an (optional) name parameter:

```
Python 3.x Version ≥ 3.2
from datetime import datetime, timedelta, timezone
JST = timezone(timedelta(hours=+9))

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00

print(dt.tzname())
# UTC+09:00

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=timezone(timedelta(hours=9), 'JST'))
print(dt.tzname())
# 'JST'
```

For Python versions before 3.2, it is necessary to use a third party library, such as [dateutil](#). dateutil provides an equivalent class, `tzoffset`, which (as of version 2.5.3) takes arguments of the form `dateutil.tz.tzoffset(tzname, offset)`, where `offset` is specified in seconds:

Python 3.x Version < 3.2

Python 2.x 版本 < 2.7

```
from datetime import datetime, timedelta
from dateutil import tz

JST = tz.tzoffset('JST', 9 * 3600) # 每小时 3600 秒
dt = datetime(2015, 1, 1, 12, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00
print(dt.tzname())
# 'JST'
```

## 使用夏令时的时区

对于使用夏令时的时区，Python 标准库没有提供标准类，因此需要使用第三方库。pytz 和 dateutil 是提供时区类的流行库。

除了静态时区外，dateutil 还提供使用夏令时的时区类（参见 tz 模块的文档）。你可以使用 tz.gettz() 方法获取时区对象，然后直接传递给 datetime 构造函数：

```
from datetime import datetime
from dateutil import tz
local = tz.gettz() # 本地时间
PT = tz.gettz('US/Pacific') # 太平洋时间

dt_l = datetime(2015, 1, 1, 12, tzinfo=local) # 我在东部标准时间
dt_pst = datetime(2015, 1, 1, 12, tzinfo=PT)
dt_pdt = datetime(2015, 7, 1, 12, tzinfo=PT) # 夏令时自动处理
print(dt_l)
# 2015-01-01 12:00:00-05:00
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-07-01 12:00:00-07:00
```

注意：从版本2.5.3开始，dateutil 无法正确处理模糊的日期时间，总是默认使用较晚的日期。无法构造一个带有dateutil时区的对象，例如 2015-11-01 1:30 EDT-4，因为这是夏令时转换期间。

使用pytz时，所有边界情况都能正确处理，但pytz时区不应直接通过构造函数附加到时间区。相反，应该使用时区的localize

方法附加pytz时区：

```
from datetime import datetime, timedelta
import pytz

PT = pytz.timezone('US/Pacific')
dt_pst = PT.localize(datetime(2015, 1, 1, 12))
dt_pdt = PT.localize(datetime(2015, 11, 1, 0, 30))
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-11-01 00:30:00-07:00
```

请注意，如果对一个带有pytz时区信息的日期时间进行算术运算，必须要么在UTC时间下进行计算（如果你想要绝对的经过时间），要么必须对结果调用normalize()方法：

Python 2.x Version < 2.7

```
from datetime import datetime, timedelta
from dateutil import tz

JST = tz.tzoffset('JST', 9 * 3600) # 3600 seconds per hour
dt = datetime(2015, 1, 1, 12, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00
print(dt.tzname())
# 'JST'
```

## Zones with daylight savings time

For zones with daylight savings time, python standard libraries do not provide a standard class, so it is necessary to use a third party library. [pytz](#) and [dateutil](#) are popular libraries providing time zone classes.

In addition to static time zones, dateutil provides time zone classes that use daylight savings time (see [the documentation for the tz module](#)). You can use the tz.gettz() method to get a time zone object, which can then be passed directly to the `datetime` constructor:

```
from datetime import datetime
from dateutil import tz
local = tz.gettz() # Local time
PT = tz.gettz('US/Pacific') # Pacific time

dt_l = datetime(2015, 1, 1, 12, tzinfo=local) # I am in EST
dt_pst = datetime(2015, 1, 1, 12, tzinfo=PT)
dt_pdt = datetime(2015, 7, 1, 12, tzinfo=PT) # DST is handled automatically
print(dt_l)
# 2015-01-01 12:00:00-05:00
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-07-01 12:00:00-07:00
```

**CAUTION:** As of version 2.5.3, dateutil does not handle ambiguous datetimes correctly, and will always default to the *later* date. There is no way to construct an object with a dateutil timezone representing, for example 2015-11-01 1:30 EDT-4, since this is *during* a daylight savings time transition.

All edge cases are handled properly when using pytz, but pytz time zones should *not* be directly attached to time zones through the constructor. Instead, a pytz time zone should be attached using the time zone's localize method:

```
from datetime import datetime, timedelta
import pytz

PT = pytz.timezone('US/Pacific')
dt_pst = PT.localize(datetime(2015, 1, 1, 12))
dt_pdt = PT.localize(datetime(2015, 11, 1, 0, 30))
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-11-01 00:30:00-07:00
```

Be aware that if you perform datetime arithmetic on a pytz-aware time zone, you must either perform the calculations in UTC (if you want absolute elapsed time), or you must call normalize() on the result:

```

dt_new = dt_pdt + timedelta(hours=3) # 这应该是PST时间的凌晨2:30
print(dt_new)
# 2015-11-01 03:30:00-07:00
dt_corrected = PT.normalize(dt_new)
print(dt_corrected)
# 2015-11-01 02:30:00-08:00

```

## 第5.3节：计算时间差

timedelta模块在计算时间差时非常有用：

```

from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23) # datetime.datetime(2016, 05, 23, 0, 0, 0)

```

创建新的datetime对象时，指定时间是可选的

```
delta = now-then
```

delta的类型是timedelta

```

print(delta.days)
# 60
print(delta.seconds)
# 40826

```

获取n天后的日期和n天前的日期，我们可以使用：

**n天后的日期：**

```

def get_n_days_after_date(date_format="%d %B %Y", add_days=120):
    date_n_days_after = datetime.datetime.now() + timedelta(days=add_days)
    return date_n_days_after.strftime(date_format)

```

**n天前的日期：**

```

def get_n_days_before_date(self, date_format="%d %B %Y", days_before=120):
    date_n_days_ago = datetime.datetime.now() - timedelta(days=days_before)
    return date_n_days_ago.strftime(date_format)

```

## 第5.4节：基本日期时间对象的使用

datetime 模块包含三种主要类型的对象——日期（date）、时间（time）和日期时间（datetime）。

```

import datetime

# 日期对象
today = datetime.date.today()
new_year = datetime.date(2017, 01, 01) #datetime.date(2017, 1, 1)

# 时间对象
noon = datetime.time(12, 0, 0) #datetime.time(12, 0)

# 当前日期时间
now = datetime.datetime.now()

```

```

dt_new = dt_pdt + timedelta(hours=3) # This should be 2:30 AM PST
print(dt_new)
# 2015-11-01 03:30:00-07:00
dt_corrected = PT.normalize(dt_new)
print(dt_corrected)
# 2015-11-01 02:30:00-08:00

```

## Section 5.3: Computing time differences

the timedelta module comes in handy to compute differences between times:

```

from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23) # datetime.datetime(2016, 05, 23, 0, 0, 0)

```

Specifying time is optional when creating a new `datetime` object

```
delta = now-then
```

delta is of type timedelta

```

print(delta.days)
# 60
print(delta.seconds)
# 40826

```

To get n day's after and n day's before date we could use:

**n day's after date:**

```

def get_n_days_after_date(date_format="%d %B %Y", add_days=120):
    date_n_days_after = datetime.datetime.now() + timedelta(days=add_days)
    return date_n_days_after.strftime(date_format)

```

**n day's before date:**

```

def get_n_days_before_date(self, date_format="%d %B %Y", days_before=120):
    date_n_days_ago = datetime.datetime.now() - timedelta(days=days_before)
    return date_n_days_ago.strftime(date_format)

```

## Section 5.4: Basic datetime objects usage

The datetime module contains three primary types of objects - date, time, and datetime.

```

import datetime

# Date object
today = datetime.date.today()
new_year = datetime.date(2017, 01, 01) #datetime.date(2017, 1, 1)

# Time object
noon = datetime.time(12, 0, 0) #datetime.time(12, 0)

# Current datetime
now = datetime.datetime.now()

```

```
# 日期时间对象  
millenium_turn = datetime.datetime(2000, 1, 1, 0, 0) #datetime.datetime(2000, 1, 1, 0, 0)
```

这些对象的算术运算仅支持同一数据类型之间，使用不同类型的实例进行简单算术运算将导致 `TypeError`。

```
# 从 today 中减去 noon  
中午-今天  
回溯（最近一次调用最后）：  
文件 "<stdin>", 第 1 行, 在 <模块>  
类型错误：不支持的操作数类型 (s) 用于 -: '日期时间.time' 和 '日期时间.date'  
但是, 在类型之间转换是直接的。  
  
# 改为这样做  
print('自千年纪午夜以来的时间：',  
      日期时间.日期时间(今天.year, 今天.month, 今天.day) - 千年纪转折点)  
  
# 或者这样  
print('自千年纪中午以来的时间：',  
      日期时间.日期时间.combine(今天, 中午) - 千年纪转折点)
```

## 第5.5节：时区切换

要切换时区，您需要具有时区感知的日期时间对象。

```
from datetime import datetime  
from dateutil import tz  
  
utc = tz.tzutc()  
local = tz.tzlocal()  
  
utc_now = datetime.utcnow()  
utc_now # 不是时区感知的。  
  
utc_now = utc_now.replace(tzinfo=utc)  
utc_now # 带时区信息的时间。  
  
local_now = utc_now.astimezone(local)  
local_now # 转换为本地时间。
```

```
# Datetime object  
millenium_turn = datetime.datetime(2000, 1, 1, 0, 0) #datetime.datetime(2000, 1, 1, 0, 0)
```

Arithmetic operations for these objects are only supported within same datatype and performing simple arithmetic with instances of different types will result in a `TypeError`.

```
# subtraction of noon from today  
noon-today  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for -: 'datetime.time' and 'datetime.date'  
However, it is straightforward to convert between types.  
  
# Do this instead  
print('Time since the millennium at midnight: ',  
      datetime.datetime(today.year, today.month, today.day) - millenium_turn)  
  
# Or this  
print('Time since the millennium at noon: ',  
      datetime.datetime.combine(today, noon) - millenium_turn)
```

## Section 5.5: Switching between time zones

To switch between time zones, you need `datetime` objects that are timezone-aware.

```
from datetime import datetime  
from dateutil import tz  
  
utc = tz.tzutc()  
local = tz.tzlocal()  
  
utc_now = datetime.utcnow()  
utc_now # Not timezone-aware.  
  
utc_now = utc_now.replace(tzinfo=utc)  
utc_now # Timezone-aware.  
  
local_now = utc_now.astimezone(local)  
local_now # Converted to local time.
```

## 第5.6节：简单日期运算

日期不是孤立存在的。通常你需要计算日期之间的时间差，或者确定明天的日期。这可以通过`timedelta`对象来实现

```
import datetime  
  
today = datetime.date.today()  
print('今天:', today)  
  
yesterday = today - datetime.timedelta(days=1)  
print('昨天:', yesterday)  
  
tomorrow = today + datetime.timedelta(days=1)  
print('明天:', tomorrow)  
  
print('明天和昨天之间的时间差:', tomorrow - yesterday)
```

## Section 5.6: Simple date arithmetic

Dates don't exist in isolation. It is common that you will need to find the amount of time between dates or determine what the date will be tomorrow. This can be accomplished using `timedelta` objects

```
import datetime  
  
today = datetime.date.today()  
print('Today:', today)  
  
yesterday = today - datetime.timedelta(days=1)  
print('Yesterday:', yesterday)  
  
tomorrow = today + datetime.timedelta(days=1)  
print('Tomorrow:', tomorrow)  
  
print('Time between tomorrow and yesterday:', tomorrow - yesterday)
```

这将产生类似以下的结果：

```
今天：2016-04-15  
昨天：2016-04-14  
明天：2016-04-16  
明天和昨天的差异：2天，0:00:00
```

## 第5.7节：将时间戳转换为日期时间

datetime模块可以将POSIX时间戳转换为ITC日期时间对象。

纪元时间是1970年1月1日午夜。

```
import time  
from datetime import datetime  
seconds_since_epoch=time.time() #1469182681.709  
  
utc_date=datetime.utcnow(timestamp(seconds_since_epoch)) #datetime.datetime(2016, 7, 22, 10, 18, 1,  
709000)
```

## 第5.8节：准确地从日期中减去月份

使用calendar模块

```
import calendar  
from datetime import date  
  
def monthdelta(date, delta):  
    m, y = (date.month+delta) % 12, date.year + ((date.month)+delta-1) // 12  
    if not m: m = 12  
    d = min(date.day, calendar.monthrange(y, m)[1])  
    return date.replace(day=d,month=m, year=y)  
  
next_month = monthdelta(date.today(), 1) #datetime.date(2016, 10, 23)
```

使用dateutils模块

```
import datetime  
import dateutil.relativedelta  
  
d = datetime.strptime("2013-03-31", "%Y-%m-%d")  
d2 = d - dateutil.relativedelta.relativedelta(months=1) #datetime.datetime(2013, 2, 28, 0, 0)
```

## 第5.9节：使用最少的库解析任意ISO 8601时间戳

Python对解析ISO 8601时间戳的支持有限。对于strptime，你需要确切知道它的格式。复杂之处在于，datetime的字符串化是一个ISO 8601时间戳，使用空格作为分隔符，并带有6位小数：

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))  
# '2016-07-22 09:25:59.555555'
```

但如果小数部分为0，则不输出小数部分

This will produce results similar to:

```
Today: 2016-04-15  
Yesterday: 2016-04-14  
Tomorrow: 2016-04-16  
Difference between tomorrow and yesterday: 2 days, 0:00:00
```

## Section 5.7: Converting timestamp to datetime

The `datetime` module can convert a POSIX `timestamp` to a ITC `datetime` object.

The Epoch is January 1st, 1970 midnight.

```
import time  
from datetime import datetime  
seconds_since_epoch=time.time() #1469182681.709  
  
utc_date=datetime.utcnow(timestamp(seconds_since_epoch)) #datetime.datetime(2016, 7, 22, 10, 18, 1,  
709000)
```

## Section 5.8: Subtracting months from a date accurately

Using the `calendar` module

```
import calendar  
from datetime import date  
  
def monthdelta(date, delta):  
    m, y = (date.month+delta) % 12, date.year + ((date.month)+delta-1) // 12  
    if not m: m = 12  
    d = min(date.day, calendar.monthrange(y, m)[1])  
    return date.replace(day=d,month=m, year=y)  
  
next_month = monthdelta(date.today(), 1) #datetime.date(2016, 10, 23)
```

Using the `dateutils` module

```
import datetime  
import dateutil.relativedelta  
  
d = datetime.strptime("2013-03-31", "%Y-%m-%d")  
d2 = d - dateutil.relativedelta.relativedelta(months=1) #datetime.datetime(2013, 2, 28, 0, 0)
```

## Section 5.9: Parsing an arbitrary ISO 8601 timestamp with minimal libraries

Python has only limited support for parsing ISO 8601 timestamps. For `strptime` you need to know exactly what format it is in. As a complication the stringification of a `datetime` is an ISO 8601 timestamp, with space as a separator and 6 digit fraction:

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))  
# '2016-07-22 09:25:59.555555'
```

but if the fraction is 0, no fractional part is output

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

但这两种形式需要`strptime`使用不同的格式。此外，`strptime`完全不支持解析带有:的分钟时区，因此2016-07-22 09:25:59+0300可以被解析，但标准格式2016-07-22 09:25:59+03:00不能。

有一个名为`iso8601`的单文件库，它能正确解析ISO 8601时间戳，仅限于此。

它支持分数和时区，以及T分隔符，所有这些都通过一个函数实现：

```
import iso8601
iso8601.parse_date('2016-07-22 09:25:59')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22 09:25:59+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<FixedOffset '+03:00' ...>)
iso8601.parse_date('2016-07-22 09:25:59Z')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22T09:25:59.000111+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, 111, tzinfo=<FixedOffset '+03:00' ...>)
```

如果未设置时区，`iso8601.parse_date` 默认使用UTC。默认时区可以通过`default_zone`关键字参数更改。值得注意的是，如果该参数设置为None而非默认值，则那些没有明确时区的时间戳将作为无时区（naive）日期时间返回：

```
iso8601.parse_date('2016-07-22T09:25:59', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59)
iso8601.parse_date('2016-07-22T09:25:59Z', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
```

## 第5.10节：获取ISO 8601时间戳

### 无时区，带微秒

```
from datetime import datetime

datetime.now().isoformat()
# 输出: '2016-07-31T23:08:20.886783'
```

### 带时区，带微秒

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).isoformat()
# 输出: '2016-07-31T23:09:43.535074-07:00'
```

### 带时区，不带微秒

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# 输出: '2016-07-31T23:10:30-07:00'
```

更多关于ISO 8601格式的信息，请参见ISO 8601。

## 第5.11节：解析带有短时区名称的字符串

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

But these 2 forms need a *different* format for `strptime`. Furthermore, `strptime` does not support at all parsing minute timezones that have a:`in` it, thus 2016-07-22 09:25:59+0300 can be parsed, but the standard format 2016-07-22 09:25:59+03:00` cannot.

There is a [single-file library called `iso8601`](#) which properly parses ISO 8601 timestamps and only them.

It supports fractions and timezones, and the T separator all with a single function:

```
import iso8601
iso8601.parse_date('2016-07-22 09:25:59')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22 09:25:59+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<FixedOffset '+03:00' ...>)
iso8601.parse_date('2016-07-22 09:25:59Z')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22T09:25:59.000111+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, 111, tzinfo=<FixedOffset '+03:00' ...>)
```

If no timezone is set, `iso8601.parse_date` defaults to UTC. The default zone can be changed with `default_zone` keyword argument. Notably, if this is `None` instead of the default, then those timestamps that do not have an explicit timezone are returned as naive datetimes instead:

```
iso8601.parse_date('2016-07-22T09:25:59', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59)
iso8601.parse_date('2016-07-22T09:25:59Z', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
```

## Section 5.10: Get an ISO 8601 timestamp

### Without timezone, with microseconds

```
from datetime import datetime

datetime.now().isoformat()
# Out: '2016-07-31T23:08:20.886783'
```

### With timezone, with microseconds

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).isoformat()
# Out: '2016-07-31T23:09:43.535074-07:00'
```

### With timezone, without microseconds

```
from datetime import datetime
from dateutil.tz import tzlocal

datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# Out: '2016-07-31T23:10:30-07:00'
```

See [ISO 8601](#) for more information about the ISO 8601 format.

## Section 5.11: Parsing a string with a short time zone name into

## 一个带时区信息的日期时间对象

使用`dateutil`库，如前面解析带时区时间戳的示例中所示，也可以解析带有指定“短”时区名称的时间戳。

对于使用短时区名称或缩写格式的日期，这些通常是模糊的（例如CST，可能是中部标准时间、中国标准时间、古巴标准时间等——更多信息可见[这里](#)），或者不一定在标准数据库中可用，必须指定时区缩写与 `tzinfo` 对象之间的映射关系。

```
from dateutil import tz
from dateutil.parser import parse

ET = tz.gettz('US/Eastern')
CT = tz.gettz('US/Central')
MT = tz.gettz('US/Mountain')
PT = tz.gettz('US/Pacific')

us_tzinfos = {'CST': CT, 'CDT': CT,
              'EST': ET, 'EDT': ET,
              'MST': MT, 'MDT': MT,
              'PST': PT, 'PDT': PT}

dt_est = parse('2014-01-02 04:00:00 EST', tzinfos=us_tzinfos)
dt_pst = parse('2016-03-11 16:00:00 PST', tzinfos=us_tzinfos)
```

运行此代码后：

```
dt_est
# datetime.datetime(2014, 1, 2, 4, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Eastern'))
dt_pst
# datetime.datetime(2016, 3, 11, 16, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Pacific'))
```

值得注意的是，如果使用`pytz`时区配合此方法，时区将无法正确本地化：

```
from dateutil.parser import parse
import pytz

EST = pytz.timezone('America/New_York')
dt = parse('2014-02-03 09:17:00 EST', tzinfos={'EST': EST})
```

这只是将`pytz`时区附加到`datetime`对象上：

```
dt.tzinfo # 将是本地平均时间！
# <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

如果使用此方法，解析后你可能需要重新本地化日期时间的简单部分：

```
dt_fixed = dt.tzinfo.localize(dt.replace(tzinfo=None))
dt_fixed.tzinfo # 现在是东部标准时间 (EST)。
# <DstTzInfo 'America/New_York' EST-1 day, 19:00:00 STD>
```

## 第5.12节：模糊日期时间解析（从文本中提取日期时间）

可以使用`dateutil`解析器的“模糊”模式从文本中提取日期，其中包含的组件

## a timezone aware datetime object

Using the `dateutil` library as in the previous example on parsing timezone-aware timestamps, it is also possible to parse timestamps with a specified "short" time zone name.

For dates formatted with short time zone names or abbreviations, which are generally ambiguous (e.g. CST, which could be Central Standard Time, China Standard Time, Cuba Standard Time, etc - more can be found [here](#)) or not necessarily available in a standard database, it is necessary to specify a mapping between time zone abbreviation and `tzinfo` object.

```
from dateutil import tz
from dateutil.parser import parse

ET = tz.gettz('US/Eastern')
CT = tz.gettz('US/Central')
MT = tz.gettz('US/Mountain')
PT = tz.gettz('US/Pacific')

us_tzinfos = {'CST': CT, 'CDT': CT,
              'EST': ET, 'EDT': ET,
              'MST': MT, 'MDT': MT,
              'PST': PT, 'PDT': PT}

dt_est = parse('2014-01-02 04:00:00 EST', tzinfos=us_tzinfos)
dt_pst = parse('2016-03-11 16:00:00 PST', tzinfos=us_tzinfos)
```

After running this:

```
dt_est
# datetime.datetime(2014, 1, 2, 4, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Eastern'))
dt_pst
# datetime.datetime(2016, 3, 11, 16, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Pacific'))
```

It is worth noting that if using a `pytz` time zone with this method, it will *not* be properly localized:

```
from dateutil.parser import parse
import pytz

EST = pytz.timezone('America/New_York')
dt = parse('2014-02-03 09:17:00 EST', tzinfos={'EST': EST})
```

This simply attaches the `pytz` time zone to the `datetime`:

```
dt.tzinfo # Will be in Local Mean Time!
# <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

If using this method, you should probably re-localize the naive portion of the `datetime` after parsing:

```
dt_fixed = dt.tzinfo.localize(dt.replace(tzinfo=None))
dt_fixed.tzinfo # Now it's EST.
# <DstTzInfo 'America/New_York' EST-1 day, 19:00:00 STD>
```

## Section 5.12: Fuzzy datetime parsing (extracting datetime out of a text)

It is possible to extract a date out of a text using the `dateutil parser` in a "fuzzy" mode, where components of the

未被识别为日期部分的字符串将被忽略。

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
```

dt 现在是一个 datetime 对象，您将看到 `datetime.datetime(2047, 1, 1, 8, 21)` 被打印出来。

## 第5.13节：遍历日期

有时你想要遍历从开始日期到某个结束日期的日期范围。你可以使用 `datetime` 库和 `timedelta` 对象来实现：

```
import datetime

# 每步的天数大小
day_delta = datetime.timedelta(days=1)

start_date = datetime.date.today()
end_date = start_date + 7*day_delta

for i in range((end_date - start_date).days):
    print(start_date + i*day_delta)
```

输出结果为：

```
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

string not recognized as being part of a date are ignored.

```
from dateutil.parser import parse

dt = parse("Today is January 1, 2047 at 8:21:00AM", fuzzy=True)
print(dt)
```

dt 现在是一个 `datetime object`，您将看到 `datetime.datetime(2047, 1, 1, 8, 21)` 被打印出来。

## Section 5.13: Iterate over dates

Sometimes you want to iterate over a range of dates from a start date to some end date. You can do it using `datetime` library and `timedelta` object:

```
import datetime

# The size of each step in days
day_delta = datetime.timedelta(days=1)

start_date = datetime.date.today()
end_date = start_date + 7*day_delta

for i in range((end_date - start_date).days):
    print(start_date + i*day_delta)
```

Which produces:

```
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

# 第6章：日期格式化

## 第6.1节：两个日期时间之间的时间间隔

```
from datetime import datetime

a = datetime(2016, 10, 06, 0, 0, 0)
b = datetime(2016, 10, 01, 23, 59, 59)

a-b
# datetime.timedelta(4, 1)

(a-b).days
# 4
(a-b).total_seconds()
# 518399.0
```

## 第6.2节：将datetime对象输出为字符串

使用C标准格式代码。

```
from datetime import datetime
datetime_for_string = datetime(2016, 10, 1, 0, 0)
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strftime(datetime_for_string, datetime_string_format)
# 2016年10月01日, 00:00:00
```

## 第6.3节：将字符串解析为datetime对象

使用C标准格式代码。

```
from datetime import datetime
datetime_string = 'Oct 1 2016, 00:00:00'
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strptime(datetime_string, datetime_string_format)
# datetime.datetime(2016, 10, 1, 0, 0)
```

# Chapter 6: Date Formatting

## Section 6.1: Time between two date-times

```
from datetime import datetime

a = datetime(2016, 10, 06, 0, 0, 0)
b = datetime(2016, 10, 01, 23, 59, 59)

a-b
# datetime.timedelta(4, 1)

(a-b).days
# 4
(a-b).total_seconds()
# 518399.0
```

## Section 6.2: Outputting datetime object to string

Uses C standard [format codes](#).

```
from datetime import datetime
datetime_for_string = datetime(2016, 10, 1, 0, 0)
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strftime(datetime_for_string, datetime_string_format)
# Oct 01 2016, 00:00:00
```

## Section 6.3: Parsing string to datetime object

Uses C standard [format codes](#).

```
from datetime import datetime
datetime_string = 'Oct 1 2016, 00:00:00'
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strptime(datetime_string, datetime_string_format)
# datetime.datetime(2016, 10, 1, 0, 0)
```

# 第7章：枚举

## 第7.1节：创建枚举（Python 2.4 到 3.3）

枚举已从Python 3.4回移植到Python 2.4至Python 3.3。你可以从PyPI获取这个 [enum34](#) 回移植包。

```
pip install enum34
```

枚举的创建方式与Python 3.4及以上版本相同

```
from enum import Enum

class Color(Enum):
    red = 1
    green = 2
    blue = 3

print(Color.red) # Color.red
print(Color(1)) # Color.red
print(Color['red']) # Color.red
```

## 第7.2节：迭代

枚举是可迭代的：

```
class Color(Enum):
    red = 1
    green = 2
    blue = 3

[c for c in Color] # [<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
```

# Chapter 7: Enum

## Section 7.1: Creating an enum (Python 2.4 through 3.3)

Enums have been backported from Python 3.4 to Python 2.4 through Python 3.3. You can get this the [enum34](#) backport from PyPI.

```
pip install enum34
```

Creation of an enum is identical to how it works in Python 3.4+

```
from enum import Enum

class Color(Enum):
    red = 1
    green = 2
    blue = 3

print(Color.red) # Color.red
print(Color(1)) # Color.red
print(Color['red']) # Color.red
```

## Section 7.2: Iteration

Enums are iterable:

```
class Color(Enum):
    red = 1
    green = 2
    blue = 3

[c for c in Color] # [<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
```

# 第8章：集合

## 第8.1节：集合的运算

### 与其他集合

```
# 交集
{1, 2, 3, 4, 5}.intersection({3, 4, 5, 6}) # {3, 4, 5}
{1, 2, 3, 4, 5} & {3, 4, 5, 6} # {3, 4, 5}

# 并集
{1, 2, 3, 4, 5}.union({3, 4, 5, 6}) # {1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5} | {3, 4, 5, 6} # {1, 2, 3, 4, 5, 6}

# 差集
{1, 2, 3, 4}.difference({2, 3, 5}) # {1, 4}
{1, 2, 3, 4} - {2, 3, 5} # {1, 4}

# 差集运算
{1, 2, 3, 4}.symmetric_difference({2, 3, 5}) # {1, 4, 5}
{1, 2, 3, 4} ^ {2, 3, 5} # {1, 4, 5}

# 对称差集运算
{1, 2}.issuperset({1, 2, 3}) # False
{1, 2} >= {1, 2, 3} # False

# 超集检查
{1, 2}.issubset({1, 2, 3}) # True
{1, 2} <= {1, 2, 3} # True

# 子集检查
{1, 2}.isdisjoint({3, 4}) # True
{1, 2}.isdisjoint({1, 4}) # False
```

### 单个元素操作

```
# 存在性检查
2 in {1,2,3} # True
4 in {1,2,3} # False
4 not in {1,2,3} # True

# 添加与删除
s = {1,2,3}
s.add(4) # s == {1,2,3,4}

s.discard(3) # s == {1,2,4}
s.discard(5) # s == {1,2,4}

s.remove(2) # s == {1,4}
s.remove(2) # KeyError!
```

集合操作返回新的集合，但有对应的原地操作版本：

方法	原地操作	原地方法
并集	s  = t	更新
交集	s &= t	交集更新
差集	s -= t	差集更新

# Chapter 8: Set

## Section 8.1: Operations on sets

### with other sets

```
# Intersection
{1, 2, 3, 4, 5}.intersection({3, 4, 5, 6}) # {3, 4, 5}
{1, 2, 3, 4, 5} & {3, 4, 5, 6} # {3, 4, 5}

# Union
{1, 2, 3, 4, 5}.union({3, 4, 5, 6}) # {1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5} | {3, 4, 5, 6} # {1, 2, 3, 4, 5, 6}

# Difference
{1, 2, 3, 4}.difference({2, 3, 5}) # {1, 4}
{1, 2, 3, 4} - {2, 3, 5} # {1, 4}

# Symmetric difference with
{1, 2, 3, 4}.symmetric_difference({2, 3, 5}) # {1, 4, 5}
{1, 2, 3, 4} ^ {2, 3, 5} # {1, 4, 5}

# Superset check
{1, 2}.issuperset({1, 2, 3}) # False
{1, 2} >= {1, 2, 3} # False

# Subset check
{1, 2}.issubset({1, 2, 3}) # True
{1, 2} <= {1, 2, 3} # True

# Disjoint check
{1, 2}.isdisjoint({3, 4}) # True
{1, 2}.isdisjoint({1, 4}) # False
```

### with single elements

```
# Existence check
2 in {1,2,3} # True
4 in {1,2,3} # False
4 not in {1,2,3} # True

# Add and Remove
s = {1,2,3}
s.add(4) # s == {1,2,3,4}

s.discard(3) # s == {1,2,4}
s.discard(5) # s == {1,2,4}

s.remove(2) # s == {1,4}
s.remove(2) # KeyError!
```

Set operations return new sets, but have the corresponding in-place versions:

method	in-place operation	in-place method
union	s  = t	update
intersection	s &= t	intersection_update
difference	s -= t	difference_update

对称差集  $s \Delta t$

对称差集更新

例如：

```
s = {1, 2}  
s.update({3, 4}) # s == {1, 2, 3, 4}
```

## 第8.2节：获取列表中的唯一元素

假设你有一个餐厅列表——也许你是从文件中读取的。你关心列表中唯一的餐厅。获取列表中唯一元素的最佳方法是将其转换为集合：

```
restaurants = ["麦当劳", "汉堡王", "麦当劳", "鸡鸡"]  
unique_restaurants = set(restaurants)  
print(unique_restaurants)  
# 输出 {'鸡鸡', '麦当劳', '汉堡王'}
```

注意，集合的顺序与原始列表不同；这是因为集合是无序的，就像字典dict一样。

这可以很容易地用Python内置的list函数转换回列表，得到另一个与原始列表相同但没有重复项的列表：

```
list(unique_restaurants)  
# ['鸡鸡', '麦当劳', '汉堡王']
```

这种写法也很常见，写成一行：

```
# 移除所有重复项并返回另一个列表  
list(set(餐厅))
```

现在可以再次对原始列表执行任何操作。

## 第8.3节：集合的集合

`{}{1,2}, {}{3,4}`

导致：

`TypeError: unhashable type: 'set'`

相反，使用frozenset：

```
{frozenset({1, 2}), frozenset({3, 4})}
```

## 第8.4节：使用方法和内置函数的集合操作

我们定义两个集合a和b

```
>>> a = {1, 2, 2, 3, 4}  
>>> b = {3, 3, 4, 4, 5}
```

注意：`{1}` 创建一个包含一个元素的集合，但 `{}` 创建一个空的 dict。创建空集合的正确方法是 `set()`。

`symmetric_difference s  $\Delta$  t`

`symmetric_difference_update`

For example:

```
s = {1, 2}  
s.update({3, 4}) # s == {1, 2, 3, 4}
```

## Section 8.2: Get the unique elements of a list

Let's say you've got a list of restaurants -- maybe you read it from a file. You care about the *unique* restaurants in the list. The best way to get the unique elements from a list is to turn it into a set:

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]  
unique_restaurants = set(restaurants)  
print(unique_restaurants)  
# prints {'Chicken Chicken', 'McDonald's', 'Burger King'}
```

Note that the set is not in the same order as the original list; that is because sets are *unordered*, just like `dicts`.

This can easily be transformed back into a List with Python's built in `list` function, giving another list that is the same list as the original but without duplicates:

```
list(unique_restaurants)  
# ['Chicken Chicken', 'McDonald's', 'Burger King']
```

It's also common to see this as one line:

```
# Removes all duplicates and returns another list  
list(set(restaurants))
```

Now any operations that could be performed on the original list can be done again.

## Section 8.3: Set of Sets

`{}{1,2}, {}{3,4}`

leads to:

`TypeError: unhashable type: 'set'`

Instead, use `frozenset`:

```
{frozenset({1, 2}), frozenset({3, 4})}
```

## Section 8.4: Set Operations using Methods and Builtins

We define two sets a and b

```
>>> a = {1, 2, 2, 3, 4}  
>>> b = {3, 3, 4, 4, 5}
```

NOTE: `{1}` creates a set of one element, but `{}` creates an empty `dict`. The correct way to create an empty set is `set()`.

## 交集

a.intersection(b) 返回一个新集合，包含同时存在于 a 和 b 中的元素

```
>>> a.intersection(b)
{3, 4}
```

## 并集

a.union(b) 返回一个新集合，包含存在于 a 或 b 中的元素

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

## 差集

a.difference(b) 返回一个新集合，包含存在于 a 但不在 b 中的元素

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
{5}
```

## 对称差集

a.symmetric\_difference(b) 返回一个新集合，包含存在于 a 或 b 中但不同时存在于两者中的元素

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

**注意:** a.symmetric\_difference(b) == b.symmetric\_difference(a)

## 子集和超集

c.issubset(a) 测试 c 的每个元素是否都在 a 中。

a.issuperset(c) 测试 a 的每个元素是否都包含 c 中的元素。

```
>>> c = {1, 2}
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

后者操作具有等效的运算符，如下所示：

方法	运算符
a.intersection(b)	a & b
a.union(b)	a b
a.difference(b)	a - b
a.symmetric_difference(b)	a ^ b
a.issubset(b)	a <= b
a.issuperset(b)	a >= b

## 不相交集合

## Intersection

a.intersection(b) returns a new set with elements present in both a and b

```
>>> a.intersection(b)
{3, 4}
```

## Union

a.union(b) returns a new set with elements present in either a and b

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

## Difference

a.difference(b) returns a new set with elements present in a but not in b

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
{5}
```

## Symmetric Difference

a.symmetric\_difference(b) returns a new set with elements present in either a or b but not in both

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

**NOTE:** a.symmetric\_difference(b) == b.symmetric\_difference(a)

## Subset and superset

c.issubset(a) tests whether each element of c is in a.

a.issuperset(c) tests whether each element of c is in a.

```
>>> c = {1, 2}
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

The latter operations have equivalent operators as shown below:

Method	Operator
a.intersection(b)	a & b
a.union(b)	a b
a.difference(b)	a - b
a.symmetric_difference(b)	a ^ b
a.issubset(b)	a <= b
a.issuperset(b)	a >= b

## Disjoint sets

如果集合 a 中没有元素也在集合 d 中，且反之亦然，则集合 a 和 d 是不相交的。

```
>>> d = {5, 6}
>>> a.isdisjoint(b) # {2, 3, 4} 同时存在于两个集合中
False
>>> a.isdisjoint(d)
True

# 这是一个等效的检查，但效率较低
>>> len(a & d) == 0
True

# 这甚至更低效
>>> a & d == set()
True
```

## 测试成员资格

内置的 in 关键字用于搜索出现的元素

```
>>> 1 in a
True
>>> 6 in a
False
```

## 长度

内置的 len() 函数返回集合中的元素数量

```
>>> len(a)
4
>>> len(b)
3
```

## 第8.5节：集合与多重集合

集合是无序且元素互不相同的集合。但有时我们希望处理无序且元素不一定互不相同的集合，并且跟踪元素的重复次数。

考虑以下示例：

```
>>> setA = {'a', 'b', 'b', 'c'}
>>> setA
set(['a', 'c', 'b'])
```

将字符串 'a'、'b'、'b'、'c' 保存到集合数据结构中时，我们丢失了 'b' 出现两次的这一信息。当然，将元素保存到列表中可以保留该信息

```
>>> listA = ['a', 'b', 'b', 'c']
>>> listA
['a', 'b', 'b', 'c']
```

但列表数据结构引入了额外且不必要的顺序，这会降低我们的计算速度。

为了实现多重集合，Python从2.7版本开始提供了collections模块中的Counter类：

Python 2.x 版本 ≥ 2.7

Sets a and d are disjoint if no element in a is also in d and vice versa.

```
>>> d = {5, 6}
>>> a.isdisjoint(b) # {2, 3, 4} are in both sets
False
>>> a.isdisjoint(d)
True

# This is an equivalent check, but less efficient
>>> len(a & d) == 0
True

# This is even less efficient
>>> a & d == set()
True
```

## Testing membership

The builtin in keyword searches for occurrences

```
>>> 1 in a
True
>>> 6 in a
False
```

## Length

The builtin len() function returns the number of elements in the set

```
>>> len(a)
4
>>> len(b)
3
```

## Section 8.5: Sets versus multisets

Sets are unordered collections of distinct elements. But sometimes we want to work with unordered collections of elements that are not necessarily distinct and keep track of the elements' multiplicities.

Consider this example:

```
>>> setA = {'a', 'b', 'b', 'c'}
>>> setA
set(['a', 'c', 'b'])
```

By saving the strings 'a', 'b', 'b', 'c' into a set data structure we've lost the information on the fact that 'b' occurs twice. Of course saving the elements to a list would retain this information

```
>>> listA = ['a', 'b', 'b', 'c']
>>> listA
['a', 'b', 'b', 'c']
```

but a list data structure introduces an extra unneeded ordering that will slow down our computations.

For implementing multisets Python provides the Counter class from the [collections](#) module (starting from version 2.7):

Python 2.x Version ≥ 2.7

```
>>> from collections import Counter  
>>> counterA = Counter(['a', 'b', 'b', 'c'])  
>>> counterA  
Counter({'b': 2, 'a': 1, 'c': 1})
```

Counter 是一种字典，其中元素作为字典的键存储，计数作为字典的值存储。和所有字典一样，它是一个无序集合。

```
>>> from collections import Counter  
>>> counterA = Counter(['a', 'b', 'b', 'c'])  
>>> counterA  
Counter({'b': 2, 'a': 1, 'c': 1})
```

Counter is a dictionary where elements are stored as dictionary keys and their counts are stored as dictionary values. And as all dictionaries, it is an unordered collection.

# 第9章：简单的数学运算符

## 数值类型及其元类

numbers模块包含数值类型的抽象元类：

子类	<a href="#">numbers.Number</a>	<a href="#">numbers.Integral</a>	<a href="#">numbers.Rational</a>	<a href="#">numbers.Real</a>	<a href="#">numbers.Complex</a>
<a href="#">bool</a>	✓	✓	✓	✓	✓
<a href="#">整数</a>	✓	✓	✓	✓	✓
<a href="#">fractions.Fraction</a>	✓	-	✓	✓	✓
<a href="#">浮点数</a>	✓	-	-	✓	✓
<a href="#">复数</a>	✓	-	-	-	✓
<a href="#">decimal.Decimal</a>	-	-	-	-	-

Python 自带常用的数学运算符，包括整数和浮点数的除法、乘法、幂运算、加法和减法。math 模块（包含在所有标准 Python 版本中）提供了扩展功能，如三角函数、开方运算、对数等更多功能。

## 第9.1节：除法

当两个操作数都是整数时，Python 执行整数除法。Python 的除法运算符行为在 Python 2.x 和 3.x 之间有所变化（另见整数除法）。

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x 版本  $\leq$  2.7

在 Python 2 中，'/' 运算符的结果取决于分子和分母的类型。

```
a / b          # = 1
a / c          # = 1.5
d / b          # = -2
b / a          # = 0
d / e          # = -1
```

注意，因为 a 和 b 都是 int 类型，结果也是 int 类型。

结果总是向下取整 (floor)。

因为 c 是浮点数，a / c 的结果是 float 类型。

你也可以使用 operator 模块：

```
import operator      # operator 模块提供了两个参数的算术函数
operator.div(a, b)  # = 1
operator.__div__(a, b) # = 1
```

Python 2.x 版本  $\geq$  2.2

如果你想要浮点数除法：

推荐：

# Chapter 9: Simple Mathematical Operators

## Numerical types and their metaclasses

The numbers module contains the abstract metaclasses for the numerical types:

subclasses	<a href="#">numbers.Number</a>	<a href="#">numbers.Integral</a>	<a href="#">numbers.Rational</a>	<a href="#">numbers.Real</a>	<a href="#">numbers.Complex</a>
<a href="#">bool</a>	✓	✓	✓	✓	✓
<a href="#">int</a>	✓	✓	✓	✓	✓
<a href="#">fractions.Fraction</a>	✓	-	✓	✓	✓
<a href="#">float</a>	✓	-	-	✓	✓
<a href="#">complex</a>	✓	-	-	-	✓
<a href="#">decimal.Decimal</a>	✓	-	-	-	-

Python does common mathematical operators on its own, including integer and float division, multiplication, exponentiation, addition, and subtraction. The math module (included in all standard Python versions) offers expanded functionality like trigonometric functions, root operations, logarithms, and many more.

## Section 9.1: Division

Python does integer division when both operands are integers. The behavior of Python's division operators have changed from Python 2.x and 3.x (see also Integer Division ).

```
a, b, c, d, e = 3, 2, 2.0, -3, 10
```

Python 2.x Version  $\leq$  2.7

In Python 2 the result of the '/' operator depends on the type of the numerator and denominator.

```
a / b          # = 1
a / c          # = 1.5
d / b          # = -2
b / a          # = 0
d / e          # = -1
```

Note that because both a and b are ints, the result is an int.

The result is always rounded down (floored).

Because c is a float, the result of a / c is a float.

You can also use the operator module:

```
import operator      # the operator module provides 2-argument arithmetic functions
operator.div(a, b)  # = 1
operator.__div__(a, b) # = 1
```

Python 2.x Version  $\geq$  2.2

What if you want float division:

Recommended:

```
from __future__ import division # 对整个模块应用 Python 3 风格的除法
a / b                         # = 1.5
a // b                         # = 1
```

好的（如果你不想应用到整个模块）：

```
a / (b * 1.0)                 # = 1.5
1.0 * a / b                   # = 1.5
a / b * 1.0                   # = 1.0    (注意运算顺序)
```

```
from operator import truediv
truediv(a, b)                 # = 1.5
```

不推荐（可能会引发 `TypeError`, 例如参数是复数时）：

```
float(a) / b                  # = 1.5
a / float(b)                   # = 1.5
```

Python 2.x 版本  $\geq$  2.2

Python 2 中的 '`//`' 运算符无论类型如何，都会强制进行向下取整除法。

```
a // b                         # = 1
a // c                         # = 1.0
```

Python 3.x 版本  $\geq$  3.0

在 Python 3 中，`/` 运算符执行“真”除法，无论类型如何。`//` 运算符执行向下取整除法并保持类型。

```
a / b                         # = 1.5
e / b                         # = 5.0
a // b                         # = 1
a // c                         # = 1.0
```

```
import operator      # operator 模块提供两个参数的算术函数
operator.truediv(a, b)    # = 1.5
operator.floordiv(a, b)   # = 1
operator.floordiv(a, c)   # = 1.0
```

可能的组合（内置类型）：

- `int` 和 `int`（在 Python 2 中返回 `int`, Python 3 中返回 `float`）
- `int` 和 `float`（返回 `float`）
- `int` 和 `complex`（返回 `complex`）
- `float` 和 `float`（返回 `float`）
- `float` 和 `complex`（返回 `complex`）
- `complex` 和 `complex`（返回 `complex`）

详见 [PEP 238](#) 了解更多信息。

## 第9.2节：加法

```
a, b = 1, 2
```

```
# 使用“+”运算符：
a + b                         # = 3
```

```
from __future__ import division # applies Python 3 style division to the entire module
a / b                         # = 1.5
a // b                         # = 1
```

Okay (if you don't want to apply to the whole module):

```
a / (b * 1.0)                 # = 1.5
1.0 * a / b                   # = 1.5
a / b * 1.0                   # = 1.0    (careful with order of operations)
```

```
from operator import truediv
truediv(a, b)                 # = 1.5
```

Not recommended (may raise `TypeError`, eg if argument is complex):

```
float(a) / b                  # = 1.5
a / float(b)                   # = 1.5
```

Python 2.x Version  $\geq$  2.2

The '`//`' operator in Python 2 forces floored division regardless of type.

```
a // b                         # = 1
a // c                         # = 1.0
```

Python 3.x Version  $\geq$  3.0

In Python 3 the `/` operator performs 'true' division regardless of types. The `//` operator performs floor division and maintains type.

```
a / b                         # = 1.5
e / b                         # = 5.0
a // b                         # = 1
a // c                         # = 1.0
```

```
import operator      # the operator module provides 2-argument arithmetic functions
operator.truediv(a, b)    # = 1.5
operator.floordiv(a, b)   # = 1
operator.floordiv(a, c)   # = 1.0
```

Possible combinations (builtin types):

- `int` and `int`（在 Python 2 中返回 `int`，Python 3 中返回 `float`）
- `int` and `float`（返回 `float`）
- `int` and `complex`（返回 `complex`）
- `float` and `float`（返回 `float`）
- `float` and `complex`（返回 `complex`）
- `complex` and `complex`（返回 `complex`）

See [PEP 238](#) for more information.

## Section 9.2: Addition

```
a, b = 1, 2
```

```
# Using the "+" operator:
a + b                         # = 3
```

```
# 使用“就地”加法运算符“+=”进行加法并赋值：
a += b          # a = 3 (等同于 a = a + b)

import operator      # 包含示例中的两个参数算术函数

operator.add(a, b)    # = 5 因为 a 在此行之前被设置为3

# “+”运算符等同于：
a = operator.iadd(a, b)  # a = 5 因为 a 在此行之前被设置为3
```

可能的组合（内置类型）：

- `int` 和 `int` (结果为 `int`)
- `int` 和 `float` (结果为 `float`)
- `int` 和 `complex` (结果为 `complex`)
- `float` 和 `float` (结果为 `float`)
- `float` 和 `complex` (结果为 `complex`)
- `complex` 和 `complex` (结果为 `complex`)

注意：`+` 运算符也用于连接字符串、列表和元组：

```
"first string" + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

## 第9.3节：指数运算

```
a, b = 2, 3

(a ** b)          # = 8
pow(a, b)         # = 8

import math
math.pow(a, b)    # = 8.0 (始终为浮点数；不允许复数结果)

import operator
operator.pow(a, b) # = 8
```

内置的`pow`和`math.pow`之间的另一个区别是，内置的`pow`可以接受三个参数：

```
a, b, c = 2, 3, 2

pow(2, 3, 2)      # 0, 计算(2 ** 3) % 2, 但根据Python文档,
                  # 这样做更高效
```

### 特殊函数

函数`math.sqrt(x)`计算 `x` 的平方根。

```
import math
import cmath
c = 4
math.sqrt(c)        # = 2.0 (始终为浮点数；不允许复数结果)
cmath.sqrt(c)       # = (2+0j) (始终为复数)
```

要计算其他根，例如立方根，将数字提升到根的次数的倒数。这可以用任何指数函数或运算符完成。

```
# Using the “in-place” “+=” operator to add and assign:
a += b          # a = 3 (equivalent to a = a + b)

import operator      # contains 2 argument arithmetic functions for the examples

operator.add(a, b)    # = 5 since a is set to 3 right before this line

# The “+” operator is equivalent to:
a = operator.iadd(a, b)  # a = 5 since a is set to 3 right before this line
```

Possible combinations (builtin types):

- `int` 和 `int` (gives an `int`)
- `int` 和 `float` (gives a `float`)
- `int` 和 `complex` (gives a `complex`)
- `float` 和 `float` (gives a `float`)
- `float` 和 `complex` (gives a `complex`)
- `complex` 和 `complex` (gives a `complex`)

Note: the `+` operator is also used for concatenating strings, lists and tuples:

```
"first string" + "second string"    # = 'first string second string'

[1, 2, 3] + [4, 5, 6]                # = [1, 2, 3, 4, 5, 6]
```

## Section 9.3: Exponentiation

```
a, b = 2, 3

(a ** b)          # = 8
pow(a, b)         # = 8

import math
math.pow(a, b)    # = 8.0 (always float; does not allow complex results)

import operator
operator.pow(a, b) # = 8
```

Another difference between the built-in `pow` and `math.pow` is that the built-in `pow` can accept three arguments:

```
a, b, c = 2, 3, 2

pow(2, 3, 2)      # 0, calculates (2 ** 3) % 2, but as per Python docs,
                  # does so more efficiently
```

### Special functions

The function `math.sqrt(x)` calculates the square root of `x`.

```
import math
import cmath
c = 4
math.sqrt(c)        # = 2.0 (always float; does not allow complex results)
cmath.sqrt(c)       # = (2+0j) (always complex)
```

To compute other roots, such as a cube root, raise the number to the reciprocal of the degree of the root. This could be done with any of the exponential functions or operator.

```
import math
x = 8
math.pow(x, 1/3) # 计算结果为 2.0
x**(1/3) # 计算结果为 2.0
```

函数math.exp(x)计算  $e^{** x}$ 。

```
math.exp(0) # 1.0
math.exp(1) # 2.718281828459045 (e)
```

函数math.expm1(x)计算  $e^{** x} - 1$ 。当 x 较小时，这比

math.exp(x) 提供显著更高的精度。

```
math.expm1(0) # 0.0
```

```
math.exp(1e-6) - 1 # 1.0000004999621837e-06
math.expm1(1e-6) # 1.000000500001665e-06
# exact result # 1.0000005000016666708333341666...
```

## 第9.4节：三角函数

a, b = 1, 2

```
import math
```

```
math.sin(a) # 返回角度'a'的弧度正弦值
# 输出: 0.8414709848078965
```

```
math.cosh(b) # 返回角度'b'的反双曲余弦值
# 输出: 3.7621956910836314
```

```
math.atan(math.pi) # 返回'pi'的反正切值，单位为弧度
# 输出: 1.2626272556789115
```

```
math.hypot(a, b) # 返回欧几里得范数，与 math.sqrt(a*a + b*b) 相同
# 输出: 2.23606797749979
```

注意，math.hypot(x, y) 也是从原点 (0, 0) 到点 (x, y) 的向量长度（或欧几里得距离）。

要计算两点  $(x_1, y_1)$  和  $(x_2, y_2)$  之间的欧几里得距离，可以使用 math.hypot，方法如下

```
math.hypot(x2-x1, y2-y1)
```

要分别将弧度转换为角度和角度转换为弧度，可以使用 math.degrees 和 math.radians

```
math.degrees(a)
# 输出: 57.29577951308232
```

```
math.radians(57.29577951308232)
# 输出: 1.0
```

```
import math
x = 8
math.pow(x, 1/3) # evaluates to 2.0
x**(1/3) # evaluates to 2.0
```

The function math.exp(x) computes  $e^{** x}$ .

```
math.exp(0) # 1.0
math.exp(1) # 2.718281828459045 (e)
```

The function math.expm1(x) computes  $e^{** x} - 1$ . When x is small, this gives significantly better precision than  
math.exp(x) - 1.

```
math.expm1(0) # 0.0
```

```
math.exp(1e-6) - 1 # 1.0000004999621837e-06
math.expm1(1e-6) # 1.000000500001665e-06
# exact result # 1.0000005000016666708333341666...
```

## Section 9.4: Trigonometric Functions

a, b = 1, 2

```
import math
```

```
math.sin(a) # returns the sine of 'a' in radians
# Out: 0.8414709848078965
```

```
math.cosh(b) # returns the inverse hyperbolic cosine of 'b' in radians
# Out: 3.7621956910836314
```

```
math.atan(math.pi) # returns the arc tangent of 'pi' in radians
# Out: 1.2626272556789115
```

```
math.hypot(a, b) # returns the Euclidean norm, same as math.sqrt(a*a + b*b)
# Out: 2.23606797749979
```

Note that math.hypot(x, y) is also the length of the vector (or Euclidean distance) from the origin (0, 0) to the point (x, y).

To compute the Euclidean distance between two points  $(x_1, y_1)$  &  $(x_2, y_2)$  you can use math.hypot as follows

```
math.hypot(x2-x1, y2-y1)
```

To convert from radians -> degrees and degrees -> radians respectively use math.degrees and math.radians

```
math.degrees(a)
# Out: 57.29577951308232
```

```
math.radians(57.29577951308232)
# Out: 1.0
```

## 第9.5节：就地操作

在应用程序中，通常需要如下代码：

```
a = a + 1
```

或者

```
a = a * 2
```

这些就地操作有一个有效的快捷方式：

```
a += 1  
# 和  
a *= 2
```

任何数学运算符都可以在“=”字符前使用，以进行原地操作：

- -= 原地递减变量
- += 原地递增变量
- \*= 原地乘以变量
- /= 原地除以变量
- //= 原地取整除变量 # Python 3
- %= 原地取变量的模
- \*\*= 原地幂运算

其他原地运算符也存在于按位运算符 (^, | 等) 中

## 第9.6节：减法

```
a, b = 1, 2
```

# 使用“-”运算符：

```
b - a          # = 1
```

```
import operator      # 包含两个参数的算术函数  
operator.sub(b, a)  # = 1
```

可能的组合（内置类型）：

- `int` 和 `int` (结果为 `int`)
- `int` 和 `float` (结果为 `float`)
- `int` 和 `complex` (结果为 `complex`)
- `float` 和 `float` (结果为 `float`)
- `float` 和 `complex` (结果为 `complex`)
- `complex` 和 `complex` (结果为 `complex`)

## 第9.7节：乘法

```
a, b = 2, 3
```

```
a * b          # = 6
```

```
import operator
```

## Section 9.5: Inplace Operations

It is common within applications to need to have code like this:

```
a = a + 1
```

or

```
a = a * 2
```

There is an effective shortcut for these in place operations:

```
a += 1  
# and  
a *= 2
```

Any mathematic operator can be used before the '=' character to make an inplace operation:

- -= decrement the variable in place
- += increment the variable in place
- \*= multiply the variable in place
- /= divide the variable in place
- //= floor divide the variable in place # Python 3
- %= return the modulus of the variable in place
- \*\*= raise to a power in place

Other in place operators exist for the bitwise operators (^, | etc)

## Section 9.6: Subtraction

```
a, b = 1, 2
```

# Using the “-” operator:

```
b - a          # = 1
```

```
import operator      # contains 2 argument arithmetic functions  
operator.sub(b, a)  # = 1
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int`)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

## Section 9.7: Multiplication

```
a, b = 2, 3
```

```
a * b          # = 6
```

```
import operator
```

```
operator.mul(a, b) # = 6
```

可能的组合（内置类型）：

- `int` 和 `int` (结果为 `int`)
- `int` 和 `float` (结果为 `float`)
- `int` 和 `complex` (结果为 `complex`)
- `float` 和 `float` (结果为 `float`)
- `float` 和 `complex` (结果为 `complex`)
- `complex` 和 `complex` (结果为 `complex`)

注意：\* 运算符也用于字符串、列表和元组的重复连接：

```
3 * 'ab' # = 'ababab'  
3 * ('a', 'b') # = ('a', 'b', 'a', 'b', 'a', 'b')
```

## 第9.8节：对数

默认情况下，`math.log` 函数计算一个数的以 e 为底的对数。你可以选择性地指定第二个参数作为底数。

```
import math  
import cmath  
  
math.log(5) # = 1.6094379124341003  
# 可选的底数参数。默认是 math.e  
math.log(5, math.e) # = 1.6094379124341003  
cmath.log(5) # = (1.6094379124341003+0j)  
math.log(1000, 10) # 3.0 (总是返回浮点数)  
cmath.log(1000, 10) # (3+0j)
```

针对不同底数，存在`math.log` 函数的特殊变体。

```
# 以 e 为底的对数减 1 (对小值更高精度)  
math.log1p(5) # = 1.791759469228055  
  
# 以 2 为底的对数  
math.log2(8) # = 3.0  
  
# 以 10 为底的对数  
math.log10(100) # = 2.0  
cmath.log10(100) # = (2+0j)
```

## 第9.9节：取模运算

像许多其他语言一样，Python 使用 % 运算符来计算取模。

```
3%4 # 3  
10%2 # 0  
6%4 # 2
```

或者使用operator模块：

```
import operator  
  
operator.mod(3, 4) # 3
```

```
operator.mul(a, b) # = 6
```

Possible combinations (builtin types):

- `int` and `int` (gives an `int`)
- `int` and `float` (gives a `float`)
- `int` and `complex` (gives a `complex`)
- `float` and `float` (gives a `float`)
- `float` and `complex` (gives a `complex`)
- `complex` and `complex` (gives a `complex`)

Note: The \* operator is also used for repeated concatenation of strings, lists, and tuples:

```
3 * 'ab' # = 'ababab'  
3 * ('a', 'b') # = ('a', 'b', 'a', 'b', 'a', 'b')
```

## Section 9.8: Logarithms

By default, the `math.log` function calculates the logarithm of a number, base e. You can optionally specify a base as the second argument.

```
import math  
import cmath  
  
math.log(5) # = 1.6094379124341003  
# optional base argument. Default is math.e  
math.log(5, math.e) # = 1.6094379124341003  
cmath.log(5) # = (1.6094379124341003+0j)  
math.log(1000, 10) # 3.0 (always returns float)  
cmath.log(1000, 10) # (3+0j)
```

Special variations of the `math.log` function exist for different bases.

```
# Logarithm base e - 1 (higher precision for low values)  
math.log1p(5) # = 1.791759469228055  
  
# Logarithm base 2  
math.log2(8) # = 3.0  
  
# Logarithm base 10  
math.log10(100) # = 2.0  
cmath.log10(100) # = (2+0j)
```

## Section 9.9: Modulus

Like in many other languages, Python uses the % operator for calculating modulus.

```
3 % 4 # 3  
10 % 2 # 0  
6 % 4 # 2
```

Or by using the `operator` module:

```
import operator  
  
operator.mod(3, 4) # 3
```

```
operator.mod(10 , 2)      # 0  
operator.mod(6 , 4)      # 2
```

你也可以使用负数。

```
-9 % 7      # 5  
9% -7      # -5  
- 9 % - 7   # -2
```

如果你需要计算整数除法和取模的结果，可以使用divmod函数作为快捷方式：

```
商, 余数 = divmod(9, 4)  
# 商 = 2, 余数 = 1, 因为 4 * 2 + 1 == 9
```

```
operator.mod(10 , 2)      # 0  
operator.mod(6 , 4)      # 2
```

You can also use negative numbers.

```
-9 % 7      # 5  
9 % -7      # -5  
-9 % -7     # -2
```

If you need to find the result of integer division and modulus, you can use the `divmod` function as a shortcut:

```
quotient, remainder = divmod(9, 4)  
# quotient = 2, remainder = 1 as 4 * 2 + 1 == 9
```

# 视频：Python 数据 科学与机器 学习训练营

学习如何使用 NumPy、Pandas、Seaborn、  
Matplotlib、Plotly、Scikit-Learn、机器学习、  
Tensorflow 等！



- ✓ 使用 Python 进行数据科学和机器学习
- ✓ 使用 Spark 进行大数据分析
- ✓ 实现机器学习算法
- ✓ 学习使用 NumPy 处理数值数据
- ✓ 学习使用 Pandas 进行数据分析
- ✓ 学习使用 Matplotlib 进行 Python 绘图
- ✓ 学习使用 Seaborn 进行统计图表绘制
- ✓ 使用 Plotly 进行交互式动态可视化
- ✓ 使用 SciKit-Learn 进行机器学习任务
- ✓ K-均值聚类
- ✓ 逻辑回归
- ✓ 线性回归
- ✓ 随机森林和决策树
- ✓ 神经网络
- ✓ 支持向量机今天观看→

# VIDEO: Python for Data Science and Machine Learning Bootcamp

Learn how to use NumPy, Pandas, Seaborn,  
Matplotlib , Plotly, Scikit-Learn , Machine Learning,  
Tensorflow, and more!



- ✓ Use Python for Data Science and Machine Learning
- ✓ Use Spark for Big Data Analysis
- ✓ Implement Machine Learning Algorithms
- ✓ Learn to use NumPy for Numerical Data
- ✓ Learn to use Pandas for Data Analysis
- ✓ Learn to use Matplotlib for Python Plotting
- ✓ Learn to use Seaborn for statistical plots
- ✓ Use Plotly for interactive dynamic visualizations
- ✓ Use SciKit-Learn for Machine Learning Tasks
- ✓ K-Means Clustering
- ✓ Logistic Regression
- ✓ Linear Regression
- ✓ Random Forest and Decision Trees
- ✓ Neural Networks
- ✓ Support Vector Machines

**Watch Today →**

# 第10章：按位运算符

按位操作在位级别上改变二进制字符串。这些操作非常基础，并且由处理器直接支持。这些少数操作在处理设备驱动程序、低级图形、密码学和网络通信时是必需的。本节提供了有关Python按位运算符的有用知识和示例。

## 第10.1节：按位非

~ 运算符将翻转数字中的所有位。由于计算机使用有符号数表示法——尤其是二进制补码表示法来编码负二进制数，其中负数以1开头而非0开头。

这意味着如果你使用8位来表示二进制补码数，你会将模式从0000 0000到0111 1111视为表示0到127的数字，并保留1xxx xxxx来表示负数。

八位二进制补码数

位	无符号值	二进制补码值
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
1111 1110	254	-2
1111 1111	255	-1

本质上，这意味着虽然1010 0110的无符号值是166（通过相加( $128 * 1$ ) + ( $64 * 0$ ) + ( $32 * 1$ ) + ( $16 * 0$ ) + ( $8 * 0$ ) + ( $4 * 1$ ) + ( $2 * 1$ ) + ( $1 * 0$ )）得到，但它的二进制补码值是-90（通过相加( $128 * 1$ ) - ( $64 * 0$ ) - ( $32 * 1$ ) - ( $16 * 0$ ) - ( $8 * 0$ ) - ( $4 * 1$ ) - ( $2 * 1$ ) - ( $1 * 0$ ），并对该值取补码得到）。

通过这种方式，负数的范围可以达到-128（1000 0000）。零（0）表示为0000 0000，负一（-1）表示为1111 1111。

一般来说，这意味着 $\sim n = -n - 1$ 。

```
# 0 = 0b0000 0000
~0
# 输出: -1
# -1 = 0b1111 1111

# 1 = 0b0000 0001
~1
# 输出: -2
# -2 = 1111 1110

# 2 = 0b0000 0010
~2
```

# Chapter 10: Bitwise Operators

Bitwise operations alter binary strings at the bit level. These operations are incredibly basic and are directly supported by the processor. These few operations are necessary in working with device drivers, low-level graphics, cryptography, and network communications. This section provides useful knowledge and examples of Python's bitwise operators.

## Section 10.1: Bitwise NOT

The ~ operator will flip all of the bits in the number. Since computers use [signed number representations](#) — most notably, the [two's complement notation](#) to encode negative binary numbers where negative numbers are written with a leading one (1) instead of a leading zero (0).

This means that if you were using 8 bits to represent your two's-complement numbers, you would treat patterns from **0000 0000** to **0111 1111** to represent numbers from 0 to 127 and reserve **1xxx xxxx** to represent negative numbers.

Eight-bit two's-complement numbers

Bits	Unsigned Value	Two's-complement Value
0000 0000 0	0	0
0000 0001 1	1	1
0000 0010 2	2	2
0111 1110 126	126	126
0111 1111 127	127	127
1000 0000 128	-128	-128
1000 0001 129	-127	-127
1000 0010 130	-126	-126
1111 1110 254	-2	-2
1111 1111 255	-1	-1

In essence, this means that whereas **1010 0110** has an unsigned value of 166 (arrived at by adding ( $128 * 1$ ) + ( $64 * 0$ ) + ( $32 * 1$ ) + ( $16 * 0$ ) + ( $8 * 0$ ) + ( $4 * 1$ ) + ( $2 * 1$ ) + ( $1 * 0$ )), it has a two's-complement value of -90 (arrived at by adding ( $128 * 1$ ) - ( $64 * 0$ ) - ( $32 * 1$ ) - ( $16 * 0$ ) - ( $8 * 0$ ) - ( $4 * 1$ ) - ( $2 * 1$ ) - ( $1 * 0$ ), and complementing the value).

In this way, negative numbers range down to -128 (**1000 0000**). Zero (0) is represented as **0000 0000**, and minus one (-1) as **1111 1111**.

In general, though, this means  $\sim n = -n - 1$ .

```
# 0 = 0b0000 0000
~0
# Out: -1
# -1 = 0b1111 1111

# 1 = 0b0000 0001
~1
# Out: -2
# -2 = 1111 1110

# 2 = 0b0000 0010
~2
```

```
# 输出: -3  
# -3 = 0b1111 1101  
  
# 123 = 0b0111 1011  
~123  
# 输出: -124  
# -124 = 0b1000 0100
```

注意，当该操作应用于正数时，其整体效果可以总结为：

$\sim n \rightarrow -|n+1|$

然后，当应用于负数时，相应的效果是：

$\sim n \rightarrow |n-1|$

以下示例说明了这个最后的规则...

```
# -0 = 0b0000 0000  
~-0  
# 输出: -1  
# -1 = 0b1111 1111  
# 0 是这个规则的明显例外，因为 -0 == 0 总是成立  
  
# -1 = 0b1000 0001  
~-1  
# 输出: 0  
# 0 = 0b0000 0000  
  
# -2 = 0b1111 1110  
~-2  
# 输出: 1  
# 1 = 0b0000 0001  
  
# -123 = 0b1111 1011  
~-123  
# 输出: 122  
# 122 = 0b0111 1010
```

## 第10.2节：按位异或 (XOR)

`^`运算符将执行二进制XOR操作，只有当且仅当恰好有一个操作数的值为二进制1时，结果位才为1。换句话说，结果为1仅当操作数不同。示例包括：

```
# 0 ^ 0 = 0  
# 0 ^ 1 = 1  
# 1 ^ 0 = 1  
# 1 ^ 1 = 0  
  
# 60 = 0b111100  
# 30 = 0b011110  
60^30  
# 输出: 34  
# 34 = 0b100010  
  
bin(60 ^ 30)
```

```
# Out: -3  
# -3 = 0b1111 1101  
  
# 123 = 0b0111 1011  
~123  
# Out: -124  
# -124 = 0b1000 0100
```

Note, the overall effect of this operation when applied to positive numbers can be summarized:

$\sim n \rightarrow -|n+1|$

And then, when applied to negative numbers, the corresponding effect is:

$\sim n \rightarrow |n-1|$

The following examples illustrate this last rule...

```
# -0 = 0b0000 0000  
~-0  
# Out: -1  
# -1 = 0b1111 1111  
# 0 is the obvious exception to this rule, as -0 == 0 always  
  
# -1 = 0b1000 0001  
~-1  
# Out: 0  
# 0 = 0b0000 0000  
  
# -2 = 0b1111 1110  
~-2  
# Out: 1  
# 1 = 0b0000 0001  
  
# -123 = 0b1111 1011  
~-123  
# Out: 122  
# 122 = 0b0111 1010
```

## Section 10.2: Bitwise XOR (Exclusive OR)

The `^` operator will perform a binary **XOR** in which a binary 1 is copied if and only if it is the value of exactly **one** operand. Another way of stating this is that the result is 1 only if the operands are different. Examples include:

```
# 0 ^ 0 = 0  
# 0 ^ 1 = 1  
# 1 ^ 0 = 1  
# 1 ^ 1 = 0  
  
# 60 = 0b111100  
# 30 = 0b011110  
60 ^ 30  
# Out: 34  
# 34 = 0b100010  
  
bin(60 ^ 30)
```

```
# 输出: 0b100010
```

## 第10.3节：按位与

&运算符将执行二进制AND操作，只有当位在两个操作数中都存在时，该位才被复制。也就是说：

```
# 0 & 0 = 0  
# 0 & 1 = 0  
# 1 & 0 = 0  
# 1 & 1 = 1
```

```
# 60 = 0b111100  
# 30 = 0b011110  
60& 30  
# 输出: 28  
# 28 = 0b11100
```

```
bin(60 & 30)  
# 输出: 0b11100
```

```
# Out: 0b100010
```

## Section 10.3: Bitwise AND

The & operator will perform a binary **AND**, where a bit is copied if it exists in **both** operands. That means:

```
# 0 & 0 = 0  
# 0 & 1 = 0  
# 1 & 0 = 0  
# 1 & 1 = 1
```

```
# 60 = 0b111100  
# 30 = 0b011110  
60 & 30  
# Out: 28  
# 28 = 0b11100
```

```
bin(60 & 30)  
# Out: 0b11100
```

## 第10.4节：按位或

|运算符将执行二进制“或”操作，只要任一操作数的对应位为1，该位就为1。也就是说：

```
# 0 | 0 = 0  
# 0 | 1 = 1  
# 1 | 0 = 1  
# 1 | 1 = 1
```

```
# 60 = 0b111100  
# 30 = 0b011110  
60| 30  
# 输出: 62  
# 62 = 0b111110
```

```
bin(60 | 30)  
# 输出: 0b111110
```

```
# 0 | 0 = 0  
# 0 | 1 = 1  
# 1 | 0 = 1  
# 1 | 1 = 1
```

```
# 60 = 0b111100  
# 30 = 0b011110  
60 | 30  
# Out: 62  
# 62 = 0b111110
```

```
bin(60 | 30)  
# Out: 0b111110
```

## 第10.5节：按位左移

<< 运算符将执行按位“左移”，即左操作数的值向左移动由右操作数指定的位数。

```
# 2 = 0b10  
2 << 2  
# 输出: 8  
# 8 = 0b1000
```

```
bin(2 << 2)  
# 输出: 0b1000
```

执行左移位 1 等同于乘以 2：

```
7 << 1  
# 输出: 14
```

对 n 进行左移位操作等同于乘以 2 的 n 次方：

## Section 10.5: Bitwise Left Shift

The << operator will perform a bitwise "left shift," where the left operand's value is moved left by the number of bits given by the right operand.

```
# 2 = 0b10  
2 << 2  
# Out: 8  
# 8 = 0b1000
```

```
bin(2 << 2)  
# Out: 0b1000
```

Performing a left bit shift of 1 is equivalent to multiplication by 2:

```
7 << 1  
# Out: 14
```

Performing a left bit shift of n is equivalent to multiplication by  $2^{**n}$ :

```
3 << 4  
# 输出: 48
```

## 第10.6节：按位右移

>> 运算符将执行按位“右移”，即将左操作数的值向右移动由右操作数给定的位数。

```
# 8 = 0b1000  
8 >> 2  
# 输出: 2  
# 2 = 0b10
```

```
bin(8 >> 2)  
# 输出: 0b10
```

对1执行右移位操作等同于整数除以2：

```
36 >> 1  
# 输出: 18
```

```
15 >> 1  
# 输出: 7
```

对n执行右移位操作等同于整数除以2的n次方：

```
48 >> 4  
# 输出: 3
```

```
59 >> 3  
# 输出: 7
```

## 第10.7节：就地操作

所有按位运算符（除~外）都有各自的就地版本

```
a = 0b001  
a &= 0b010  
# a = 0b000
```

```
a = 0b001  
a |= 0b010  
# a = 0b011
```

```
a = 0b001  
a <<= 2  
# a = 0b100
```

```
a = 0b100  
a >>= 2  
# a = 0b001
```

```
a = 0b101  
a ^= 0b011  
# a = 0b110
```

```
3 << 4  
# Out: 48
```

## Section 10.6: Bitwise Right Shift

The `>>` operator will perform a bitwise "right shift," where the left operand's value is moved right by the number of bits given by the right operand.

```
# 8 = 0b1000  
8 >> 2  
# Out: 2  
# 2 = 0b10
```

```
bin(8 >> 2)  
# Out: 0b10
```

Performing a right bit shift of 1 is equivalent to integer division by 2:

```
36 >> 1  
# Out: 18
```

```
15 >> 1  
# Out: 7
```

Performing a right bit shift of n is equivalent to integer division by  $2^{**n}$ :

```
48 >> 4  
# Out: 3
```

```
59 >> 3  
# Out: 7
```

## Section 10.7: Inplace Operations

All of the Bitwise operators (except `~`) have their own in place versions

```
a = 0b001  
a &= 0b010  
# a = 0b000
```

```
a = 0b001  
a |= 0b010  
# a = 0b011
```

```
a = 0b001  
a <<= 2  
# a = 0b100
```

```
a = 0b100  
a >>= 2  
# a = 0b001
```

```
a = 0b101  
a ^= 0b011  
# a = 0b110
```

# 第11章：布尔运算符

## 第11.1节：`and` 和 `or` 不保证返回布尔值

当你使用 `or` 时，如果表达式中的第一个值为真，它将返回该值，否则它会盲目地返回第二个值。也就是说，`or` 等价于：

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

对于 `and`，如果第一个值为假，它将返回该值，否则返回最后一个值：

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

## 第11.2节：一个简单的例子

在 Python 中，你可以使用两个二元运算符来比较单个元素——一个在元素的左侧，另一个在右侧：

```
如果 3.14 < x < 3.142:
    print("x 接近圆周率")
```

在许多（大多数？）编程语言中，这将以与常规数学相反的方式进行计算：`(3.14 < x) < 3.142`，但在 Python 中，它被视为 `3.14 < x 且 x < 3.142`，就像大多数非程序员所期望的那样。

## 第11.3节：短路求值

Python [最小化求值](#) 布尔表达式。

```
>>> 定义 true_func():
...     打印("true_func()")
...     返回 True
...
>>> 定义 false_func():
...     打印("false_func()")
...     返回 False
...
>>> true_func() or false_func()
true_func()
True
>>> false_func() or true_func()
false_func()
true_func()
True
>>> true_func() and false_func()
true_func()
false_func()
False
>>> false_func() and false_func()
```

# Chapter 11: Boolean Operators

## Section 11.1: `and` and `or` are not guaranteed to return a boolean

When you use `or`, it will either return the first value in the expression if it's true, else it will blindly return the second value. I.e. `or` is equivalent to:

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

For `and`, it will return its first value if it's false, else it returns the last value:

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

## Section 11.2: A simple example

In Python you can compare a single element using two binary operators--one on either side:

```
if 3.14 < x < 3.142:
    print("x is near pi")
```

In many (most?) programming languages, this would be evaluated in a way contrary to regular math: `(3.14 < x) < 3.142`, but in Python it is treated like `3.14 < x and x < 3.142`, just like most non-programmers would expect.

## Section 11.3: Short-circuit evaluation

Python [minimally evaluates](#) Boolean expressions.

```
>>> def true_func():
...     打印("true_func()")
...     返回 True
...
>>> def false_func():
...     打印("false_func()")
...     返回 False
...
>>> true_func() or false_func()
true_func()
True
>>> false_func() or true_func()
false_func()
true_func()
True
>>> true_func() and false_func()
true_func()
false_func()
False
>>> false_func() and false_func()
```

```
false_func()  
False
```

## 第11.4节：and

仅当两个参数都为真值时，求值结果为第二个参数。否则，求值结果为第一个假值参数。

```
x = True  
y = True  
z = x 和 y # z = True  
  
x = True  
y = False  
z = x 和 y # z = False  
  
x = False  
y = True  
z = x 和 y # z = False  
  
x = False  
y = False  
z = x 和 y # z = False  
  
x = 1  
y = 1  
z = x 和 y # z = y, 所以 z = 1, 参见 `and` 和 `or` 不保证返回布尔值  
  
x = 0  
y = 1  
z = x 和 y # z = x, 所以 z = 0 (见上文)  
  
x = 1  
y = 0  
z = x 和 y # z = y, 所以 z = 0 (见上文)  
  
x = 0  
y = 0  
z = x 和 y # z = x, 所以 z = 0 (见上文)
```

上面例子中的1可以更改为任何真值，0可以更改为任何假值。

## 第11.5节：或

如果任一参数为真，则返回第一个真值参数。如果两个参数都为假，则返回第二个参数。

```
x = True  
y = True  
z = x 或 y # z = True  
  
x = True  
y = False  
z = x 或 y # z = True  
  
x = False  
y = True  
z = x or y # z = True
```

```
false_func()  
False
```

## Section 11.4: and

Evaluates to the second argument if and only if both of the arguments are truthy. Otherwise evaluates to the first falsy argument.

```
x = True  
y = True  
z = x and y # z = True  
  
x = True  
y = False  
z = x and y # z = False  
  
x = False  
y = True  
z = x and y # z = False  
  
x = False  
y = False  
z = x and y # z = False  
  
x = 1  
y = 1  
z = x and y # z = y, so z = 1, see `and` and `or` are not guaranteed to be a boolean  
  
x = 0  
y = 1  
z = x and y # z = x, so z = 0 (see above)  
  
x = 1  
y = 0  
z = x and y # z = y, so z = 0 (see above)  
  
x = 0  
y = 0  
z = x and y # z = x, so z = 0 (see above)
```

The 1's in the above example can be changed to any truthy value, and the 0's can be changed to any falsy value.

## Section 11.5: or

Evaluates to the first truthy argument if either one of the arguments is truthy. If both arguments are falsey, evaluates to the second argument.

```
x = True  
y = True  
z = x or y # z = True  
  
x = True  
y = False  
z = x or y # z = True  
  
x = False  
y = True  
z = x or y # z = True
```

```
x = False  
y = False  
z = x or y # z = False
```

```
x = 1  
y = 1  
z = x or y # z = x, 所以 z = 1, 见`and`和`or`不保证返回布尔值
```

```
x = 1  
y = 0  
z = x or y # z = x, 所以 z = 1 (见上文)
```

```
x = 0  
y = 1  
z = x or y # z = y, 所以 z = 1 (见上文)
```

```
x = 0  
y = 0  
z = x or y # z = y, 所以 z = 0 (见上文)
```

上面例子中的1可以更改为任何真值，0可以更改为任何假值。

## 第11.6节：not

它返回以下语句的相反值：

```
x = True  
y = not x # y = False
```

```
x = False  
y = not x # y = True
```

```
x = False  
y = False  
z = x or y # z = False
```

```
x = 1  
y = 1  
z = x or y # z = x, so z = 1, see `and` and `or` are not guaranteed to be a boolean
```

```
x = 1  
y = 0  
z = x or y # z = x, so z = 1 (see above)
```

```
x = 0  
y = 1  
z = x or y # z = y, so z = 1 (see above)
```

```
x = 0  
y = 0  
z = x or y # z = y, so z = 0 (see above)
```

The 1's in the above example can be changed to any truthy value, and the 0's can be changed to any falsey value.

## Section 11.6: not

It returns the opposite of the following statement:

```
x = True  
y = not x # y = False
```

```
x = False  
y = not x # y = True
```

# 第12章：运算符优先级

Python 运算符有一套优先级顺序，用于确定在可能存在歧义的表达式中哪个运算符先被计算。例如，在表达式 `3 * 2 + 7` 中，先计算 `3` 乘以 `2`，然后将结果加上 `7`，得到 `13`。表达式不会反过来计算，因为 `*` 的优先级高于 `+`。

下面是按优先级排列的运算符列表，以及它们（通常）执行的简要说明。

## 第12.1节：Python 中简单的运算符优先级示例

Python 遵循 PEMDAS 规则。PEMDAS 代表括号（Parentheses）、指数（Exponents）、乘法和除法（Multiplication and Division）、加法和减法（Addition and Subtraction）。

示例：

```
>>> a, b, c, d = 2, 3, 5, 7
>>> a ** (b + c) # 括号
256
>>> a * b ** c # 指数运算：等同于 `a * (b ** c)`
776
>>> a + b * c / d # 乘除运算：等同于 `a + (b * c / d)`
4.142857142857142
```

附加说明：数学规则依然适用，但不总是：

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

# Chapter 12: Operator Precedence

Python operators have a set **order of precedence**, which determines what operators are evaluated first in a potentially ambiguous expression. For instance, in the expression `3 * 2 + 7`, first `3` is multiplied by `2`, and then the result is added to `7`, yielding `13`. The expression is not evaluated the other way around, because `*` has a higher precedence than `+`.

Below is a list of operators by precedence, and a brief description of what they (usually) do.

## Section 12.1: Simple Operator Precedence Examples in python

Python follows PEMDAS rule. PEMDAS stands for Parentheses, Exponents, Multiplication and Division, and Addition and Subtraction.

Example:

```
>>> a, b, c, d = 2, 3, 5, 7
>>> a ** (b + c) # parentheses
256
>>> a * b ** c # exponent: same as `a * (b ** c)`
776
>>> a + b * c / d # multiplication / division: same as `a + (b * c / d)`
4.142857142857142
```

Extras: mathematical rules hold, but [not always](#):

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

# 第13章：变量作用域与绑定

## 第13.1节：非局部变量

Python 3.x 版本 ≥ 3.0

Python 3新增了一个名为nonlocal的关键字。nonlocal关键字为内部作用域添加了作用域覆盖。

你可以在[PEP 3104](#)中详细了解它。通过几个代码示例可以更好地说明这一点。其中最常见的示例之一是创建一个可以递增的函数：

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

如果你尝试运行这段代码，你会收到一个UnboundLocalError，因为num变量在最内层函数中被引用时还未被赋值。让我们加入nonlocal来解决这个问题：

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

基本上，nonlocal允许你给外层作用域中的变量赋值，但不能赋值给全局作用域。所以你不能在我们的counter函数中使用onlocal，因为那样会尝试赋值给全局作用域。试试看，你会很快遇到一个SyntaxError。相反，你必须在嵌套函数中使用nonlocal。

(注意，这里展示的功能用生成器实现会更好。)

## 第13.2节：全局变量

在Python中，函数内部的变量只有在出现在赋值语句的左侧，或其他绑定出现时，才被视为局部变量；否则，这种绑定会在外层函数中查找，直到全局作用域。即使赋值语句从未执行，这条规则依然成立。

```
x = 'Hi'

def read_x():
    print(x) # x 只是被引用，因此假定为全局变量

read_x() # 打印 Hi

def read_y():
    print(y) # 这里 y 只是被引用，因此假定为全局变量
```

# Chapter 13: Variable Scope and Binding

## Section 13.1: Nonlocal Variables

Python 3.x Version ≥ 3.0

Python 3 added a new keyword called **nonlocal**. The nonlocal keyword adds a scope override to the inner scope.

You can read all about it in [PEP 3104](#). This is best illustrated with a couple of code examples. One of the most common examples is to create function that can increment:

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

If you try running this code, you will receive an **UnboundLocalError** because the **num** variable is referenced before it is assigned in the innermost function. Let's add nonlocal to the mix:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

Basically **nonlocal** will allow you to assign to variables in an outer scope, but not a global scope. So you can't use **nonlocal** in our counter function because then it would try to assign to a global scope. Give it a try and you will quickly get a **SyntaxError**. Instead you must use **nonlocal** in a nested function.

(Note that the functionality presented here is better implemented using generators.)

## Section 13.2: Global Variables

In Python, variables inside functions are considered local if and only if they appear in the left side of an assignment statement, or some other binding occurrence; otherwise such a binding is looked up in enclosing functions, up to the global scope. This is true even if the assignment statement is never executed.

```
x = 'Hi'

def read_x():
    print(x) # x is just referenced, therefore assumed global

read_x() # prints Hi

def read_y():
    print(y) # here y is just referenced, therefore assumed global
```

```

read_y()      # NameError: 全局名称 'y' 未定义

def read_y():
    y = 'Hey'  # y 出现在赋值中, 因此它是局部变量
    print(y)   # 会找到局部变量 y

read_y()      # 打印 Hey

def read_x_local_fail():
    if False:
        x = 'Hey'  # x 出现在赋值中, 因此它是局部变量
        print(x)   # 会查找局部的 x, 但未赋值, 因此找不到

read_x_local_fail()  # UnboundLocalError: 局部变量 'x' 在赋值前被引用

```

通常, 作用域内的赋值会遮蔽任何同名的外部变量:

```

x = 'Hi'

def change_local_x():
    x = 'Bye'
    print(x)
change_local_x()  # 输出 Bye
print(x)  # 输出 Hi

```

声明一个名字为**global**意味着, 在其余的作用域中, 对该名字的任何赋值都将在模块的顶层进行:

```

x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x()  # 输出 Bye
print(x)  # 输出 Bye

```

关键字**global**意味着赋值将在模块的顶层进行, 而不是程序的顶层。其他模块仍然需要通过通常的点号访问来访问模块内的变量。

总结: 为了判断变量 `x`是否是函数的局部变量, 你应该阅读整个函数:

1. 如果你找到了全局`x`, 那么`x`是一个全局变量
2. 如果你发现了`nonlocal x`, 那么 `x` 属于一个外层函数, 既不是局部变量也不是全局变量
3. 如果你发现了 `x = 5` 或者 `for x in range(3)` 或其他绑定, 那么 `x` 是一个局部变量
4. 否则 `x` 属于某个外层作用域 (函数作用域、全局作用域或内置作用域)

## 第13.3节 : 局部变量

如果一个名字在函数内部被绑定, 默认情况下它只在函数内部可访问:

```

def foo():
    a = 5
    print(a) # ok

print(a) # NameError: name 'a' is not defined

```

```

read_y()      # NameError: global name 'y' is not defined

def read_y():
    y = 'Hey'  # y appears in an assignment, therefore it's local
    print(y)   # will find the local y

read_y()      # prints Hey

def read_x_local_fail():
    if False:
        x = 'Hey'  # x appears in an assignment, therefore it's local
        print(x)   # will look for the _local_ z, which is not assigned, and will not be found

read_x_local_fail()  # UnboundLocalError: local variable 'x' referenced before assignment

```

Normally, an assignment inside a scope will shadow any outer variables of the same name:

```

x = 'Hi'

def change_local_x():
    x = 'Bye'
    print(x)
change_local_x()  # prints Bye
print(x)  # prints Hi

```

Declaring a name **global** means that, for the rest of the scope, any assignments to the name will happen at the module's top level:

```

x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x()  # prints Bye
print(x)  # prints Bye

```

The **global** keyword means that assignments will happen at the module's top level, not at the program's top level. Other modules will still need the usual dotted access to variables within the module.

To summarize: in order to know whether a variable `x` is local to a function, you should read the *entire* function:

1. if you've found `global x`, then `x` is a **global** variable
2. If you've found `nonlocal x`, then `x` belongs to an enclosing function, and is neither local nor global
3. If you've found `x = 5` or `for x in range(3)` or some other binding, then `x` is a **local** variable
4. Otherwise `x` belongs to some enclosing scope (function scope, global scope, or builtins)

## Section 13.3: Local Variables

If a name is *bound* inside a function, it is by default accessible only within the function:

```

def foo():
    a = 5
    print(a) # ok

print(a) # NameError: name 'a' is not defined

```

控制流结构对作用域没有影响（except 除外），但访问尚未赋值的变量会导致错误：

```
def foo():
    if True:
        a = 5
        print(a) # ok

b = 3
def bar():
    if False:
        b = 5
print(b) # UnboundLocalError: local variable 'b' referenced before assignment
```

常见的绑定操作有赋值、for 循环，以及类似 a += 5 的增强赋值

## 第13.4节：del命令

此命令有几种相关但不同的形式。

**del v**

如果 v 是一个变量，命令 del v 会将该变量从其作用域中移除。例如：

```
x = 5
print(x) # 输出: 5
del x
print(x) # NameError: name 'f' is not defined
```

注意 del 是一个 *binding occurrence*（绑定出现），这意味着除非明确使用 **nonlocal** 或 **global**，否则 del v 会使 v 在当前作用域内成为局部变量。如果你打算删除外层作用域中的 v，请在与 del v 语句相同的作用域中使用 **nonlocal v** 或 **global v**。

在以下所有情况下，命令的意图是默认行为，但语言并不强制执行。一个类可能以使该意图无效的方式编写。

**del v.name**

该命令会触发对 v.\_\_delattr\_\_(name) 的调用。

其意图是使属性 name 不可用。例如：

```
class A:
    pass

a = A()
a.x = 7
print(a.x) # 输出: 7
del a.x
print(a.x) # 错误: AttributeError: 'A' 对象没有属性 'x'
del v[item]
```

此命令触发对 v.\_\_delitem\_\_(item) 的调用。

其意图是 item 不属于对象 v 实现的映射。例如：

Control flow constructs have no impact on the scope (with the exception of **except**), but accessing variable that was not assigned yet is an error:

```
def foo():
    if True:
        a = 5
        print(a) # ok

b = 3
def bar():
    if False:
        b = 5
print(b) # UnboundLocalError: local variable 'b' referenced before assignment
```

Common binding operations are assignments, **for** loops, and augmented assignments such as a += 5

## Section 13.4: The del command

This command has several related yet distinct forms.

**del v**

If v is a variable, the command **del v** removes the variable from its scope. For example:

```
x = 5
print(x) # out: 5
del x
print(x) # NameError: name 'f' is not defined
```

Note that **del** is a *binding occurrence*, which means that unless explicitly stated otherwise (using **nonlocal** or **global**), **del v** will make v local to the current scope. If you intend to delete v in an outer scope, use **nonlocal v** or **global v** in the same scope of the **del v** statement.

In all the following, the intention of a command is a default behavior but is not enforced by the language. A class might be written in a way that invalidates this intention.

**del v.name**

This command triggers a call to v.\_\_delattr\_\_(name).

The intention is to make the attribute name unavailable. For example:

```
class A:
    pass

a = A()
a.x = 7
print(a.x) # out: 7
del a.x
print(a.x) # error: AttributeError: 'A' object has no attribute 'x'
del v[item]
```

This command triggers a call to v.\_\_delitem\_\_(item).

The intention is that item will not belong in the mapping implemented by the object v. For example:

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) # 输出: {'b': 2}
print(x['a']) # 错误: KeyError: 'a'
del v[a:b]
```

这实际上调用了v.\_\_delslice\_\_(a, b)。

意图与上述描述的类似，但作用于切片——一系列项目的范围，而不是单个项目。例如：

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) # 输出: [0, 3, 4]
```

另见垃圾回收#del命令。

## 第13.5节：函数在查找名称时跳过类作用域

类在定义时有局部作用域，但类内部的函数在查找名称时不使用该作用域。因为lambda是函数，且推导式是使用函数作用域实现的，这可能导致一些令人惊讶的行为。

```
a = 'global'

class Fred:
    a = 'class' # 类作用域
    b = (a for i in range(10)) # 函数作用域
    c = [a for i in range(10)] # 函数作用域
    d = a # 类作用域
    e = lambda: a # 函数作用域
    f = lambda a=a: a # 默认参数使用类作用域

    @staticmethod # 或 @classmethod, 或普通实例方法
    def g(): # 函数作用域
        return a

print(Fred.a) # 类
print(next(Fred.b)) # 全局
print(Fred.c[0]) # Python 2 中为类, Python 3 中为全局
print(Fred.d) # 类
print(Fred.e()) # 全局
print(Fred.f()) # 类
print(Fred.g()) # 全局
```

不熟悉此作用域工作方式的用户可能会期望 b、c 和 e 打印 class。

摘自 [PEP 227](#)：

类作用域中的名称不可访问。名称在最内层的封闭函数作用域中解析。

如果类定义出现在嵌套作用域链中，解析过程会跳过类定义。

摘自Python文档中关于命名和绑定的内容：

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) # out: {'b': 2}
print(x['a']) # error: KeyError: 'a'
del v[a:b]
```

This actually calls v.\_\_delslice\_\_(a, b).

The intention is similar to the one described above, but with slices - ranges of items instead of a single item. For example:

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) # out: [0, 3, 4]
```

See also Garbage Collection#The del command.

## Section 13.5: Functions skip class scope when looking up names

Classes have a local scope during definition, but functions inside the class do not use that scope when looking up names. Because lambdas are functions, and comprehensions are implemented using function scope, this can lead to some surprising behavior.

```
a = 'global'

class Fred:
    a = 'class' # class scope
    b = (a for i in range(10)) # function scope
    c = [a for i in range(10)] # function scope
    d = a # class scope
    e = lambda: a # function scope
    f = lambda a=a: a # default argument uses class scope

    @staticmethod # or @classmethod, or regular instance method
    def g(): # function scope
        return a

print(Fred.a) # class
print(next(Fred.b)) # global
print(Fred.c[0]) # class in Python 2, global in Python 3
print(Fred.d) # class
print(Fred.e()) # global
print(Fred.f()) # class
print(Fred.g()) # global
```

Users unfamiliar with how this scope works might expect b, c, and e to print class.

From [PEP 227](#):

Names in class scope are not accessible. Names are resolved in the innermost enclosing function scope. If a class definition occurs in a chain of nested scopes, the resolution process skips class definitions.

From Python's documentation on [naming and binding](#):

类代码块中定义的名称范围仅限于该类代码块；它不会扩展到方法的代码块——这包括推导式和生成器表达式，因为它们是通过函数作用域实现的。这意味着以下代码将会失败：

```
class A:  
    a = 42  
    b = list(a + i for i in range(10))
```

此示例引用了 Martijn Pieters 在[this answer](#)中的内容，该内容对这种行为进行了更深入的分析。

## 第13.6节：局部作用域与全局作用域

### 什么是局部作用域和全局作用域？

所有在代码中某处可访问的 Python 变量要么处于局部作用域，要么处于全局作用域。

解释是，局部作用域包括当前函数中定义的所有变量，全局作用域包括当前函数外定义的变量。

```
foo = 1 # 全局  
  
def func():  
    bar = 2 # 局部  
    print(foo) # 打印全局作用域中的变量 foo  
    print(bar) # 打印局部作用域中的变量 bar
```

可以检查哪些变量在哪些作用域中。内置函数locals()和globals()返回整个作用域作为字典。

```
foo = 1  
  
def func():  
    bar = 2  
    print(globals().keys()) # 打印全局作用域中所有变量名  
    print(locals().keys()) # 打印局部作用域中所有变量名
```

### 名称冲突时会发生什么？

```
foo = 1  
  
def func():  
    foo = 2 # 在局部作用域中创建一个新的变量 foo, 全局的 foo 不受影响  
  
    print(foo) # 打印 2  
  
    # 全局变量 foo 仍然存在, 未改变：  
    print(globals()['foo']) # 打印 1  
    print(locals()['foo']) # 打印 2
```

要修改全局变量，请使用关键字global：

```
foo = 1  
  
def func():  
    global foo  
    foo = 2 # 这会修改全局变量 foo, 而不是创建局部变量
```

The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods – this includes comprehensions and generator expressions since they are implemented using a function scope. This means that the following will fail:

```
class A:  
    a = 42  
    b = list(a + i for i in range(10))
```

This example uses references from [this answer](#) by Martijn Pieters, which contains more in depth analysis of this behavior.

## Section 13.6: Local vs Global Scope

### What are local and global scope?

All Python variables which are accessible at some point in code are either in *local scope* or in *global scope*.

The explanation is that local scope includes all variables defined in the current function and global scope includes variables defined outside of the current function.

```
foo = 1 # global  
  
def func():  
    bar = 2 # local  
    print(foo) # prints variable foo from global scope  
    print(bar) # prints variable bar from local scope
```

One can inspect which variables are in which scope. Built-in functions `locals()` and `globals()` return the whole scopes as dictionaries.

```
foo = 1  
  
def func():  
    bar = 2  
    print(globals().keys()) # prints all variable names in global scope  
    print(locals().keys()) # prints all variable names in local scope
```

### What happens with name clashes?

```
foo = 1  
  
def func():  
    foo = 2 # creates a new variable foo in local scope, global foo is not affected  
  
    print(foo) # prints 2  
  
    # global variable foo still exists, unchanged:  
    print(globals()['foo']) # prints 1  
    print(locals()['foo']) # prints 2
```

To modify a global variable, use keyword `global`:

```
foo = 1  
  
def func():  
    global foo  
    foo = 2 # this modifies the global foo, rather than creating a local variable
```

## 函数体的整个范围都定义了作用域！

这意味着变量不会在函数的一半是全局变量，另一半是局部变量，反之亦然。

```
foo = 1

def func():
    # 这个函数有一个局部变量 foo, 因为它在下面定义了。
    # 所以, 从这一点开始, foo 是局部变量。全局的 foo 被隐藏了。

    print(foo) # 抛出 UnboundLocalError, 因为局部变量 foo 尚未初始化
    foo = 7
    print(foo)
```

同样，反过来也成立：

```
foo = 1

def func():
    # 在此函数中, foo 从一开始就是全局变量

foo = 7 # 修改了全局变量 foo

print(foo) # 7
print(globals()['foo']) # 7

global foo # 这可以出现在函数的任何位置
print(foo) # 7
```

## 函数中的函数

函数中可能嵌套多层函数，但在任何一个函数内部，只有该函数的一个局部作用域和全局作用域。不存在中间作用域。

```
foo = 1

def f1():
bar = 1

    def f2():
baz = 2
        # 这里, foo 是一个全局变量, baz 是一个局部变量
        # bar 不在任何作用域内
        print(locals().keys()) # ['baz']
        print('bar' in locals()) # False
        print('bar' in globals()) # False

    def f3():
baz = 3
        print(bar) # 引用了 f1 中的 bar, 因此进入 f3 的局部作用域 (闭包)
        print(locals().keys()) # ['bar', 'baz']
        print('bar' in locals()) # True
        print('bar' in globals()) # False

    def f4():
bar = 4 # 一个新的局部变量 bar, 隐藏了 f1 局部作用域中的 bar
baz = 4
        print(bar)
        print(locals().keys()) # ['bar', 'baz']
        print('bar' in locals()) # True
```

## The scope is defined for the whole body of the function!

What it means is that a variable will never be global for a half of the function and local afterwards, or vice-versa.

```
foo = 1

def func():
    # This function has a local variable foo, because it is defined down below.
    # So, foo is local from this point. Global foo is hidden.

    print(foo) # raises UnboundLocalError, because local foo is not yet initialized
    foo = 7
    print(foo)
```

Likewise, the opposite:

```
foo = 1

def func():
    # In this function, foo is a global variable from the beginning

foo = 7 # global foo is modified

print(foo) # 7
print(globals()['foo']) # 7

global foo # this could be anywhere within the function
print(foo) # 7
```

## Functions within functions

There may be many levels of functions nested within functions, but within any one function there is only one local scope for that function and the global scope. There are no intermediate scopes.

```
foo = 1

def f1():
bar = 1

    def f2():
baz = 2
        # here, foo is a global variable, baz is a local variable
        # bar is not in either scope
        print(locals().keys()) # ['baz']
        print('bar' in locals()) # False
        print('bar' in globals()) # False

    def f3():
baz = 3
        print(bar) # bar from f1 is referenced so it enters local scope of f3 (closure)
        print(locals().keys()) # ['bar', 'baz']
        print('bar' in locals()) # True
        print('bar' in globals()) # False

    def f4():
bar = 4 # a new local bar which hides bar from local scope of f1
baz = 4
        print(bar)
        print(locals().keys()) # ['bar', 'baz']
        print('bar' in locals()) # True
```

```
print('bar' in globals()) # False
```

### global 与 nonlocal (仅限 Python 3)

这两个关键字都用于获得对当前函数局部之外变量的写访问权限。

global 关键字声明某个名称应被视为全局变量。

```
foo = 0 # 全局 foo

def f1():
    foo = 1 # f1 中的一个新的局部 foo

    def f2():
        def f3():
            foo = 3 # f3 中的一个新的局部 foo
            print(foo) # 3
        foo = 30 # 仅修改 f3 中的局部 foo

        def f4():
            global foo
            print(foo) # 0
    foo = 100 # 修改全局 foo
```

另一方面，nonlocal（参见非局部变量），在Python 3中可用，将一个来自外层作用域的local变量引入当前函数的局部作用域。

[来自Python文档中关于nonlocal的说明：](#)

nonlocal语句使得列出的标识符引用最近的封闭作用域中先前绑定的变量，排除全局变量。

Python 3.x 版本 ≥ 3.0

```
def f1():

    def f2():
        foo = 2 # f2 中的一个新的局部 foo

        def f3():
            nonlocal foo # 来自f2的foo, 是最近的封闭作用域
            print(foo) # 2
        foo = 20 # 修改了f2中的foo !
```

## 第13.7节：绑定出现

```
x = 5
x += 7
for x in iterable: pass
```

上述每条语句都是一个绑定出现——x绑定到对象5。如果该语句出现在函数内部，则默认情况下 x为函数局部变量。有关绑定语句的列表，请参见“语法”部分。

```
print('bar' in globals()) # False
```

### global vs nonlocal (Python 3 only)

Both these keywords are used to gain write access to variables which are not local to the current functions.

The **global** keyword declares that a name should be treated as a global variable.

```
foo = 0 # global foo

def f1():
    foo = 1 # a new foo local in f1

    def f2():
        def f3():
            foo = 3 # a new foo local in f3
            print(foo) # 3
        foo = 30 # modifies local foo in f3 only

    def f4():
        global foo
        print(foo) # 0
    foo = 100 # modifies global foo
```

On the other hand, **nonlocal** (see Nonlocal Variables), available in Python 3, takes a *local* variable from an enclosing scope into the local scope of current function.

[From the Python documentation on nonlocal:](#)

The nonlocal statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals.

Python 3.x Version ≥ 3.0

```
def f1():

    def f2():
        foo = 2 # a new foo local in f2

        def f3():
            nonlocal foo # foo from f2, which is the nearest enclosing scope
            print(foo) # 2
        foo = 20 # modifies foo from f2!
```

## Section 13.7: Binding Occurrence

```
x = 5
x += 7
for x in iterable: pass
```

Each of the above statements is a *binding occurrence* - x become bound to the object denoted by 5. If this statement appears inside a function, then x will be function-local by default. See the "Syntax" section for a list of binding statements.

# 第14章：条件语句

条件表达式，涉及if、elif和else等关键字，使Python程序能够根据布尔条件True或False执行不同的操作。本节介绍Python条件语句、布尔逻辑和三元表达式的使用。

## 第14.1节：条件表达式（或称“三元运算符”）

三元运算符用于内联条件表达式。它最适合用于简单、简洁且易于阅读的操作。

- 参数的顺序与许多其他语言（如C、Ruby、Java等）不同，这可能导致不熟悉Python“令人惊讶”的行为的人使用时出现错误（他们可能会颠倒顺序）。
- 有些人觉得它“难以驾驭”，因为它违背了正常的思维流程（先考虑条件，然后是结果）。

```
n = 5
```

```
"大于2" if n > 2 else "小于或等于2"  
# 输出：'大于2'
```

该表达式的结果将如同用英语阅读一样——如果条件表达式为True，则计算左侧的表达式，否则计算右侧的表达式。

三元运算符也可以嵌套使用，如下所示：

```
n = 5  
"你好" if n > 10 else "再见" if n > 5 else "日安"
```

它们还提供了一种在 lambda 函数中包含条件语句的方法。

## 第14.2节：if、elif 和 else

在Python中，你可以使用if定义第一个条件，使用elif定义后续条件，直到最后一个（可选的）else来处理未被其他条件捕获的情况。

```
number = 5  
  
if number > 2:  
    print("数字大于2。")  
elif number < 2: # 可选子句（你可以有多个elif）  
    print("数字小于2。")  
else: # 可选子句（你只能有一个else）  
    print("数字等于2。")
```

输出 数字 大于 2

使用else if代替elif会触发语法错误，这是不允许的。

## 第14.3节：真值

以下值被视为假值，即在布尔运算中被评估为False。

# Chapter 14: Conditionals

Conditional expressions, involving keywords such as if, elif, and else, provide Python programs with the ability to perform different actions depending on a boolean condition: True or False. This section covers the use of Python conditionals, boolean logic, and ternary statements.

## Section 14.1: Conditional Expression (or "The Ternary Operator")

The ternary operator is used for inline conditional expressions. It is best used in simple, concise operations that are easily read.

- The order of the arguments is different from many other languages (such as C, Ruby, Java, etc.), which may lead to bugs when people unfamiliar with Python's "surprising" behaviour use it (they may reverse the order).
- Some find it "unwieldy", since it goes contrary to the normal flow of thought (thinking of the condition first and then the effects).

```
n = 5
```

```
"Greater than 2" if n > 2 else "Smaller than or equal to 2"  
# Out: 'Greater than 2'
```

The result of this expression will be as it is read in English - if the conditional expression is True, then it will evaluate to the expression on the left side, otherwise, the right side.

Ternary operations can also be nested, as here:

```
n = 5  
"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"
```

They also provide a method of including conditionals in lambda functions.

## Section 14.2: if, elif, and else

In Python you can define a series of conditionals using if for the first one, elif for the rest, up until the final (optional) else for anything not caught by the other conditionals.

```
number = 5  
  
if number > 2:  
    print("Number is bigger than 2.")  
elif number < 2: # Optional clause (you can have multiple elifs)  
    print("Number is smaller than 2.")  
else: # Optional clause (you can only have one else)  
    print("Number is 2.")
```

Outputs Number is bigger than 2

Using else if instead of elif will trigger a syntax error and is not allowed.

## Section 14.3: Truth Values

The following values are considered falsey, in that they evaluate to False when applied to a boolean operator.

- 无
- 假
- 0, 或任何等同于零的数值, 例如`0L`, `0.0`, `0j`
- 空序列: `''`, `" "`, `()`, `[]`
- 空映射: `{}`
- 用户定义的类型, 其`__bool__`或`__len__`方法返回0或`False`

Python中所有其他值的布尔值均为True。

**注意:**一个常见错误是简单地检查返回不同假值的操作的假性

而忽略了差异的重要性。例如, 使用`if foo()`而不是更明确的`if foo() is None`

## 第14.4节：布尔逻辑表达式

布尔逻辑表达式除了计算为True或False外, 还返回被解释为True或False的值。这是Pythonic的表示逻辑的方式, 否则可能需要if-else测试。

### 与运算符

and运算符会计算所有表达式, 如果所有表达式均为True, 则返回最后一个表达式。  
否则它返回第一个计算结果为False的值:

```
>>> 1 和 2
2

>>> 1 和 0
0

>>> 1 和 "Hello World"
"Hello World"

>>> "" 和 "Pancakes"
""
```

### 或运算符

or运算符从左到右计算表达式, 返回第一个计算结果为 True 的值, 或者如果没有 True 的值, 则返回最后一个值。

```
>>> 1 or 2
1

>>> None or 1
1

>>> 0 or []
[]
```

### 惰性求值

当你使用这种方法时, 请记住评估是惰性的。对于确定结果不需要计算的表达式, 不会进行计算。例如:

- `None`
- `False`
- 0, or any numerical value equivalent to zero, for example `0L`, `0.0`, `0j`
- Empty sequences: `''`, `" "`, `()`, `[]`
- Empty mappings: `{}`
- User-defined types where the `__bool__` or `__len__` methods return 0 or `False`

All other values in Python evaluate to `True`.

**Note:** A common mistake is to simply check for the Falseness of an operation which returns different Falsey values where the difference matters. For example, using `if foo()` rather than the more explicit `if foo() is None`

## Section 14.4: Boolean Logic Expressions

Boolean logic expressions, in addition to evaluating to `True` or `False`, return the *value* that was interpreted as `True` or `False`. It is Pythonic way to represent logic that might otherwise require an if-else test.

### And operator

The `and` operator evaluates all expressions and returns the last expression if all expressions evaluate to `True`. Otherwise it returns the first value that evaluates to `False`:

```
>>> 1 and 2
2

>>> 1 and 0
0

>>> 1 and "Hello World"
"Hello World"

>>> "" and "Pancakes"
""
```

### Or operator

The `or` operator evaluates the expressions left to right and returns the first value that evaluates to `True` or the last value (if none are `True`).

```
>>> 1 or 2
1

>>> None or 1
1

>>> 0 or []
[]
```

### Lazy evaluation

When you use this approach, remember that the evaluation is lazy. Expressions that are not required to be evaluated to determine the result are not evaluated. For example:

```
>>> def print_me():
    print('我在这里！')
>>> 0 和 print_me()
0
```

在上述例子中，print\_me 从未被执行，因为Python在遇到 0（即 False）时，可以确定整个表达式为 False。若 print\_me 需要执行以满足程序逻辑，请注意这一点。

## 测试多个条件

检查多个条件时常见的错误是逻辑应用不正确。

这个例子试图检查两个变量是否都大于2。语句被评估为 - if (a) 和 (b > 2)。这个结果出乎意料，因为 bool(a) 当 a 不为零时评估为 True。

```
>>> a = 1
>>> b = 6
>>> 如果 a 和 b > 2:
...     打印('yes')
... 否则:
...     打印('no')

yes
```

每个变量需要单独比较。

```
>>> 如果 a > 2 且 b > 2:
...     打印('yes')
... 否则:
...     打印('no')

no
```

另一个类似的错误是在检查变量是否为多个值之一时发生的。此示例中的语句被评估为 - 如果 (a == 3) or (4) or (6)。由于 bool(4) 和bool(6) 都被评估为 True，导致了意外的结果。

```
>>> a = 1
>>> 如果 a == 3 or 4 or 6:
...     打印('yes')
... 否则:
...     打印('no')

yes
```

同样，每个比较必须单独进行

```
>>> 如果 a == 3 or a == 4 or a == 6:
...     打印('yes')
... 否则:
...     打印('no')

no
```

使用 in 运算符是编写此操作的规范方式。

```
>>> def print_me():
    print('I am here!')
>>> 0 and print_me()
0
```

In the above example, print\_me is never executed because Python can determine the entire expression is False when it encounters the 0 (False). Keep this in mind if print\_me needs to execute to serve your program logic.

## Testing for multiple conditions

A common mistake when checking for multiple conditions is to apply the logic incorrectly.

This example is trying to check if two variables are each greater than 2. The statement is evaluated as - if (a) and (b > 2). This produces an unexpected result because bool(a) evaluates as True when a is not zero.

```
>>> a = 1
>>> b = 6
>>> if a and b > 2:
...     print('yes')
... else:
...     print('no')

yes
```

Each variable needs to be compared separately.

```
>>> if a > 2 and b > 2:
...     print('yes')
... else:
...     print('no')

no
```

Another, similar, mistake is made when checking if a variable is one of multiple values. The statement in this example is evaluated as - if (a == 3) or (4) or (6). This produces an unexpected result because bool(4) and bool(6) each evaluate to True

```
>>> a = 1
>>> if a == 3 or 4 or 6:
...     print('yes')
... else:
...     print('no')

yes
```

Again each comparison must be made separately

```
>>> if a == 3 or a == 4 or a == 6:
...     print('yes')
... else:
...     print('no')

no
```

Using the in operator is the canonical way to write this.

```
>>> if a in (3, 4, 6):
...     打印('yes')
... 否则:
...     打印('no')
no
```

## 第14.5节：使用 cmp 函数获取两个对象的比较结果

Python 2 包含一个 `cmp` 函数，允许你判断一个对象是否小于、等于或大于另一个对象。该函数可用于根据这三种选项之一从列表中选择一个选项。

假设你需要在  $x > y$  时打印 'greater than'， $x < y$  时打印 'less than'， $x == y$  时打印 'equal'。

```
['equal', 'greater than', 'less than', ][cmp(x,y)]
# x,y = 1,1 输出: 'equal'
# x,y = 1,2 输出: 'less than'
# x,y = 2,1 输出: 'greater than'
```

`cmp(x,y)` 返回以下值

### 比较结果

$x < y$	-1
$x == y$	0
$x > y$	1

此函数在 Python 3 中已被移除。您可以使用位于 Python 3 中 `functools` 模块的 `cmp_to_key(func)` 辅助函数，将旧的比较函数转换为键函数。

## 第14.6节：else语句

```
if 条件:
    代码块
else:
    代码块
```

只有当前面的条件语句全部计算为 `False` 时，`else` 语句才会执行其代码块。

```
if True:
    print "这是真的！"
else:
    print "这不会被打印.."

# 输出：这是真的！

if False:
    print "这不会被打印.."
else:
    print "这是假的！"

# 输出：这是错误的！
```

```
>>> if a in (3, 4, 6):
...     print('yes')
... else:
...     print('no')
no
```

## Section 14.5: Using the cmp function to get the comparison result of two objects

Python 2 includes a `cmp` function which allows you to determine if one object is less than, equal to, or greater than another object. This function can be used to pick a choice out of a list based on one of those three options.

Suppose you need to print 'greater than' if  $x > y$ , 'less than' if  $x < y$  and 'equal' if  $x == y$ .

```
['equal', 'greater than', 'less than', ][cmp(x,y)]
# x,y = 1,1 output: 'equal'
# x,y = 1,2 output: 'less than'
# x,y = 2,1 output: 'greater than'
```

`cmp(x,y)` returns the following values

### Comparison Result

$x < y$	-1
$x == y$	0
$x > y$	1

This function is removed on Python 3. You can use the `cmp_to_key(func)` helper function located in `functools` in Python 3 to convert old comparison functions to key functions.

## Section 14.6: Else statement

```
if condition:
    body
else:
    body
```

The `else` statement will execute its body only if preceding conditional statements all evaluate to `False`.

```
if True:
    print "It is true!"
else:
    print "This won't get printed.."

# Output: It is true!

if False:
    print "This won't get printed.."
else:
    print "It is false!"

# Output: It is false!
```

## 第14.7节：测试对象是否为None并赋值

如果对象是None，表示它尚未被赋值，你通常会想给它赋一个值。这里我们使用 aDate。

最简单的方法是使用 is None 测试。

```
if aDate is None:  
    aDate=datetime.date.today()
```

(注意，更符合Python风格的写法是使用 is None而不是== None

。) 但可以稍作优化，利用 not None在布尔表达式中会被评估为True的特性。以下代码等价：

```
if not aDate:  
    aDate=datetime.date.today()
```

但还有一种更符合Python风格的写法。以下代码也等价：

```
aDate=aDate or datetime.date.today()
```

这执行了短路求值。如果 aDate 已初始化且 不是 None，则它被赋值为自身，实际上没有任何效果。如果它 是 None，则将 datetime.date.today()赋值给 aDate。

## 第14.8节：if语句

```
if 条件:  
    代码块
```

if语句检查条件。如果条件计算结果为True，则执行if语句体。如果计算结果为False，则跳过语句体。

```
if True:  
    print "是真的！"  
>> 它是真的！  
  
if False:  
    print "这不会被打印.."
```

条件可以是任何有效的表达式：

```
if 2 + 2 == 4:  
    print "我懂数学！"  
>> 我懂数学！
```

## Section 14.7: Testing if an object is None and assigning it

You'll often want to assign something to an object if it is `None`, indicating it has not been assigned. We'll use `aDate`.

The simplest way to do this is to use the `is None` test.

```
if aDate is None:  
    aDate=datetime.date.today()
```

(Note that it is more Pythonic to say `is None` instead of `== None`.)

But this can be optimized slightly by exploiting the notion that `not None` will evaluate to `True` in a boolean expression. The following code is equivalent:

```
if not aDate:  
    aDate=datetime.date.today()
```

But there is a more Pythonic way. The following code is also equivalent:

```
aDate=aDate or datetime.date.today()
```

This does a Short Circuit evaluation. If `aDate` is initialized and is `not None`, then it gets assigned to itself with no net effect. If it `is None`, then the `datetime.date.today()` gets assigned to `aDate`.

## Section 14.8: If statement

```
if condition:  
    body
```

The `if` statements checks the condition. If it evaluates to `True`, it executes the body of the `if` statement. If it evaluates to `False`, it skips the body.

```
if True:  
    print "It is true!"  
>> It is true!  
  
if False:  
    print "This won't get printed.."
```

The condition can be any valid expression:

```
if 2 + 2 == 4:  
    print "I know math!"  
>> I know math!
```

# 视频：机器学习A-Z：动手Python数据科学

向两位数据科学专家学习如何用Python创建机器学习算法。包含代码模板。



- ✓ 精通Python上的机器学习
- ✓ 对多种机器学习模型有良好的直觉
- ✓ 做出准确的预测
- ✓ 进行强有力的分析
- ✓ 构建稳健的机器学习模型
- ✓ 为您的业务创造强大的附加价值
- ✓ 将机器学习用于个人目的
- ✓ 处理强化学习、自然语言处理和深度学习等特定主题
- ✓ 掌握降维等高级技术
- ✓ 了解针对每种问题应选择哪种机器学习模型
- ✓ 构建一支强大的机器学习模型军团，并知道如何组合它们来解决任何问题

今天观看 →

# VIDEO: Machine Learning A-Z: Hands-On Python In Data Science

Learn to create Machine Learning Algorithms in Python from two Data Science experts. Code templates included.



- ✓ Master Machine Learning on Python
- ✓ Have a great intuition of many Machine Learning models
- ✓ Make accurate predictions
- ✓ Make powerful analysis
- ✓ Make robust Machine Learning models
- ✓ Create strong added value to your business
- ✓ Use Machine Learning for personal purpose
- ✓ Handle specific topics like Reinforcement Learning, NLP and Deep Learning
- ✓ Handle advanced techniques like Dimensionality Reduction
- ✓ Know which Machine Learning model to choose for each type of problem
- ✓ Build an army of powerful Machine Learning models and know how to combine them to solve any problem

Watch Today →

# 第15章：比较

参数	详情
x	第一个待比较项
y	第二个待比较项

## 第15.1节：链式比较

您可以使用链式比较对多个项和多个比较运算符进行比较。例如

```
x > y > z
```

只是以下表达式的简写：

```
x > y and y > z
```

只有当两个比较都为True时，结果才会是True。

一般形式是

```
a OP b OP c OP d ...
```

其中OP表示你可以使用的多种比较操作之一，字母代表任意有效表达式。

注意， $0 \neq 1 \neq 0$ 的结果是True，尽管 $0 \neq 0$ 是False。与常见的数学符号不同， $x \neq y \neq z$ 表示 x、y 和 z 的值都不同。链式使用==操作在大多数情况下有自然的含义，因为等号通常是传递的。

### 风格

只要语法正确，使用多少个项目和比较操作都没有理论上的限制：

```
1 > -1 < 2 > 0.5 < 100 != 24
```

如果每个比较都返回True，上述表达式返回True。然而，使用复杂的链式比较不是好的风格。好的链式比较应该是“有方向的”，且不比下面的表达式更复杂：

```
1 > x > -4 > y != 8
```

### 副作用

一旦有一次比较返回False，表达式会立即计算为False，跳过所有剩余的比较。

注意表达式exp在a > exp > b中只会被计算一次，而在以下情况下

```
a > exp and exp > b
```

如果a > exp为真，exp将被计算两次。

# Chapter 15: Comparisons

Parameter	Details
x	First item to be compared
y	Second item to be compared

## Section 15.1: Chain Comparisons

You can compare multiple items with multiple comparison operators with chain comparison. For example

```
x > y > z
```

is just a short form of:

```
x > y and y > z
```

This will evaluate to True only if both comparisons are True.

The general form is

```
a OP b OP c OP d ...
```

Where OP represents one of the multiple comparison operations you can use, and the letters represent arbitrary valid expressions.

Note that  $0 \neq 1 \neq 0$  evaluates to True, even though  $0 \neq 0$  is False. Unlike the common mathematical notation in which  $x \neq y \neq z$  means that x, y and z have different values. Chaining == operations has the natural meaning in most cases, since equality is generally transitive.

### Style

There is no theoretical limit on how many items and comparison operations you use as long you have proper syntax:

```
1 > -1 < 2 > 0.5 < 100 != 24
```

The above returns True if each comparison returns True. However, using convoluted chaining is not a good style. A good chaining will be "directional", not more complicated than

```
1 > x > -4 > y != 8
```

### Side effects

As soon as one comparison returns False, the expression evaluates immediately to False, skipping all remaining comparisons.

Note that the expression exp in a > exp > b will be evaluated only once, whereas in the case of

```
a > exp and exp > b
```

exp will be computed twice if a > exp is true.

## 第15.2节：使用`is`与`==`进行比较

一个常见的误区是混淆相等比较运算符`is`和`==`。

`a == b`比较的是`a`和`b`的值。

`a is b`比较的是`a`和`b`的身份（对象标识）。

举例说明：

```
a = 'Python is fun!'
b = 'Python is fun!'
a == b # 返回True
a is b # 返回False

a = [1, 2, 3, 4, 5]
b = a      # b引用a
a == b      # True
a is b      # True
b = a[:]    # b 现在引用 a 的一个副本
a == b      # True
a is b      # False [!!]
```

基本上，`is`可以被视为`id(a) == id(b)`的简写。

除此之外，运行时环境的一些特性使情况更加复杂。短字符串和小整数在使用`is`比较时会返回`True`，这是因为 Python 尝试为相同对象使用更少的内存。

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

但较长的字符串和较大的整数会被分别存储。

```
a = 'not so short'
b = 'not so short'
c = 1000
d = 1000
a is b # False
c is d # False
```

你应该使用`is`来测试`None`：

```
if myvar is not None:
    # 不是 None
    pass
if myvar is None:
    # 是 None
    pass
```

使用`is`的一个用途是测试“sentinel”（即一个唯一对象）。

```
sentinel = object()
def myfunc(var=sentinel):
```

## Section 15.2: Comparison by `is` vs `==`

A common pitfall is confusing the equality comparison operators `is` and `==`.

`a == b` compares the value of `a` and `b`.

`a is b` will compare the *identities* of `a` and `b`.

To illustrate:

```
a = 'Python is fun!'
b = 'Python is fun!'
a == b # returns True
a is b # returns False
```

```
a = [1, 2, 3, 4, 5]
b = a      # b references a
a == b      # True
a is b      # True
b = a[:]    # b now references a copy of a
a == b      # True
a is b      # False [!!]
```

Basically, `is` can be thought of as shorthand for `id(a) == id(b)`.

Beyond this, there are quirks of the run-time environment that further complicate things. Short strings and small integers will return `True` when compared with `is`, due to the Python machine attempting to use less memory for identical objects.

```
a = 'short'
b = 'short'
c = 5
d = 5
a is b # True
c is d # True
```

But longer strings and larger integers will be stored separately.

```
a = 'not so short'
b = 'not so short'
c = 1000
d = 1000
a is b # False
c is d # False
```

You should use `is` to test for `None`:

```
if myvar is not None:
    # not None
    pass
if myvar is None:
    # None
    pass
```

A use of `is` is to test for a “sentinel” (i.e. a unique object).

```
sentinel = object()
def myfunc(var=sentinel):
```

```
if var is sentinel:  
    # 没有提供值  
    pass  
else:  
    # 提供了值  
    pass
```

## 第15.3节：大于或小于

```
x > y  
x < y
```

这些运算符比较两种类型的值，它们是小于和大于运算符。对于数字，这只是比较数值大小以确定哪个更大：

```
12 > 4  
# 真  
12 < 4  
# 假  
1 < 4  
# 真
```

对于字符串，它们将按字典序进行比较，这类似于字母顺序，但不完全相同。

```
"alpha" < "beta"  
# 真  
"gamma" > "beta"  
# 真  
"gamma" < "OMEGA"  
# 假
```

在这些比较中，小写字母被视为“大于”大写字母，这就是为什么 "gamma" < "OMEGA" 是错误的原因。如果它们全是大写字母，则会返回预期的字母顺序结果：

```
"GAMMA" < "OMEGA"  
# 真
```

每种类型对<和>运算符的计算定义不同，因此在使用之前，您应当调查给定类型中这些运算符的含义。

## 第15.4节：不等于

```
x != y
```

如果 x 和 y 不相等，则返回True，否则返回False。

```
12 != 1  
# 真  
12 != '12'  
# 真  
'12' != '12'  
# 假
```

```
if var is sentinel:  
    # value wasn't provided  
    pass  
else:  
    # value was provided  
    pass
```

## Section 15.3: Greater than or less than

```
x > y  
x < y
```

These operators compare two types of values, they're the less than and greater than operators. For numbers this simply compares the numerical values to see which is larger:

```
12 > 4  
# True  
12 < 4  
# False  
1 < 4  
# True
```

For strings they will compare lexicographically, which is similar to alphabetical order but not quite the same.

```
"alpha" < "beta"  
# True  
"gamma" > "beta"  
# True  
"gamma" < "OMEGA"  
# False
```

In these comparisons, lowercase letters are considered 'greater than' uppercase, which is why "gamma" < "OMEGA" is false. If they were all uppercase it would return the expected alphabetical ordering result:

```
"GAMMA" < "OMEGA"  
# True
```

Each type defines its calculation with the < and > operators differently, so you should investigate what the operators mean with a given type before using it.

## Section 15.4: Not equal to

```
x != y
```

This returns **True** if x and y are not equal and otherwise returns **False**.

```
12 != 1  
# True  
12 != '12'  
# True  
'12' != '12'  
# False
```

## 第15.5节：等于

```
x == y
```

该表达式用于判断 `x` 和 `y` 是否相同，并返回布尔值结果。通常类型和值都需要匹配，因此整数 `12` 与字符串 `'12'` 不相同。

```
12 == 12  
# 真  
12 == 1  
# 假  
'12' == '12'  
# 真  
'spam' == 'spam'  
# 真  
'spam' == 'spam '  
# 假  
'12' == 12  
# 假
```

请注意，每种类型都必须定义一个函数，用于判断两个值是否相同。对于内置类型，这些函数的行为如你所料，仅基于值是否相同进行判断。然而，自定义类型可以将相等性测试定义为任意方式，包括始终返回 `True` 或始终返回 `False`。

## Section 15.5: Equal To

```
x == y
```

This expression evaluates if `x` and `y` are the same value and returns the result as a boolean value. Generally both type and value need to match, so the int `12` is not the same as the string `'12'`.

```
12 == 12  
# True  
12 == 1  
# False  
'12' == '12'  
# True  
'spam' == 'spam'  
# True  
'spam' == 'spam '  
# False  
'12' == 12  
# False
```

Note that each type has to define a function that will be used to evaluate if two values are the same. For builtin types these functions behave as you'd expect, and just evaluate things based on being the same value. However custom types could define equality testing as whatever they'd like, including always returning `True` or always returning `False`.

## 第15.6节：比较对象

为了比较自定义类的相等性，你可以通过定义 `__eq__` 和 `__ne__` 方法来重载 `==` 和 `!=`。你还可以重载 `__lt__` (`<`)、`__le__` (`<=`)、`__gt__` (`>`) 和 `__ge__` (`>=`)。注意，你只需要重载两个比较方法，Python 可以处理其余的（例如 `==` 等同于 `not <` 且 `not >`，等等）。

```
class Foo(object):  
    def __init__(self, item):  
        self.my_item = item  
    def __eq__(self, other):  
        return self.my_item == other.my_item  
  
a = Foo(5)  
b = Foo(5)  
a == b      # True  
a != b     # False  
a is b      # False
```

请注意，这个简单的比较假设 `other`（被比较的对象）是相同的对象类型。

与另一种类型比较将抛出错误：

```
class Bar(object):  
    def __init__(self, item):  
        self.other_item = item  
    def __eq__(self, other):  
        return self.other_item == other.other_item  
    def __ne__(self, other):  
        return self.other_item != other.other_item  
  
c = Bar(5)  
a == c      # 抛出 AttributeError: 'Foo' 对象没有属性 'other_item'
```

## Section 15.6: Comparing Objects

In order to compare the equality of custom classes, you can override `==` and `!=` by defining `__eq__` and `__ne__` methods. You can also override `__lt__` (`<`), `__le__` (`<=`), `__gt__` (`>`), and `__ge__` (`>=`)。Note that you only need to override two comparison methods, and Python can handle the rest (`==` is the same as `not <` and `not >`, etc.)

```
class Foo(object):  
    def __init__(self, item):  
        self.my_item = item  
    def __eq__(self, other):  
        return self.my_item == other.my_item  
  
a = Foo(5)  
b = Foo(5)  
a == b      # True  
a != b     # False  
a is b      # False
```

Note that this simple comparison assumes that `other` (the object being compared to) is the same object type. Comparing to another type will throw an error:

```
class Bar(object):  
    def __init__(self, item):  
        self.other_item = item  
    def __eq__(self, other):  
        return self.other_item == other.other_item  
    def __ne__(self, other):  
        return self.other_item != other.other_item  
  
c = Bar(5)  
a == c      # throws AttributeError: 'Foo' object has no attribute 'other_item'
```

检查 `isinstance()` 或类似方法可以帮助防止此类错误（如果需要）。

Checking `isinstance()` or similar will help prevent this (if desired).

# 第16章：循环

参数	详情
可以在布尔上下文中求值的布尔表达式，例如 <code>x &lt; 10</code>	
变量	当前元素的变量名，来自可迭代对象
可迭代对象	任何实现了迭代的对象

作为编程中最基本的功能之一，循环是几乎所有编程语言的重要组成部分。循环使开发者能够设置代码的某些部分重复执行多次，这些重复称为迭代。本主题涵盖了在Python中使用多种类型的循环及循环的应用。

## 第16.1节：循环中的break和continue

### break语句

当break语句在循环中执行时，控制流会立即“跳出”循环：

```
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("从循环中跳出")
        break
    i += 1
```

在执行break语句后，循环条件将不再被评估。请注意，语法上break语句仅允许出现在循环内部。函数内部的break语句不能用来终止调用该函数的循环。

执行以下代码会打印每个数字，直到遇到数字4时触发break语句，循环停止：

```
for i in (0, 1, 2, 3, 4):
    print(i)
    if i == 2:
        break
```

执行此循环现在打印：

0  
1  
2  
注意3和4未被打印，因为循环已经结束。

# Chapter 16: Loops

Parameter	Details
boolean expression expression that can be evaluated in a boolean context, e.g. <code>x &lt; 10</code>	
variable variable name for the current element from the iterable	
iterable anything that implements iterations	

As one of the most basic functions in programming, loops are an important piece to nearly every programming language. Loops enable developers to set certain portions of their code to repeat through a number of loops which are referred to as iterations. This topic covers using multiple types of loops and applications of loops in Python.

## Section 16.1: Break and Continue in Loops

### break statement

When a `break` statement executes inside a loop, control flow "breaks" out of the loop immediately:

```
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

The loop conditional will not be evaluated after the `break` statement is executed. Note that `break` statements are only allowed *inside loops*, syntactically. A `break` statement inside a function cannot be used to terminate loops that called that function.

Executing the following prints every digit until number 4 when the `break` statement is met and the loop stops:

```
0
1
2
3
4
Breaking from loop
```

`break` statements can also be used inside `for` loops, the other looping construct provided by Python:

```
for i in (0, 1, 2, 3, 4):
    print(i)
    if i == 2:
        break
```

Executing this loop now prints:

```
0
1
2
```

Note that 3 and 4 are not printed since the loop has ended.

如果循环有一个else子句，当循环通过break语句终止时，该子句不会执行。

## continue语句

continue语句会跳过当前代码块的剩余部分，直接进入下一次循环迭代，但循环会继续执行。与break一样，continue只能出现在循环内部：

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)
```

0  
1  
3  
5  
注意 2和 4--  
print(i) 当 i == 2 或 i == 4 时。

## 嵌套循环

break和continue只作用于单层循环。下面的例子中，break只会跳出内层的for循环，而不会跳出外层的while循环：

```
while True:
    for i in range(1,5):
        if i == 2:
            break # 只会跳出内层循环!
```

Python 没有一次性跳出多层循环的能力——如果需要这种行为，将一个或多个循环重构为函数，并用return替代break可能是可行的方法。

## 在函数内部使用return作为break

return语句会退出函数，不会执行其后的代码。

如果你在函数内部有一个循环，从该循环内使用return等同于使用break，因为循环后面的代码不会被执行（注意循环后的任何代码也不会被执行）：

```
def break_loop():
    for i in range(1, 5):
        if (i == 2):
            return(i)
        print(i)
    return(5)
```

如果你有嵌套循环，return语句将会跳出所有循环：

```
def break_all():
    for j in range(1, 5):
        for i in range(1,4):
            if i*j == 6:
                return(i)
            print(i*j)
```

将输出：

If a loop has an **else** clause, it does not execute when the loop is terminated through a **break** statement.

## continue statement

A **continue** statement will skip to the next iteration of the loop bypassing the rest of the current block but continuing the loop. As with **break**, **continue** can only appear inside loops:

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)
```

0  
1  
3  
5

Note that 2 and 4 aren't printed, this is because **continue** goes to the next iteration instead of continuing on to **print(i)** when **i == 2** or **i == 4**.

## Nested Loops

**break** and **continue** only operate on a single level of loop. The following example will only break out of the inner **for** loop, not the outer **while** loop:

```
while True:
    for i in range(1, 5):
        if i == 2:
            break # Will only break out of the inner loop!
```

Python doesn't have the ability to break out of multiple levels of loop at once -- if this behavior is desired, refactoring one or more loops into a function and replacing **break** with **return** may be the way to go.

## Use return from within a function as a break

The **return** statement exits from a function, without executing the code that comes after it.

If you have a loop inside a function, using **return** from inside that loop is equivalent to having a **break** as the rest of the code of the loop is not executed (*note that any code after the loop is not executed either*):

```
def break_loop():
    for i in range(1, 5):
        if (i == 2):
            return(i)
        print(i)
    return(5)
```

If you have nested loops, the **return** statement will break all loops:

```
def break_all():
    for j in range(1, 5):
        for i in range(1,4):
            if i*j == 6:
                return(i)
            print(i*j)
```

will output:

```
1 # 1*1  
2 # 1*2  
3 # 1*3  
4 # 1*4  
2 # 2*1  
4 # 2*2  
# 返回, 因为 2*3 = 6, 两个循环的剩余迭代不再执行
```

## 第16.2节：for 循环

for 循环遍历一个项目集合，例如 list 或 dict，并对集合中的每个元素执行一段代码。

```
for i in [0, 1, 2, 3, 4]:  
    print(i)
```

上面的 for 循环遍历一个数字列表。

每次迭代将 i 的值设置为列表中的下一个元素。所以首先是 0，然后是 1，接着是 2，依此类推。输出结果如下：

0  
1  
2  
3  
4  
范围

是一个以可迭代形式返回一系列数字的函数，因此它可以用于for循环：

```
for i in range(5):  
    print(i)
```

与第一个for循环的结果完全相同。注意这里的5不会被打印，因为range表示的是从0开始的前五个数字。

### 可迭代对象和迭代器

for循环可以遍历任何可迭代对象，即定义了\_\_getitem\_\_或\_\_iter\_\_函数的对象。  
\_\_iter\_\_函数返回一个迭代器，该对象具有一个next函数，用于访问可迭代对象的下一个元素。

## 第16.3节：遍历列表

要遍历列表，可以使用for：

```
for x in ['one', 'two', 'three', 'four']:  
    print(x)
```

这将打印出列表中的元素：

一  
二  
三  
四

```
1 # 1*1  
2 # 1*2  
3 # 1*3  
4 # 1*4  
2 # 2*1  
4 # 2*2  
# return because 2*3 = 6, the remaining iterations of both loops are not executed
```

## Section 16.2: For loops

for loops iterate over a collection of items, such as `list` or `dict`, and run a block of code with each element from the collection.

```
for i in [0, 1, 2, 3, 4]:  
    print(i)
```

The above for loop iterates over a list of numbers.

Each iteration sets the value of i to the next element of the list. So first it will be 0, then 1, then 2, etc. The output will be as follow:

0  
1  
2  
3  
4

`range` is a function that returns a series of numbers under an iterable form, thus it can be used in for loops:

```
for i in range(5):  
    print(i)
```

gives the exact same result as the first for loop. Note that 5 is not printed as the range here is the first five numbers counting from 0.

### Iterable objects and iterators

for loop can iterate on any iterable object which is an object which defines a \_\_getitem\_\_ or a \_\_iter\_\_ function. The \_\_iter\_\_ function returns an iterator, which is an object with a next function that is used to access the next element of the iterable.

## Section 16.3: Iterating over lists

To iterate through a list you can use for:

```
for x in ['one', 'two', 'three', 'four']:  
    print(x)
```

This will print out the elements of the list:

one  
two  
three  
four

range函数生成的数字通常也用于for循环中。

```
for x in range(1, 6):
    print(x)
```

结果将在python >=3中是一个特殊的[range序列类型](#)，在python <=2中是一个列表。两者都可以通过for循环遍历。

1  
2  
3  
4  
5  
如果你想遍历列表中的元素

并且同时拥有元素的索引，你可以使用Python的[enumerate](#)函数：

```
for index, item in enumerate(['one', 'two', 'three', 'four']):
    print(index, '::', item)
```

enumerate会生成元组，这些元组被解包为index（一个整数）和item（列表中的实际值）。上述循环将打印

```
(0, '::', 'one')
(1, '::', 'two')
(2, '::', 'three')
(3, '::', 'four')
```

使用map和lambda对列表进行迭代并操作值，即对列表中的每个元素应用lambda函数：

```
x = map(lambda e : e.upper(), ['one', 'two', 'three', 'four'])
print(x)
```

输出：

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

注意：在Python 3.x中，map返回的是一个迭代器而不是列表，因此如果需要列表，必须将结果转换  
`print(list(x))`

## 第16.4节：带有“else”子句的循环

for和while复合语句（循环）可以选择性地带有一个else子句（实际上，这种用法相当少见）。

else子句仅在for循环通过完整迭代结束后执行，或者在while循环的条件表达式变为假时执行。

```
for i in range(3):
    print(i)
else:
    print('done')

i = 0
```

The `range` function generates numbers which are also often used in a for loop.

```
for x in range(1, 6):
    print(x)
```

The result will be a special [range sequence type](#) in python >=3 and a list in python <=2. Both can be looped through using the for loop.

```
1
2
3
4
5
```

If you want to loop though both the elements of a list *and* have an index for the elements as well, you can use Python's `enumerate` function:

```
for index, item in enumerate(['one', 'two', 'three', 'four']):
    print(index, '::', item)
```

`enumerate` will generate tuples, which are unpacked into `index` (an integer) and `item` (the actual value from the list). The above loop will print

```
(0, '::', 'one')
(1, '::', 'two')
(2, '::', 'three')
(3, '::', 'four')
```

Iterate over a list with value manipulation using `map` and `lambda`, i.e. apply lambda function on each element in the list:

```
x = map(lambda e : e.upper(), ['one', 'two', 'three', 'four'])
print(x)
```

Output:

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

NB: in Python 3.x `map` returns an iterator instead of a list so you in case you need a list you have to cast the result  
`print(list(x))`

## Section 16.4: Loops with an "else" clause

The `for` and `while` compound statements (loops) can optionally have an `else` clause (in practice, this usage is fairly rare).

The `else` clause only executes after a `for` loop terminates by iterating to completion, or after a `while` loop terminates by its conditional expression becoming false.

```
for i in range(3):
    print(i)
else:
    print('done')

i = 0
```

```
while i < 3:  
    print(i)  
    i += 1  
else:  
    print('done')
```

输出：

0  
1  
2  
done

异常) :

```
for i in range(2):  
    print(i)  
    if i == 1:  
        break  
else:  
    print('done')
```

输出：

0  
1  
done

循环的 else 子句。特别是使用关键字 else 通常被认为令人困惑。

这种子句的最初概念可以追溯到唐纳德·克努斯 (Donald Knuth)，如果我们将循环重写为早期结构化编程之前或低级汇编语言中的 if 语句和 goto 语句，else 关键字的含义就会变得清晰。

例如：

```
while loop_condition():  
    ...  
    if break_condition():  
        break  
    ...
```

等价于：

```
# 伪代码  
  
<<start>>:  
if loop_condition():  
    ...  
    if break_condition():  
        goto <<end>>  
    ...  
    goto <<start>>
```

```
while i < 3:  
    print(i)  
    i += 1  
else:  
    print('done')
```

输出：

0  
1  
2  
done

The **else** clause does *not* execute if the loop terminates some other way (through a **break** statement or by raising an exception):

```
for i in range(2):  
    print(i)  
    if i == 1:  
        break  
else:  
    print('done')
```

输出：

0  
1  
done

Most other programming languages lack this optional **else** clause of loops. The use of the keyword **else** in particular is often considered confusing.

The original concept for such a clause dates back to Donald Knuth and the meaning of the **else** keyword becomes clear if we rewrite a loop in terms of **if** statements and **goto** statements from earlier days before structured programming or from a lower-level assembly language.

For example:

```
while loop_condition():  
    ...  
    if break_condition():  
        break  
    ...
```

is equivalent to:

```
# pseudocode  
  
<<start>>:  
if loop_condition():  
    ...  
    if break_condition():  
        goto <<end>>  
    ...  
    goto <<start>>
```

<<end>>:

如果我们给它们每个都附加一个else子句，它们仍然是等价的。

例如：

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
else:
    print('done')
```

等价于：

```
# 伪代码

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
else:
    print('done')

<<end>>:
```

带有else子句的for循环也可以这样理解。从概念上讲，只要可迭代对象或序列中还有剩余元素，循环条件就保持为True。

为什么要使用这种奇怪的结构？

for...else结构的主要用例是简洁地实现搜索，例如：

```
a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")
```

为了使该结构中的else不那么令人困惑，可以将其理解为“如果没有break”或“如果没有找到”。

关于这方面的一些讨论可以在[\[Python-ideas\] for...else 线程总结、为什么 Python 在 for 和 while 循环后使用 'else'？以及循环语句中的 Else 子句中找到。](#)

## 第16.5节：Pass语句

pass 是一个空语句，用于当 Python 语法要求必须有语句时（例如在for或while循环体内），但程序员不需要或不想执行任何操作时。这在作为尚未编写代码的占位符时非常有用。

```
for x in range(10):
```

<<end>>:

These remain equivalent if we attach an else clause to each of them.

For example:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
else:
    print('done')
```

is equivalent to:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
else:
    print('done')

<<end>>:
```

A for loop with an else clause can be understood the same way. Conceptually, there is a loop condition that remains True as long as the iterable object or sequence still has some remaining elements.

**Why would one use this strange construct?**

The main use case for the for...else construct is a concise implementation of search as for instance:

```
a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")
```

To make the else in this construct less confusing one can think of it as "if not break" or "if not found".

Some discussions on this can be found in [\[Python-ideas\] Summary of for...else threads](#), [Why does python use 'else' after for and while loops?](#), and [Else Clauses on Loop Statements](#)

## Section 16.5: The Pass Statement

pass is a null statement for when a statement is required by Python syntax (such as within the body of a for or while loop), but no action is required or desired by the programmer. This can be useful as a placeholder for code that is yet to be written.

```
for x in range(10):
```

```
pass #我们这里不想做任何事情，或者还没准备好做任何事情，所以我们用 pass
```

在这个例子中，什么都不会发生。该for循环将无错误地完成，但不会执行任何命令或代码。pass允许我们成功运行代码，而不必完全实现所有命令和操作。

同样，pass也可以用于while循环，以及选择结构和函数定义等。

```
while x == y:  
    pass
```

## 第16.6节：遍历字典

考虑以下字典：

```
d = {"a": 1, "b": 2, "c": 3}
```

要遍历其键，可以使用：

```
for key in d:  
    print(key)
```

输出：

```
"a"  
"b"  
"c"
```

这等同于：

```
for key in d.keys():  
    print(key)
```

或者在Python 2中：

```
for key in d.iterkeys():  
    print(key)
```

要遍历其值，请使用：

```
for value in d.values():  
    print(value)
```

输出：

```
pass #we don't want to do anything, or are not ready to do anything here, so we'll pass
```

In this example, nothing will happen. The **for** loop will complete without error, but no commands or code will be actioned. **pass** allows us to run our code successfully without having all commands and action fully implemented.

Similarly, **pass** can be used in **while** loops, as well as in selections and function definitions etc.

```
while x == y:  
    pass
```

## Section 16.6: Iterating over dictionaries

Considering the following dictionary:

```
d = {"a": 1, "b": 2, "c": 3}
```

To iterate through its keys, you can use:

```
for key in d:  
    print(key)
```

Output:

```
"a"  
"b"  
"c"
```

This is equivalent to:

```
for key in d.keys():  
    print(key)
```

or in Python 2:

```
for key in d.iterkeys():  
    print(key)
```

To iterate through its values, use:

```
for value in d.values():  
    print(value)
```

Output:

```
1  
2  
3
```

To iterate through its keys and values, use:

```
for key, value in d.items():  
    print(key, ":", value)
```

Output:

```
a :: 1  
b :: 2  
c :: 3
```

请注意，在 Python 2 中，`.keys()`、`.values()` 和 `.items()` 返回一个 `list` 对象。如果您只需要遍历结果，可以使用等效的 `.iterkeys()`、`.itervalues()` 和 `.iteritems()`。

`.keys()` 和 `.iterkeys()`、`.values()` 和 `.itervalues()`、`.items()` 和 `.iteritems()` 之间的区别在于，`iter*` 方法是生成器。因此，字典中的元素会在被评估时逐个生成。当返回一个 `list` 对象时，所有元素会被打包成一个列表，然后返回以供进一步处理。

还要注意，在 Python 3 中，以上述方式打印的项目顺序不遵循任何特定顺序。

## 第16.7节：“半循环” do-while

与其他语言不同，Python 没有 `do-until` 或 `do-while` 结构（这允许代码在条件测试之前执行一次）。但是，您可以将 `while True` 与 `break` 结合使用来实现相同的目的。

```
a = 10  
while True:  
    a = a-1  
    print(a)  
    if a<7:  
        break  
print('完成。')
```

这将打印：

```
, , )]
```

而不是这样做：

```
for item in collection:  
    i1 = item[0]  
    i2 = item[1]  
    i3 = item[2]  
    # 逻辑
```

或者类似这样：

```
for item in collection:
```

```
a :: 1  
b :: 2  
c :: 3
```

Note that in Python 2, `.keys()`, `.values()` and `.items()` return a `list` object. If you simply need to iterate through the result, you can use the equivalent `.iterkeys()`, `.itervalues()` and `.iteritems()`.

The difference between `.keys()` and `.iterkeys()`, `.values()` and `.itervalues()`, `.items()` and `.iteritems()` is that the `iter*` methods are generators. Thus, the elements within the dictionary are yielded one by one as they are evaluated. When a `list` object is returned, all of the elements are packed into a list and then returned for further evaluation.

Note also that in Python 3, Order of items printed in the above manner does not follow any order.

## Section 16.7: The "half loop" do-while

Unlike other languages, Python doesn't have a `do-until` or a `do-while` construct (this will allow code to be executed once before the condition is tested). However, you can combine a `while True` with a `break` to achieve the same purpose.

```
a = 10  
while True:  
    a = a-1  
    print(a)  
    if a<7:  
        break  
print('Done.')
```

This will print:

```
9  
8  
7  
6  
Done.
```

## Section 16.8: Looping and Unpacking

If you want to loop over a list of tuples for example:

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

instead of doing something like this:

```
for item in collection:  
    i1 = item[0]  
    i2 = item[1]  
    i3 = item[2]  
    # logic
```

or something like this:

```
for item in collection:
```

```
i1, i2, i3 = item  
# 逻辑
```

你可以简单地这样做：

```
for i1, i2, i3 in collection:  
    # 逻辑
```

这也适用于大多数类型的可迭代对象，而不仅仅是元组。

## 第16.9节：使用不同步长迭代列表的不同部分

假设你有一个很长的元素列表，你只对列表中的每隔一个元素感兴趣。也许你只想查看列表的第一个或最后一个元素，或者列表中的特定范围条目。Python内置了强大的索引功能。以下是实现这些场景的一些示例。

下面是一个简单的列表，将在整个示例中使用：

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

### 遍历整个列表

要遍历列表中的每个元素，可以使用如下的for循环：

```
for s in lst:  
    print s[:1] # 打印第一个字母
```

for 循环为 lst 的每个元素赋值给 s。这将打印：

a  
b  
c  
d  
e  
通常你需要元素及其索引。

enumerate 关键字完成此任务。

```
for idx, s in enumerate(lst):  
    print("%s 的索引是 %d" % (s, idx))
```

索引 idx 从零开始，每次迭代递增，而 s 包含当前处理的元素。上述代码将输出：

```
alpha 的索引是 0  
bravo 的索引是 1  
charlie 的索引是 2  
delta 的索引是 3  
echo 的索引是 4
```

### 遍历子列表

如果我们想要遍历一个范围（记住Python使用零基索引），请使用range关键字。

```
i1, i2, i3 = item  
# logic
```

You can simply do this:

```
for i1, i2, i3 in collection:  
    # logic
```

This will also work for *most* types of iterables, not just tuples.

## Section 16.9: Iterating different portion of a list with different step size

Suppose you have a long list of elements and you are only interested in every other element of the list. Perhaps you only want to examine the first or last elements, or a specific range of entries in your list. Python has strong indexing built-in capabilities. Here are some examples of how to achieve these scenarios.

Here's a simple list that will be used throughout the examples:

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

### Iteration over the whole list

To iterate over each element in the list, a **for** loop like below can be used:

```
for s in lst:  
    print s[:1] # print the first letter
```

The **for** loop assigns s for each element of lst. This will print:

a  
b  
c  
d  
e

Often you need both the element and the index of that element. The **enumerate** keyword performs that task.

```
for idx, s in enumerate(lst):  
    print("%s has an index of %d" % (s, idx))
```

The index idx will start with zero and increment for each iteration, while the s will contain the element being processed. The previous snippet will output:

```
alpha has an index of 0  
bravo has an index of 1  
charlie has an index of 2  
delta has an index of 3  
echo has an index of 4
```

### Iterate over sub-list

If we want to iterate over a range (remembering that Python uses zero-based indexing), use the **range** keyword.

```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

这将输出：

```
lst at 2 contains charlie
lst at 3 contains delta
```

列表也可以被切片。以下切片表示法从索引1的元素开始，到末尾，步长为2。  
这两个for循环给出相同的结果。

```
for s in lst[1::2]:
    print(s)

for i in range(1, len(lst), 2):
    print(lst[i])
```

上述代码片段输出：

```
bravo
delta
```

索引和切片是一个独立的话题。

## 第16.10节：while循环

一个while循环会导致循环语句被执行，直到循环条件为假。以下代码将执行循环语句共4次。

```
i = 0
while i < 4:
    #循环语句
    i = i + 1
```

虽然上述循环可以很容易地转换成更优雅的for循环，while循环在检查某些条件是否满足时非常有用。以下循环将持续执行，直到myObject准备好为止。

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

while循环也可以通过使用数字（复数或实数）或True来无条件运行：

```
import cmath

complex_num = cmath.sqrt(-1)
while complex_num:      # 你也可以用任何数字、True或任何类型的值替换 complex_num

    print(complex_num)  # 永远打印 1j
```

如果条件始终为真，while 循环将无限运行（无限循环），除非通过 break、return 语句或异常终止。

```
while True:
    print "无限循环"
```

```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

This would output:

```
lst at 2 contains charlie
lst at 3 contains delta
```

The list may also be sliced. The following slice notation goes from element at index 1 to the end with a step of 2.  
The two for loops give the same result.

```
for s in lst[1::2]:
    print(s)

for i in range(1, len(lst), 2):
    print(lst[i])
```

The above snippet outputs:

```
bravo
delta
```

Indexing and slicing is a topic of its own.

## Section 16.10: While Loop

A while loop will cause the loop statements to be executed until the loop condition is falsey. The following code will execute the loop statements a total of 4 times.

```
i = 0
while i < 4:
    #loop statements
    i = i + 1
```

While the above loop can easily be translated into a more elegant for loop, while loops are useful for checking if some condition has been met. The following loop will continue to execute until myObject is ready.

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

while loops can also run without a condition by using numbers (complex or real) or True:

```
import cmath

complex_num = cmath.sqrt(-1)
while complex_num:      # You can also replace complex_num with any number, True or a value of any type
    print(complex_num)  # Prints 1j forever
```

If the condition is always true the while loop will run forever (infinite loop) if it is not terminated by a break or return statement or an exception.

```
while True:
    print "Infinite loop"
```

```
# 无限循环  
# 无限循环  
# 无限循环  
# ...
```

```
# Infinite loop  
# Infinite loop  
# Infinite loop  
# ...
```

# 第17章：数组

参数	详情
b	表示大小为1字节的有符号整数
B	表示大小为1字节的无符号整数
c	表示大小为1字节的字符
u	表示大小为2字节的Unicode字符
h	表示大小为2字节的有符号整数
H	表示大小为2字节的无符号整数
i	表示大小为2字节的有符号整数
I	表示大小为2字节的无符号整数
w	表示大小为4字节的Unicode字符
l	表示大小为4字节的有符号整数
L	表示大小为4字节的无符号整数
f	表示大小为4字节的浮点数
d	表示大小为8字节的浮点数

Python中的“数组”并不是像C和Java等传统编程语言中的数组，更接近于列表。一个列表可以是同质或异质元素的集合，可能包含整数、字符串或其他列表。

## 第17.1节：通过索引访问单个元素

可以通过索引访问单个元素。Python数组是从零开始索引的。示例如下：

```
my_array = array('i', [1,2,3,4,5])
print(my_array[1])
# 2
print(my_array[2])
# 3
print(my_array[0])
# 1
```

## Chapter 17: Arrays

Parameter	Details
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
u	Represents unicode character of size 2 bytes
h	Represents signed integer of size 2 bytes
H	Represents unsigned integer of size 2 bytes
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
w	Represents unicode character of size 4 bytes
l	Represents signed integer of size 4 bytes
L	Represents unsigned integer of size 4 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

"Arrays" in Python are not the arrays in conventional programming languages like C and Java, but closer to lists. A list can be a collection of either homogeneous or heterogeneous elements, and may contain ints, strings or other lists.

## Section 17.1: Access individual elements through indexes

Individual elements can be accessed through indexes. Python arrays are zero-indexed. Here is an example:

```
my_array = array('i', [1,2,3,4,5])
print(my_array[1])
# 2
print(my_array[2])
# 3
print(my_array[0])
# 1
```

## 第17.2节：数组基础介绍

数组是一种存储相同数据类型值的数据结构。在Python中，这就是数组和列表之间的主要区别。

虽然Python列表可以包含对应不同数据类型的值，但Python中的数组只能包含对应相同数据类型的值。在本教程中，我们将通过几个例子来理解Python数组。

如果你是Python新手，可以先阅读Python入门文章。

要在Python语言中使用数组，你需要导入标准的array模块。这是因为数组不像字符串、整数等那样是基本数据类型。以下是如何在Python中导入array模块：

```
from array import *
```

导入array模块后，你可以声明一个数组。方法如下：

```
arrayIdentifierName = array(typecode, [Initializers])
```

## Section 17.2: Basic Introduction to Arrays

An array is a data structure that stores values of same data type. In Python, this is the main difference between arrays and lists.

While python lists can contain values corresponding to different data types, arrays in python can only contain values corresponding to same data type. In this tutorial, we will understand the Python arrays with few examples.

If you are new to Python, get started with the Python Introduction article.

To use arrays in python language, you need to import the standard `array` module. This is because array is not a fundamental data type like strings, integer etc. Here is how you can import `array` module in python :

```
from array import *
```

Once you have imported the `array` module, you can declare an array. Here is how you do it:

```
arrayIdentifierName = array(typecode, [Initializers])
```

在上述声明中，arrayIdentifierName是数组的名称，typecode让Python知道数组的类型，Initializers是用来初始化数组的值。

类型码是用来定义数组值类型或数组类型的代码。参数部分的表格显示了声明数组及其类型时可以使用的可能值。

下面是一个Python数组声明的实际示例：

```
my_array = array('i',[1,2,3,4])
```

在上面的例子中，使用的类型码是 i。该类型码表示大小为2字节的有符号整数。

这是一个包含5个整数的数组的简单示例

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

## 第17.3节：使用append()方法向数组追加任意值

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

注意，值6被追加到了现有数组的值中。

## 第17.4节：使用insert()方法在数组中插入值

我们可以使用insert()方法在数组的任意索引处插入一个值。示例如下：

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

在上述示例中，值0被插入到了索引0处。注意，第一个参数是索引，第二个参数是值。

## 第17.5节：使用extend()方法扩展Python数组

Python数组可以使用extend()方法扩展多个值。示例如下：

```
my_array = array('i', [1,2,3,4,5])
my_extnd_array = array('i', [7,8,9,10])
my_array.extend(my_extnd_array)
# array('i', [1, 2, 3, 4, 5, 7, 8, 9, 10])
```

我们看到数组 my\_array 被从 my\_extnd\_array 扩展了值。

In the declaration above, arrayIdentifierName is the name of array, typecode lets python know the type of array and Initializers are the values with which array is initialized.

Typecodes are the codes that are used to define the type of array values or the type of array. The table in the parameters section shows the possible values you can use when declaring an array and its type.

Here is a real world example of python array declaration :

```
my_array = array('i',[1,2,3,4])
```

In the example above, typecode used is i. This typecode represents signed integer whose size is 2 bytes.

Here is a simple example of an array containing 5 integers

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

## Section 17.3: Append any value to the array using append() method

```
my_array = array('i', [1,2,3,4,5])
my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

Note that the value 6 was appended to the existing array values.

## Section 17.4: Insert value in an array using insert() method

We can use the insert() method to insert a value at any index of the array. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

In the above example, the value 0 was inserted at index 0. Note that the first argument is the index while second argument is the value.

## Section 17.5: Extend python array using extend() method

A python array can be extended with more than one value using extend() method. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_extnd_array = array('i', [7,8,9,10])
my_array.extend(my_extnd_array)
# array('i', [1, 2, 3, 4, 5, 7, 8, 9, 10])
```

We see that the array my\_array was extended with values from my\_extnd\_array.

## 第17.6节：使用 fromlist() 方法将列表中的项目添加到数组中

下面是一个示例：

```
my_array = array('i', [1,2,3,4,5])
c=[11,12,13]
my_array.fromlist(c)
# array('i', [1, 2, 3, 4, 5, 11, 12, 13])
```

所以我们看到值 11、12 和 13 是从列表 c 添加到 my\_array 中的。

## 第17.7节：使用 remove() 方法删除任意数组元素

下面是一个示例：

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

我们看到元素4已从数组中移除。

## 第17.8节：使用pop()方法移除数组的第一个元素

pop 从数组中移除最后一个元素。示例如下：

```
my_array = array('i', [1,2,3,4,5])
my_array.pop()
# array('i', [1, 2, 3, 4])
```

所以我们看到最后一个元素（5）已从数组中弹出。

## 第17.9节：使用index()

方法通过索引获取任意元素

index() 返回匹配值的第一个索引。请记住数组是从零开始索引的。

```
my_array = array('i', [1,2,3,4,5])
print(my_array.index(5))
# 5
my_array = array('i', [1,2,3,3,5])
print(my_array.index(3))
# 3
```

请注意，在第二个例子中，尽管该值在数组中出现了两次，但只返回了一个索引

## 第17.10节：使用reverse()方法反转Python数组

reverse() 方法的作用正如其名——反转数组。以下是一个示例：

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

## Section 17.6: Add items from list into array using fromlist() method

Here is an example:

```
my_array = array('i', [1,2,3,4,5])
c=[11,12,13]
my_array.fromlist(c)
# array('i', [1, 2, 3, 4, 5, 11, 12, 13])
```

So we see that the values 11,12 and 13 were added from list c to my\_array.

## Section 17.7: Remove any array element using remove() method

Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

We see that the element 4 was removed from the array.

## Section 17.8: Remove last array element using pop() method

pop removes the last element from the array. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.pop()
# array('i', [1, 2, 3, 4])
```

So we see that the last element (5) was popped out of array.

## Section 17.9: Fetch any element through its index using index() method

index() returns first index of the matching value. Remember that arrays are zero-indexed.

```
my_array = array('i', [1,2,3,4,5])
print(my_array.index(5))
# 5
my_array = array('i', [1,2,3,3,5])
print(my_array.index(3))
# 3
```

Note in that second example that only one index was returned, even though the value exists twice in the array

## Section 17.10: Reverse a python array using reverse() method

The reverse() method does what the name says it will do - reverses the array. Here is an example :

```
my_array = array('i', [1,2,3,4,5])
my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

## 第17.11节：通过 buffer\_info()方法获取数组缓冲区信息

该方法提供数组缓冲区在内存中的起始地址和数组中元素的数量。以下是一个示例：

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
(33881712, 5)
```

## 第17.12节：使用count()方法检查元素出现的次数

count() 将返回元素在数组中出现的次数。在下面的例子中，我们看到值 3 出现了两次。

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
# 2
```

## 第17.13节：使用 tostring() 方法将数组转换为字符串

tostring() 将数组转换为字符串。

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

## 第17.14节：使用 tolist() 方法将数组转换为具有相同元素的 Python 列表

当你需要一个 Python list 对象时，可以使用 tolist() 方法将数组转换为列表。

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

## 第17.15节：使用fromstring()方法将字符串追加到字符数组

你可以使用fromstring()方法将字符串追加到字符数组中

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

## Section 17.11: Get array buffer information through buffer\_info() method

This method provides you the array buffer start address in memory and number of elements in array. Here is an example:

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
(33881712, 5)
```

## Section 17.12: Check for number of occurrences of an element using count() method

count() will return the number of times an element appears in an array. In the following example we see that the value 3 occurs twice.

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
# 2
```

## Section 17.13: Convert array to string using tostring() method

tostring() converts the array to a string.

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

## Section 17.14: Convert array to a python list with same elements using tolist() method

When you need a Python list object, you can utilize the tolist() method to convert your array to a list.

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

## Section 17.15: Append a string to char array using fromstring() method

You are able to append a string to a character array using fromstring()

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

# 第18章：多维数组

## 第18.1节：列表中的列表

将二维数组形象化的一个好方法是将其视为列表的列表。类似这样：

```
lst=[[1,2,3],[4,5,6],[7,8,9]]
```

这里外层列表lst包含三个元素。每个元素又是一个列表：第一个是：[1,2,3]，第二个是：[4,5,6]，第三个是：[7,8,9]。你可以像访问列表的其他元素一样访问这些列表，方法如下：

```
print (lst[0])  
#输出: [1, 2, 3]
```

```
print (lst[1])  
#输出: [4, 5, 6]
```

```
print (lst[2])  
#输出: [7, 8, 9]
```

你可以用相同的方式访问这些列表中的不同元素：

```
print (lst[0][0])  
#输出: 1
```

```
print (lst[0][1])  
#输出: 2
```

这里第一个[]括号内的数字表示获取该位置的列表。在上面的例子中，我们使用数字0表示获取第0个位置的列表，即[1,2,3]。第二组[]括号表示从内层列表中获取该位置的元素。在这个例子中，我们使用了0和1，列表中第0个位置的元素是数字1，第1个位置的元素是2。

你也可以用相同的方式设置这些列表中的值：

```
lst[0]=[10,11,12]
```

现在列表变成了[[10,11,12],[4,5,6],[7,8,9]]。在这个例子中，我们将第一个列表整体替换成了一个全新的列表。

```
lst[1][2]=15
```

现在列表变成了[[10,11,12],[4,5,15],[7,8,9]]。在这个例子中，我们修改了内层列表中的一个元素。首先进入位置1的列表，然后修改该列表中位置2的元素，原来是6，现在变成了15。

## 第18.2节：嵌套列表.....

这种行为可以扩展。这里是一个三维数组：

```
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,233]],[[311,312,  
313],[321,322,323],[331,332,333]]]
```

# Chapter 18: Multidimensional arrays

## Section 18.1: Lists in lists

A good way to visualize a 2d array is as a list of lists. Something like this:

```
lst=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

here the outer list lst has three things in it. each of those things is another list: The first one is: [1, 2, 3], the second one is: [4, 5, 6] and the third one is: [7, 8, 9]. You can access these lists the same way you would access another other element of a list, like this:

```
print (lst[0])  
#output: [1, 2, 3]
```

```
print (lst[1])  
#output: [4, 5, 6]
```

```
print (lst[2])  
#output: [7, 8, 9]
```

You can then access the different elements in each of those lists the same way:

```
print (lst[0][0])  
#output: 1
```

```
print (lst[0][1])  
#output: 2
```

Here the first number inside the [] brackets means get the list in that position. In the above example we used the number 0 to mean get the list in the 0th position which is [1, 2, 3]. The second set of [] brackets means get the item in that position from the inner list. In this case we used both 0 and 1 the 0th position in the list we got is the number 1 and in the 1st position it is 2

You can also set values inside these lists the same way:

```
lst[0]=[10, 11, 12]
```

Now the list is [[10, 11, 12], [4, 5, 6], [7, 8, 9]]. In this example we changed the whole first list to be a completely new list.

```
lst[1][2]=15
```

Now the list is [[10, 11, 12], [4, 5, 15], [7, 8, 9]]. In this example we changed a single element inside of one of the inner lists. First we went into the list at position 1 and changed the element within it at position 2, which was 6 now it's 15.

## Section 18.2: Lists in lists in lists ..

This behaviour can be extended. Here is a 3-dimensional array:

```
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,233]],[[311,312,  
313],[321,322,323],[331,332,333]]]
```

显然，这样写会有点难以阅读。使用反斜杠来分隔不同的维度：

```
[[[111, 112, 113], [121, 122, 123], [131, 132, 133]], \  
 [[211, 212, 213], [221, 222, 223], [231, 232, 233]], \  
 [[311, 312, 313], [321, 322, 323], [331, 332, 333]]]
```

通过这样嵌套列表，你可以扩展到任意高的维度。

访问方式类似于二维数组：

```
print(myarray)  
print(myarray[1])  
print(myarray[2][1])  
print(myarray[1][0][2])  
etc.
```

编辑方式也类似：

```
myarray[1]=new_n-1_d_list  
myarray[2][1]=new_n-2_d_list  
myarray[1][0][2]=new_n-3_d_list #或者如果你处理的是三维数组，则为单个数字  
等等。
```

As is probably obvious, this gets a bit hard to read. Use backslashes to break up the different dimensions:

```
[[[111, 112, 113], [121, 122, 123], [131, 132, 133]], \  
 [[211, 212, 213], [221, 222, 223], [231, 232, 233]], \  
 [[311, 312, 313], [321, 322, 323], [331, 332, 333]]]
```

By nesting the lists like this, you can extend to arbitrarily high dimensions.

Accessing is similar to 2D arrays:

```
print(myarray)  
print(myarray[1])  
print(myarray[2][1])  
print(myarray[1][0][2])  
etc.
```

And editing is also similar:

```
myarray[1]=new_n-1_d_list  
myarray[2][1]=new_n-2_d_list  
myarray[1][0][2]=new_n-3_d_list #or a single number if you're dealing with 3D arrays  
etc.
```

# 第19章：字典

参数	详情
键	要查找的目标键
值	要设置或返回的值

## 第19.1节：字典简介

字典是一个键值存储的例子，在Python中也称为映射。它允许你通过引用键来存储和检索元素。由于字典是通过键来引用的，因此查找速度非常快。由于它们主要用于通过键引用项目，所以它们是无序的。

### 创建字典

字典可以通过多种方式初始化：

#### 字面语法

```
d = {}          # 空字典
d = {'key': 'value'}    # 带初始值的字典

Python 3.x 版本 ≥ 3.5

# 也可以使用字面语法解包一个或多个字典

# 创建 otherdict 的浅拷贝
d = {**otherdict}
# 还会用 yetanotherdict 的内容更新浅拷贝。
d = {**otherdict, **yetanotherdict}
```

#### 字典推导式

```
d = {k:v for k,v in [('key', 'value')]}
```

另见：推导式

#### 内置类：dict()

```
d = dict()          # 空字典
d = dict(key='value')    # 显式关键字参数
d = dict([('key', 'value')])  # 传入键/值对列表
# 浅拷贝另一个字典（仅当键都是字符串时可行）
d = dict(**otherdict)
```

#### 修改字典

要向字典添加项，只需创建一个带有值的新键：

```
d['newkey'] = 42
```

也可以添加list和dictionary作为值：

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

要删除项，从字典中删除该键：

```
del d['newkey']
```

# Chapter 19: Dictionary

Parameter	Details
key	The desired key to lookup
value	The value to set or return

## Section 19.1: Introduction to Dictionary

A dictionary is an example of a *key value store* also known as *Mapping* in Python. It allows you to store and retrieve elements by referencing a key. As dictionaries are referenced by key, they have very fast lookups. As they are primarily used for referencing items by key, they are not sorted.

### creating a dict

Dictionaries can be initiated in many ways:

#### literal syntax

```
d = {}          # empty dict
d = {'key': 'value'}    # dict with initial values
```

Python 3.x Version ≥ 3.5

```
# Also unpacking one or multiple dictionaries with the literal syntax is possible

# makes a shallow copy of otherdict
d = {**otherdict}
# also updates the shallow copy with the contents of the yetanotherdict.
d = {**otherdict, **yetanotherdict}
```

#### dict comprehension

```
d = {k:v for k,v in [('key', 'value')]}
```

see also: Comprehensions

#### built-in class: dict()

```
d = dict()          # empty dict
d = dict(key='value')    # explicit keyword arguments
d = dict([('key', 'value')])  # passing in a list of key/value pairs
# make a shallow copy of another dict (only possible if keys are only strings!)
d = dict(**otherdict)
```

#### modifying a dict

To add items to a dictionary, simply create a new key with a value:

```
d['newkey'] = 42
```

It also possible to add list and dictionary as value:

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

To delete an item, delete the key from the dictionary:

```
del d['newkey']
```

## 第19.2节：避免KeyError异常

使用字典时一个常见的陷阱是访问不存在的键。这通常会导致**KeyError**异常

```
mydict = {}
mydict['not there']

追踪 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <模块>
KeyError: 'not there'
```

避免键错误的一种方法是使用`dict.get`方法，该方法允许你指定一个默认值，以便在键缺失时返回。

```
value = mydict.get(key, default_value)
```

如果存在，则返回`mydict[key]`，否则返回`default_value`。注意这不会将key添加到`mydict`中。因此，如果你想保留该键值对，应使用`mydict.setdefault(key, default_value)`，该方法会存储该键值对。

```
mydict = {}
print(mydict)
# {}

print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}

print(mydict.setdefault("foo", "bar"))
# bar
print(mydict)
# {'foo': 'bar'}
```

处理该问题的另一种方法是捕获异常

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

你也可以检查键是否在字典中。

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

但请注意，在多线程环境中，可能在你检查之后键被从字典中移除，导致仍然会抛出异常的竞态条件。

另一种选择是使用`dict`的子类`collections.defaultdict`，它有一个`default_factory`，当给定一个`new_key`时会在字典中创建新条目。

## 第19.3节：遍历字典

如果你将字典用作迭代器（例如在`for`语句中），它会遍历字典的键。例如：

## Section 19.2: Avoiding KeyError Exceptions

One common pitfall when using dictionaries is to access a non-existent key. This typically results in a **KeyError** exception

```
mydict = {}
mydict['not there']

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

One way to avoid key errors is to use the `dict.get` method, which allows you to specify a default value to return in the case of an absent key.

```
value = mydict.get(key, default_value)
```

Which returns `mydict[key]` if it exists, but otherwise returns `default_value`. Note that this doesn't add key to `mydict`. So if you want to retain that key value pair, you should use `mydict.setdefault(key, default_value)`, which *does* store the key value pair.

```
mydict = {}
print(mydict)
# {}

print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}

print(mydict.setdefault("foo", "bar"))
# bar
print(mydict)
# {'foo': 'bar'}
```

An alternative way to deal with the problem is catching the exception

```
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

You could also check if the key is in the dictionary.

```
if key in mydict:
    value = mydict[key]
else:
    value = default_value
```

Do note, however, that in multi-threaded environments it is possible for the key to be removed from the dictionary after you check, creating a race condition where the exception can still be thrown.

Another option is to use a subclass of `dict`, `collections.defaultdict`, that has a `default_factory` to create new entries in the dict when given a `new_key`.

## Section 19.3: Iterating Over a Dictionary

If you use a dictionary as an iterator (e.g. in a `for` statement), it traverses the **keys** of the dictionary. For example:

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
    print(key, d[key])
# c 3
# b 2
# a 1
```

在推导式中使用时也是如此

```
print([key for key in d])
# ['c', 'b', 'a']
```

Python 3.x 版本 ≥ 3.0

items()方法可以用来同时遍历键和值：

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

虽然values()方法可以用来仅遍历值，正如预期的那样：

```
for key, value in d.values():
    print(key, value)
# 3
# 2
# 1
```

Python 2.x 版本 ≥ 2.2

这里，方法keys()、values()和items()返回列表，还有三个额外的方法iterkeys()、itervalues()和iteritems()返回迭代器。

## 第19.4节：带默认值的字典

在标准库中以defaultdict形式提供

```
from collections import defaultdict

d = defaultdict(int)
d['key'] # 0
d['key'] = 5
d['key'] # 5

d = defaultdict(lambda: 'empty')
d['key'] # 'empty'
d['key'] = 'full'
d['key'] # 'full'
```

[\*] 或者，如果你必须使用内置的 dict 类，使用 dict.setdefault() 可以让你在访问之前不存在的键时创建一个默认值：

```
>>> d = {}
>>> d.setdefault('Another_key', []).append("这有效！")
>>> d
```

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
    print(key, d[key])
# c 3
# b 2
# a 1
```

The same is true when used in a comprehension

```
print([key for key in d])
# ['c', 'b', 'a']
```

Python 3.x 版本 ≥ 3.0

The items() method can be used to loop over both the **key** and **value** simultaneously:

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

While the values() method can be used to iterate over only the values, as would be expected:

```
for key, value in d.values():
    print(key, value)
# 3
# 2
# 1
```

Python 2.x 版本 ≥ 2.2

Here, the methods keys(), values() and items() return lists, and there are the three extra methods iterkeys(), itervalues() and iteritems() to return iterators.

## Section 19.4: Dictionary with default values

Available in the standard library as [defaultdict](#)

```
from collections import defaultdict

d = defaultdict(int)
d['key'] # 0
d['key'] = 5
d['key'] # 5

d = defaultdict(lambda: 'empty')
d['key'] # 'empty'
d['key'] = 'full'
d['key'] # 'full'
```

[\*] Alternatively, if you must use the built-in **dict** class, using **dict.setdefault()** will allow you to create a default whenever you access a key that did not exist before:

```
>>> d = {}
>>> d.setdefault('Another_key', []).append("This worked!")
>>> d
```

```
{'Another_key': ['这有效 !']}
```

请记住，如果你有许多值要添加，`dict.setdefault()` 每次调用时都会创建初始值的新实例（在本例中是 `[]`）——这可能会产生不必要的开销。

[\*] Python Cookbook, 第3版, 作者 David Beazley 和 Brian K. Jones (O'Reilly)。版权归 David Beazley 和 Brian Jones 所有, 2013年, ISBN 978-1-449-34037-7。

## 第19.5节：合并字典

考虑以下字典：

```
>>> fish = {'name': 'Nemo', 'hands': '鳍', 'special': '鳃'}
>>> dog = {'name': 'Clifford', 'hands': '爪', 'color': '红色'}
```

Python 3及以上版本

```
>>> fishdog = {**fish, **dog}
>>> fishdog
{'手': '爪', '颜色': '红色', '名字': '克利福德', '特殊': '鳃'}
```

正如这个例子所示，重复的键会映射到它们最后一个对应的值（例如“克利福德”覆盖了“尼莫”）。

Python 3.3+

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'手': '鳍', '颜色': '红色', '特殊': '鳃', '名字': '尼莫'}
```

使用这种技术，给定键的最前面的值优先，而不是最后一个（“克利福德”被抛弃，优先使用“尼莫”）。

Python 2.x, 3.x

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'手': '爪', '颜色': '红色', '名字': '克利福德', '特殊': '鳃'}
```

这使用了最后一个值，就像基于\*\*的合并技术一样（“克利福德”覆盖了“尼莫”）。

```
>>> fish.update(dog)
>>> fish
{'color': '红色', 'hands': '爪子', 'name': '克利福德', 'special': '鳃'}
```

`dict.update` 使用后一个字典覆盖前一个字典。

## 第19.6节：访问键和值

在处理字典时，通常需要访问字典中的所有键和值，无论是在 `for` 循环、列表推导式，还是作为普通列表。

给定如下字典：

```
mydict = {
    'a': '1',
```

```
{'Another_key': ['This worked!']}
```

Keep in mind that if you have many values to add, `dict.setdefault()` will create a new instance of the initial value (in this example a `[]`) every time it's called - which may create unnecessary workloads.

[\*] Python Cookbook, 3rd edition, by David Beazley and Brian K. Jones (O'Reilly). Copyright 2013 David Beazley and Brian Jones, 978-1-449-34037-7.

## Section 19.5: Merging dictionaries

Consider the following dictionaries:

```
>>> fish = {'name': 'Nemo', 'hands': 'fins', 'special': 'gills'}
>>> dog = {'name': 'Clifford', 'hands': 'paws', 'color': 'red'}
```

Python 3.5+

```
>>> fishdog = {**fish, **dog}
>>> fishdog
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

As this example demonstrates, duplicate keys map to their lattermost value (for example "Clifford" overrides "Nemo").

Python 3.3+

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'hands': 'fins', 'color': 'red', 'special': 'gills', 'name': 'Nemo'}
```

With this technique the foremost value takes precedence for a given key rather than the last ("Clifford" is thrown out in favor of "Nemo").

Python 2.x, 3.x

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

This uses the lattermost value, as with the \*\*-based technique for merging ("Clifford" overrides "Nemo").

```
>>> fish.update(dog)
>>> fish
{'color': 'red', 'hands': 'paws', 'name': 'Clifford', 'special': 'gills'}
```

`dict.update` uses the latter dict to overwrite the previous one.

## Section 19.6: Accessing keys and values

When working with dictionaries, it's often necessary to access all the keys and values in the dictionary, either in a `for` loop, a list comprehension, or just as a plain list.

Given a dictionary like:

```
mydict = {
    'a': '1',
```

```
'b': '2'  
}
```

你可以使用 `keys()` 方法获取键的列表：

```
print(mydict.keys())  
# Python2: ['a', 'b']  
# Python3: dict_keys(['b', 'a'])
```

如果你想要一个值的列表，可以使用 `values()` 方法：

```
print(mydict.values())  
# Python2: [1, 2]  
# Python3: dict_values([2, 1])
```

如果你想同时使用键及其对应的值，可以使用 `items()` 方法：

```
print(mydict.items())  
# Python2: [('a', 1), ('b', 2)]  
# Python3: dict_items([('b', 2), ('a', 1)])
```

**注意：**因为字典(`dict`)是无序的，`keys()`、`values()`和`items()`没有排序顺序。如果你关心这些方法返回的顺序，可以使用 `sort()`、`sorted()` 或 `OrderedDict`。

**Python 2/3 差异：**在 Python 3 中，这些方法返回特殊的可迭代对象，而不是列表，相当于 Python 2 中的 `iterkeys()`、`itervalues()` 和 `iteritems()` 方法。这些对象大部分情况下可以像列表一样使用，但存在一些差异。详情请参见 PEP 3106。

## 第19.7节：访问字典的值

```
dictionary = {"Hello": 1234, "World": 5678}  
print(dictionary["Hello"])
```

上述代码将打印 1234。

在此示例中，字符串 "Hello" 称为 key。它用于通过将键放在方括号中来查找 dict 中的值。

数字 1234 出现在 dict 定义中相应的冒号后面。这称为 value，即 "Hello" 在此 dict 中映射到的值。

使用不存在的键查找值将引发 `KeyError` 异常，如果未捕获，程序将停止执行。如果我们想访问值而不冒 `KeyError` 的风险，可以使用 `dictionary.get` 方法。默认情况下，如果键不存在，该方法将返回 `None`。我们可以传递第二个值，在查找失败时返回该值，而不是 `None`。

```
w = dictionary.get("whatever")  
x = dictionary.get("whatever", "nuh-uh")
```

在此示例中，w 将获得值 `None`，x 将获得值 "nuh-uh"。

## 第19.8节：创建字典

创建字典的规则：

```
'b': '2'  
}
```

You can get a list of keys using the `keys()` method:

```
print(mydict.keys())  
# Python2: ['a', 'b']  
# Python3: dict_keys(['b', 'a'])
```

If instead you want a list of values, use the `values()` method:

```
print(mydict.values())  
# Python2: [1, 2]  
# Python3: dict_values([2, 1])
```

If you want to work with both the key and its corresponding value, you can use the `items()` method:

```
print(mydict.items())  
# Python2: [('a', 1), ('b', 2)]  
# Python3: dict_items([('b', 2), ('a', 1)])
```

**NOTE:** Because a `dict` is unsorted, `keys()`, `values()`, and `items()` have no sort order. Use `sort()`, `sorted()`, or an `OrderedDict` if you care about the order that these methods return.

**Python 2/3 Difference:** In Python 3, these methods return special iterable objects, not lists, and are the equivalent of the Python 2 `iterkeys()`, `itervalues()`, and `iteritems()` methods. These objects can be used like lists for the most part, though there are some differences. See [PEP 3106](#) for more details.

## Section 19.7: Accessing values of a dictionary

```
dictionary = {"Hello": 1234, "World": 5678}  
print(dictionary["Hello"])
```

The above code will print 1234.

The string "Hello" in this example is called a key. It is used to lookup a value in the `dict` by placing the key in square brackets.

The number 1234 is seen after the respective colon in the `dict` definition. This is called the value that "Hello" maps to in this `dict`.

Looking up a value like this with a key that does not exist will raise a `KeyError` exception, halting execution if uncaught. If we want to access a value without risking a `KeyError`, we can use the `dictionary.get` method. By default if the key does not exist, the method will return `None`. We can pass it a second value to return instead of `None` in the event of a failed lookup.

```
w = dictionary.get("whatever")  
x = dictionary.get("whatever", "nuh-uh")
```

In this example w will get the value `None` and x will get the value "nuh-uh".

## Section 19.8: Creating a dictionary

Rules for creating a dictionary:

- 每个键必须是唯一的 (否则会被覆盖)
- 每个键必须是可哈希的 (可以使用`hash`函数进行哈希；否则会抛出`TypeError`)
- 键没有特定顺序。

```
# 创建并填充它的值
stock = {'eggs': 5, 'milk': 2}

# 或者创建一个空字典
dictionary = {}

# 然后再填充它
dictionary['eggs'] = 5
dictionary['milk'] = 2

# 值也可以是列表
mydict = {'a': [1, 2, 3], 'b': ['one', 'two', 'three']}

# 使用 list.append() 方法向值列表添加新元素
mydict['a'].append(4) # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three']}
mydict['b'].append('four') # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three', 'four']}

# 我们也可以使用包含两个元素元组的列表来创建字典
iterable = [('eggs', 5), ('milk', 2)]
dictionary = dict(iterable)

# 或者使用关键字参数：
dictionary = dict(eggs=5, milk=2)

# 另一种方法是使用 dict.fromkeys()：
dictionary = dict.fromkeys((milk, eggs)) # => {'milk': None, 'eggs': None}
dictionary = dict.fromkeys((milk, eggs), (2, 5)) # => {'milk': 2, 'eggs': 5}
```

## 第19.9节：创建有序字典

你可以创建一个有序字典，在遍历字典的键时会遵循确定的顺序。

使用collections模块中的OrderedDict。遍历时，这将始终按照原始插入顺序返回字典元素。

```
from collections import OrderedDict

d = OrderedDict()
d['first'] = 1
d['second'] = 2
d['third'] = 3
d['last'] = 4

# 输出 "first 1", "second 2", "third 3", "last 4"
for key in d:
    print(key, d[key])
```

## 第19.10节：使用\*\*运算符解包字典

你可以使用\*\*关键字参数解包操作符，将字典中的键值对传递给函数的参数。以下是官方文档中的一个简化示例：

>>>

- Every key must be **unique** (otherwise it will be overridden)
- Every key must be **hashable** (can use the `hash` function to hash it; otherwise `TypeError` will be thrown)
- There is no particular order for the keys.

```
# Creating and populating it with values
stock = {'eggs': 5, 'milk': 2}

# Or creating an empty dictionary
dictionary = {}

# And populating it after
dictionary['eggs'] = 5
dictionary['milk'] = 2

# Values can also be lists
mydict = {'a': [1, 2, 3], 'b': ['one', 'two', 'three']}

# Use list.append() method to add new elements to the values list
mydict['a'].append(4) # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three']}
mydict['b'].append('four') # => {'a': [1, 2, 3, 4], 'b': ['one', 'two', 'three', 'four']}

# We can also create a dictionary using a list of two-items tuples
iterable = [('eggs', 5), ('milk', 2)]
dictionary = dict(iterable)

# Or using keyword argument:
dictionary = dict(eggs=5, milk=2)

# Another way will be to use the dict.fromkeys():
dictionary = dict.fromkeys((milk, eggs)) # => {'milk': None, 'eggs': None}
dictionary = dict.fromkeys((milk, eggs), (2, 5)) # => {'milk': 2, 'eggs': 5}
```

## Section 19.9: Creating an ordered dictionary

You can create an ordered dictionary which will follow a determined order when iterating over the keys in the dictionary.

Use `OrderedDict` from the `collections` module. This will always return the dictionary elements in the original insertion order when iterated over.

```
from collections import OrderedDict

d = OrderedDict()
d['first'] = 1
d['second'] = 2
d['third'] = 3
d['last'] = 4

# Outputs "first 1", "second 2", "third 3", "last 4"
for key in d:
    print(key, d[key])
```

## Section 19.10: Unpacking dictionaries using the \*\* operator

You can use the `**` keyword argument unpacking operator to deliver the key-value pairs in a dictionary into a function's arguments. A simplified example from the [official documentation](#):

>>>

```
>>> def parrot(voltage, state, action):
...     print("这只鹦鹉不会", action, end=' ')
...     print("如果你给它施加了", voltage, "伏特电压。", end=' ')
...     print("它的状态是", state, "!")
...
>>> d = {"voltage": "四百万", "state": "快死了", "action": "VOOM"}
>>> parrot(**d)
```

这只鹦鹉不会 VOOM，如果你给它施加四百万伏特电压。它的状态是快死了！

从 Python 3.5 开始，你也可以使用这种语法合并任意数量的dict对象。

```
>>> fish = {'name': "尼莫", 'hands': '鳍', 'special': '鳃'}
>>> dog = {'name': "克利福德", 'hands': '爪子', 'color': '红色'}
>>> fishdog = {**fish, **dog}
>>> fishdog
```

{'手': '爪', '颜色': '红色', '名字': '克利福德', '特殊': '鳃'}

正如这个例子所示，重复的键会映射到它们最后一个对应的值（例如“克利福德”覆盖了“尼莫”）。

## 第19.11节：尾随逗号

和列表及元组一样，你可以在字典中包含尾随逗号。

```
role = {"白天": "一个典型的程序员",
        "晚上": "仍然是一个典型的程序员", }
```

PEP 8 规定在尾随逗号和闭合大括号之间应留一个空格。

## 第19.12节：dict() 构造函数

dict() 构造函数可以用来通过关键字参数、单个键值对可迭代对象，或单个字典和关键字参数来创建字典。

```
dict(a=1, b=2, c=3)           # {'a': 1, 'b': 2, 'c': 3}
dict([(d, 4), (e, 5), (f, 6)]) # {'d': 4, 'e': 5, 'f': 6}
dict([(a, 1)], b=2, c=3)       # {'a': 1, 'b': 2, 'c': 3}
dict({'a': 1, 'b': 2}, c=3)    # {'a': 1, 'b': 2, 'c': 3}
```

## 第19.13节：字典示例

字典将键映射到值。

```
car = {}
car["wheels"] = 4
car["color"] = "红色"
car["model"] = "科尔维特"
```

可以通过键访问字典的值。

```
print "Little " + car["color"] + " " + car["model"] + "!"
# 这将打印出 "小红科尔维特!"
```

字典也可以用 JSON 风格创建：

```
>>> def parrot(voltage, state, action):
...     print("This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
```

This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !

As of Python 3.5 you can also use this syntax to merge an arbitrary number of dict objects.

```
>>> fish = {'name': "Nemo", 'hands': 'fins', 'special': 'gills'}
>>> dog = {'name': "Clifford", 'hands': 'paws', 'color': 'red'}
>>> fishdog = {**fish, **dog}
>>> fishdog
```

{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}

As this example demonstrates, duplicate keys map to their lattermost value (for example "Clifford" overrides "Nemo").

## Section 19.11: The trailing comma

Like lists and tuples, you can include a trailing comma in your dictionary.

```
role = {"By day": "A typical programmer",
        "By night": "Still a typical programmer", }
```

PEP 8 dictates that you should leave a space between the trailing comma and the closing brace.

## Section 19.12: The dict() constructor

The dict() constructor can be used to create dictionaries from keyword arguments, or from a single iterable of key-value pairs, or from a single dictionary and keyword arguments.

```
dict(a=1, b=2, c=3)           # {'a': 1, 'b': 2, 'c': 3}
dict([(d, 4), (e, 5), (f, 6)]) # {'d': 4, 'e': 5, 'f': 6}
dict([(a, 1)], b=2, c=3)       # {'a': 1, 'b': 2, 'c': 3}
dict({'a': 1, 'b': 2}, c=3)    # {'a': 1, 'b': 2, 'c': 3}
```

## Section 19.13: Dictionaries Example

Dictionaries map keys to values.

```
car = {}
car["wheels"] = 4
car["color"] = "Red"
car["model"] = "Corvette"
```

Dictionary values can be accessed by their keys.

```
print "Little " + car["color"] + " " + car["model"] + "!"
# This would print out "Little Red Corvette!"
```

Dictionaries can also be created in a JSON style:

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

可以遍历字典的值：

```
for key in car:  
    print key + ": " + car[key]
```

```
# wheels: 4  
# color: Red  
# model: Corvette
```

## 第19.14节：字典值的所有组合

```
options = {  
    "x": ["a", "b"],  
    "y": [10, 20, 30]  
}
```

给定如上所示的字典，其中有一个列表表示要探索的对应键的一组值。假设你想探索 "x"="a" 与 "y"=10，然后 "x"="a" 与 "y"=20，依此类推，直到你探索了所有可能的组合。

您可以使用以下代码创建一个返回所有此类值组合的列表。

```
import itertools  
  
options = {  
    "x": ["a", "b"],  
    "y": [10, 20, 30]}  
  
keys = options.keys()  
values = (options[key] for key in keys)  
combinations = [dict(zip(keys, combination)) for combination in itertools.product(*values)]  
print combinations
```

这将给我们存储在变量 `combinations` 中的以下列表：

```
[{'x': 'a', 'y': 10},  
 {'x': 'b', 'y': 10},  
 {'x': 'a', 'y': 20},  
 {'x': 'b', 'y': 20},  
 {'x': 'a', 'y': 30},  
 {'x': 'b', 'y': 30}]
```

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

Dictionary values can be iterated over:

```
for key in car:  
    print key + ": " + car[key]
```

```
# wheels: 4  
# color: Red  
# model: Corvette
```

## Section 19.14: All combinations of dictionary values

```
options = {  
    "x": ["a", "b"],  
    "y": [10, 20, 30]  
}
```

Given a dictionary such as the one shown above, where there is a list representing a set of values to explore for the corresponding key. Suppose you want to explore "x"="a" with "y"=10, then "x"="a" with "y"=10, and so on until you have explored all possible combinations.

You can create a list that returns all such combinations of values using the following code.

```
import itertools  
  
options = {  
    "x": ["a", "b"],  
    "y": [10, 20, 30]}  
  
keys = options.keys()  
values = (options[key] for key in keys)  
combinations = [dict(zip(keys, combination)) for combination in itertools.product(*values)]  
print combinations
```

This gives us the following list stored in the variable `combinations`:

```
[{'x': 'a', 'y': 10},  
 {'x': 'b', 'y': 10},  
 {'x': 'a', 'y': 20},  
 {'x': 'b', 'y': 20},  
 {'x': 'a', 'y': 30},  
 {'x': 'b', 'y': 30}]
```

# 视频：机器学习 、数据科学和使用 Pyth on的深度学习

完整的动手机器学习教程，涵盖数据科学、Tensorflow、人工智能和神经网络



- ✓ 使用Tensorflow和Keras构建人工神经网络
- ✓ 使用深度学习对图像、数据和情感进行分类
- ✓ 使用线性回归、多项式回归和多元回归进行预测
- ✓ 使用Matplotlib和Seaborn进行数据可视化
- ✓ 利用Apache Spark的MLlib实现大规模机器学习
- ✓ 理解强化学习——以及如何构建吃豆人机器人
- ✓ 使用K-Means聚类、支持向量机（SVM）、K近邻（KNN）、决策树、朴素贝叶斯和主成分分析（PCA）对数据进行分类
- ✓ 使用训练/测试和K折交叉验证选择和调优模型
- ✓ 使用基于物品和基于用户的协同过滤构建电影推荐系统

今天观看 →

# VIDEO: Machine Learning, Data Science and Deep Learning with Python

Complete hands-on machine learning tutorial with data science, Tensorflow, artificial intelligence, and neural networks



- ✓ Build artificial neural networks with Tensorflow and Keras
- ✓ Classify images, data, and sentiments using deep learning
- ✓ Make predictions using linear regression, polynomial regression, and multivariate regression
- ✓ Data Visualization with Matplotlib and Seaborn
- ✓ Implement machine learning at massive scale with Apache Spark's MLlib
- ✓ Understand reinforcement learning - and how to build a Pac-Man bot
- ✓ Classify data using K-Means clustering, Support Vector Machines (SVM), KNN, Decision Trees, Naive Bayes, and PCA
- ✓ Use train/test and K-Fold cross validation to choose and tune your models
- ✓ Build a movie recommender system using item-based and user-based collaborative filtering

Watch Today →

# 第20章：列表

Python中的列表是一种广泛应用的通用数据结构。它们在其他语言中也存在，通常被称为动态数组。列表既是可变的，也是序列数据类型，允许通过索引和切片进行访问。列表可以包含不同类型的对象，包括其他列表对象。

## 第20.1节：列表方法和支持的操作符

从给定列表a开始：

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` – 在列表末尾添加一个新元素。

```
# 向列表中添加值6、7和7
a.append(6)
a.append(7)
a.append(7)
# a: [1, 2, 3, 4, 5, 6, 7, 7]

# 添加另一个列表
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]

# 添加不同类型的元素，因为列表元素不需要具有相同类型
my_string = "hello world"
a.append(my_string)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9], "hello world"]
```

**注意 `append()` 方法只会在列表末尾添加一个新元素。如果你将一个列表追加到另一个列表，追加的列表将作为第一个列表末尾的单个元素。**

```
# 将一个列表追加到另一个列表
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
a[8]
# 返回值: [8,9]
```

2. `extend(enumerable)` – 通过追加另一个可枚举对象的元素来扩展列表。

```
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9, 10]

# 通过追加 b 中的所有元素来扩展列表
a.extend(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]

# 使用非列表的可枚举对象扩展列表:
a.extend(range(3))
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 0, 1, 2]
```

列表也可以用 `+` 运算符连接。注意，这不会修改任何原始列表：

# Chapter 20: List

The Python **List** is a general data structure widely used in Python programs. They are found in other languages, often referred to as *dynamic arrays*. They are both *mutable* and a *sequence* data type that allows them to be *indexed* and *sliced*. The list can contain different types of objects, including other list objects.

## Section 20.1: List methods and supported operators

Starting with a given list a:

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` – appends a new element to the end of the list.

```
# Append values 6, 7, and 7 to the list
a.append(6)
a.append(7)
a.append(7)
# a: [1, 2, 3, 4, 5, 6, 7, 7]

# Append another list
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]

# Append an element of a different type, as list elements do not need to have the same type
my_string = "hello world"
a.append(my_string)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9], "hello world"]
```

**Note that** the `append()` method only appends one new element to the end of the list. If you append a list to another list, the list that you append becomes a single element at the end of the first list.

```
# Appending a list to another list
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
a[8]
# Returns: [8,9]
```

2. `extend(enumerable)` – extends the list by appending elements from another enumerable.

```
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9, 10]

# Extend list by appending all elements from b
a.extend(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]

# Extend list with elements from a non-list enumerable:
a.extend(range(3))
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 0, 1, 2]
```

Lists can also be concatenated with the `+` operator. Note that this does not modify any of the original lists:

```
a = [1, 2, 3, 4, 5, 6] + [7, 7] + b  
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

3. `index(value, [startIndex])` – 获取输入值首次出现的索引。如果输入值不在列表中，则会引发`ValueError`异常。如果提供了第二个参数，则从指定的索引开始搜索。

```
a.index(7)  
# Returns: 6  
  
a.index(49) # ValueError, because 49 is not in a.  
  
a.index(7, 7)  
# Returns: 7  
  
a.index(7, 8) # ValueError, because there is no 7 starting at index 8.
```

4. `insert(index, value)` – 在指定的 `index` 之前插入 `value`。插入后，新元素占据位置 `index`。元素占据位置 `index`。

```
a.insert(0, 0) # 在位置 0 插入 0  
a.insert(2, 5) # 在位置 2 插入 5  
# a: [0, 1, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

5. `pop([index])` – 移除并返回位置 `index` 的元素。如果不传参数，则移除并返回列表的最后一个元素。

```
a.pop(2)  
# Returns: 5  
# a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]  
a.pop(8)  
# Returns: 7  
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
# 无参数时:  
a.pop()  
# Returns: 10  
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6. `remove(value)` – 移除指定值的第一个匹配项。如果找不到指定值，则会抛出 `ValueError` 异常。

```
a.remove(0)  
a.remove(9)  
# a: [1, 2, 3, 4, 5, 6, 7, 8]  
a.remove(10)  
# ValueError, because 10 is not in a
```

7. `reverse()` – 就地反转列表并返回 `None`。

```
a.reverse()  
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

还有其他反转列表的方法。

```
a = [1, 2, 3, 4, 5, 6] + [7, 7] + b  
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

3. `index(value, [startIndex])` – gets the index of the first occurrence of the input value. If the input value is not in the list a `ValueError` exception is raised. If a second argument is provided, the search is started at that specified index.

```
a.index(7)  
# Returns: 6  
  
a.index(49) # ValueError, because 49 is not in a.  
  
a.index(7, 7)  
# Returns: 7  
  
a.index(7, 8) # ValueError, because there is no 7 starting at index 8
```

4. `insert(index, value)` – inserts value just before the specified index. Thus after the insertion the new element occupies position index.

```
a.insert(0, 0) # insert 0 at position 0  
a.insert(2, 5) # insert 5 at position 2  
# a: [0, 1, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

5. `pop([index])` – removes and returns the item at index. With no argument it removes and returns the last element of the list.

```
a.pop(2)  
# Returns: 5  
# a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]  
a.pop(8)  
# Returns: 7  
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
# With no argument:  
a.pop()  
# Returns: 10  
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6. `remove(value)` – removes the first occurrence of the specified value. If the provided value cannot be found, a `ValueError` is raised.

```
a.remove(0)  
a.remove(9)  
# a: [1, 2, 3, 4, 5, 6, 7, 8]  
a.remove(10)  
# ValueError, because 10 is not in a
```

7. `reverse()` – reverses the list in-place and returns `None`.

```
a.reverse()  
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

There are also other ways of reversing a list.

8. count(value) – 统计列表中某个值出现的次数。

```
a.count(7)
# 返回值: 2
```

9. sort() – 按数值和字典序排序列表，返回 None。

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# 按数值顺序排序列表
```

列表也可以通过在 sort() 方法中使用 reverse=True 标志来反转排序顺序。

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

如果想按项目的属性排序，可以使用 key 关键字参数：

```
import datetime

class Person(object):
    def __init__(self, name, birthday, height):
        self.name = name
        self.birthday = birthday
        self.height = height

    def __repr__(self):
        return self.name

l = [Person("约翰·塞纳", datetime.date(1992, 9, 12), 175),
     Person("查克·诺里斯", datetime.date(1990, 8, 28), 180),
     Person("乔恩·斯基特", datetime.date(1991, 7, 6), 185)]

l.sort(key=lambda item: item.name)
# l: [查克·诺里斯, 约翰·塞纳, 乔恩·斯基特]

l.sort(key=lambda item: item.birthday)
# l: [查克·诺里斯, 乔恩·斯基特, 约翰·塞纳]

l.sort(key=lambda item: item.height)
# l: [约翰·塞纳, 查克·诺里斯, 乔恩·斯基特]
```

对于字典列表，概念是相同的：

```
import datetime

l = [{"name": "约翰·塞纳", "birthday": datetime.date(1992, 9, 12), "height": 175},
      {"name": "查克·诺里斯", "birthday": datetime.date(1990, 8, 28), "height": 180},
      {"name": "乔恩·斯基特", "birthday": datetime.date(1991, 7, 6), "height": 185}]

l.sort(key=lambda item: item['name'])
# l: [查克·诺里斯, 约翰·塞纳, 乔恩·斯基特]

l.sort(key=lambda item: item['birthday'])
# l: [查克·诺里斯, 乔恩·斯基特, 约翰·塞纳]

l.sort(key=lambda item: item['height'])
# l: [约翰·塞纳, 查克·诺里斯, 乔恩·斯基特]
```

8. count(value) – counts the number of occurrences of some value in the list.

```
a.count(7)
# Returns: 2
```

9. sort() – sorts the list in numerical and lexicographical order and returns None.

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Sorts the list in numerical order
```

Lists can also be reversed when sorted using the reverse=True flag in the sort() method.

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

If you want to sort by attributes of items, you can use the key keyword argument:

```
import datetime

class Person(object):
    def __init__(self, name, birthday, height):
        self.name = name
        self.birthday = birthday
        self.height = height

    def __repr__(self):
        return self.name

l = [Person("John Cena", datetime.date(1992, 9, 12), 175),
     Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
     Person("Jon Skeet", datetime.date(1991, 7, 6), 185)]

l.sort(key=lambda item: item.name)
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item.birthday)
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item.height)
# l: [John Cena, Chuck Norris, Jon Skeet]
```

In case of list of dicts the concept is the same:

```
import datetime

l = [{"name": "John Cena", "birthday": datetime.date(1992, 9, 12), "height": 175},
      {"name": "Chuck Norris", "birthday": datetime.date(1990, 8, 28), "height": 180},
      {"name": "Jon Skeet", "birthday": datetime.date(1991, 7, 6), "height": 185}]

l.sort(key=lambda item: item['name'])
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item['birthday'])
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

按子字典排序：

```
import datetime

l = [{"姓名": "约翰·塞纳", "生日": datetime.date(1992, 9, 12), "尺寸": {"身高": 175, "体重": 100}}, {"姓名": "查克·诺里斯", "生日": datetime.date(1990, 8, 28), "尺寸": {"身高": 180, "体重": 90}}, {"姓名": "乔恩·斯基特", "生日": datetime.date(1991, 7, 6), "尺寸": {"身高": 185, "体重": 110}}]

l.sort(key=lambda item: item['尺寸']['身高'])
# l: [约翰·塞纳, 查克·诺里斯, 乔恩·斯基特]
```

## 使用attrgetter和itemgetter更好的排序方式

列表也可以使用operator模块中的attrgetter和itemgetter函数进行排序。这些函数有助于提高代码的可读性和可重用性。以下是一些示例，

```
from operator import itemgetter, attrgetter

people = [{"姓名": "chandan", "年龄": 20, "薪水": 2000}, {"姓名": "chetan", "年龄": 18, "薪水": 5000}, {"姓名": "guru", "年龄": 30, "薪水": 3000}]
by_age = itemgetter('年龄')
by_salary = itemgetter('薪水')

people.sort(key=by_age) #按年龄原地排序
people.sort(key=by_salary) #按薪水原地排序
```

itemgetter 也可以传入索引。如果你想根据元组的索引进行排序，这非常有用。

```
list_of_tuples = [(1, 2), (3, 4), (5, 0)]
list_of_tuples.sort(key=itemgetter(1))
print(list_of_tuples) #[[5, 0], [1, 2], [3, 4]]
```

如果你想根据对象的属性排序，可以使用attrgetter，

```
persons = [Person("John Cena", datetime.date(1992, 9, 12), 175), Person("Chuck Norris", datetime.date(1990, 8, 28), 180), Person("Jon Skeet", datetime.date(1991, 7, 6), 185)] #重用上面示例中的Person类
```

```
person.sort(key=attrgetter('name')) #按名字排序
by_birthday = attrgetter('birthday')
person.sort(key=by_birthday) #按生日排序
```

10. clear() – 从列表中移除所有元素

```
a.clear()
# a = []
```

11. 复制–将一个已有列表乘以一个整数，将生成一个包含该数量副本的更大列表原始列表。这在例如列表初始化时非常有用：

```
b = ["blah"] * 3
# b = ["blah", "blah", "blah"]
```

Sort by sub dict:

```
import datetime

l = [{"name": "John Cena", "birthday": datetime.date(1992, 9, 12), "size": {"height": 175, "weight": 100}}, {"name": "Chuck Norris", "birthday": datetime.date(1990, 8, 28), "size": {"height": 180, "weight": 90}}, {"name": "Jon Skeet", "birthday": datetime.date(1991, 7, 6), "size": {"height": 185, "weight": 110}}]

l.sort(key=lambda item: item['size']['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

## Better way to sort using attrgetter and itemgetter

Lists can also be sorted using attrgetter and itemgetter functions from the operator module. These can help improve readability and reusability. Here are some examples,

```
from operator import itemgetter, attrgetter

people = [{"name": "chandan", "age": 20, "salary": 2000}, {"name": "chetan", "age": 18, "salary": 5000}, {"name": "guru", "age": 30, "salary": 3000}]
by_age = itemgetter('age')
by_salary = itemgetter('salary')

people.sort(key=by_age) #in-place sorting by age
people.sort(key=by_salary) #in-place sorting by salary
```

itemgetter 也可以给定一个索引。如果你想根据元组的索引进行排序，这非常有用。

```
list_of_tuples = [(1, 2), (3, 4), (5, 0)]
list_of_tuples.sort(key=itemgetter(1))
print(list_of_tuples) #[[5, 0], [1, 2], [3, 4]]
```

Use the attrgetter if you want to sort by attributes of an object,

```
persons = [Person("John Cena", datetime.date(1992, 9, 12), 175), Person("Chuck Norris", datetime.date(1990, 8, 28), 180), Person("Jon Skeet", datetime.date(1991, 7, 6), 185)] #reusing Person class from above example
```

```
person.sort(key=attrgetter('name')) #sort by name
by_birthday = attrgetter('birthday')
person.sort(key=by_birthday) #sort by birthday
```

10. clear() – removes all items from the list

```
a.clear()
# a = []
```

11. Replication – multiplying an existing list by an integer will produce a larger list consisting of that many copies of the original. This can be useful for example for list initialization:

```
b = ["blah"] * 3
# b = ["blah", "blah", "blah"]
```

```
b = [1, 3, 5] * 5  
# [1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5]
```

如果你的列表包含对对象的引用（例如列表的列表），使用此方法时要小心，详见常见陷阱 - 列表乘法和常见引用。

12. 元素删除-可以使用 `del` 关键字和切片在列表中删除多个元素  
表示法：

```
a = list(range(10))  
del a[:2]  
# a = [1, 3, 5, 7, 9]  
del a[-1]  
# a = [1, 3, 5, 7]  
del a[:]  
# a = []
```

### 13. 复制

默认赋值“=”是将原列表的引用赋给新名称。也就是说，原名称和新名称都指向同一个列表对象。通过其中任何一个所做的更改都会反映在另一个上。这通常不是你想要的。

```
b = a  
a.append(6)  
# b: [1, 2, 3, 4, 5, 6]
```

如果你想创建下面列表的副本，有以下几种选择。

你可以切片：

```
new_list = old_list[:]
```

你可以使用内置的 `list()` 函数：

```
new_list = list(old_list)
```

你可以使用通用的 `copy.copy()`：

```
import copy  
new_list = copy.copy(old_list) #插入原始列表中对象的引用。
```

这比 `list()` 稍慢一些，因为它必须先确定 `old_list` 的数据类型。

如果列表中包含对象，并且你也想复制它们，请使用通用的 `copy.deepcopy()`：

```
import copy  
new_list = copy.deepcopy(old_list) # 插入原始列表中对象的副本。
```

显然这是最慢且最占内存的方法，但有时不可避免。

```
b = [1, 3, 5] * 5  
# [1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5]
```

Take care doing this if your list contains references to objects (eg a list of lists), see Common Pitfalls - List multiplication and common references.

12. **Element deletion** – it is possible to delete multiple elements in the list using the `del` keyword and slice notation:

```
a = list(range(10))  
del a[:2]  
# a = [1, 3, 5, 7, 9]  
del a[-1]  
# a = [1, 3, 5, 7]  
del a[:]  
# a = []
```

### 13. Copying

The default assignment “=” assigns a reference of the original list to the new name. That is, the original name and new name are both pointing to the same list object. Changes made through any of them will be reflected in another. This is often not what you intended.

```
b = a  
a.append(6)  
# b: [1, 2, 3, 4, 5, 6]
```

If you want to create a copy of the list you have below options.

You can slice it:

```
new_list = old_list[:]
```

You can use the built in `list()` function:

```
new_list = list(old_list)
```

You can use generic `copy.copy()`:

```
import copy  
new_list = copy.copy(old_list) #inserts references to the objects found in the original.
```

This is a little slower than `list()` because it has to find out the datatype of `old_list` first.

If the list contains objects and you want to copy them as well, use generic `copy.deepcopy()`:

```
import copy  
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Obviously the slowest and most memory-needing method, but sometimes unavoidable.

`copy()` – 返回列表的浅拷贝

```
aa = a.copy()  
# aa = [1, 2, 3, 4, 5]
```

## 第20.2节：访问列表值

Python 列表是从零开始索引的，行为类似其他语言中的数组。

```
lst = [1, 2, 3, 4]  
lst[0] # 1  
lst[1] # 2
```

尝试访问列表边界之外的索引将引发`IndexError`。

```
lst[4] # IndexError: list index out of range
```

负索引被解释为从列表的末尾开始计数。

```
lst[-1] # 4  
lst[-2] # 3  
lst[-5] # IndexError: list index out of range
```

这在功能上等同于

```
lst[len(lst)-1] # 4
```

列表允许使用切片表示法，格式为`lst[start:end:step]`。切片表示法的输出是一个新列表，包含从索引start到end-1的元素。如果省略选项，start默认为列表开头，end默认为列表末尾，step默认为1：

```
lst[1:] # [2, 3, 4]  
lst[:3] # [1, 2, 3]  
lst[::2] # [1, 3]  
lst[::-1] # [4, 3, 2, 1]  
lst[-1:0:-1] # [4, 3, 2]  
lst[5:8] # [] 因为起始索引大于 lst 的长度，返回空列表  
lst[1:10] # [2, 3, 4] 与省略结束索引相同
```

考虑到这一点，你可以通过调用来打印列表的反转版本

```
lst[::-1] # [4, 3, 2, 1]
```

当使用负步长时，起始索引必须大于结束索引，否则结果将是空列表。

```
lst[3:1:-1] # [4, 3]
```

使用负步长索引等同于以下代码：

```
reversed(lst)[0:2] # 0 = 1 -1  
# 2 = 3 -1
```

使用的索引比负索引中使用的索引小 1，并且顺序相反。

`copy()` – Returns a shallow copy of the list

```
aa = a.copy()  
# aa = [1, 2, 3, 4, 5]
```

## Section 20.2: Accessing list values

Python lists are zero-indexed, and act like arrays in other languages.

```
lst = [1, 2, 3, 4]  
lst[0] # 1  
lst[1] # 2
```

Attempting to access an index outside the bounds of the list will raise an `IndexError`.

```
lst[4] # IndexError: list index out of range
```

Negative indices are interpreted as counting from the *end* of the list.

```
lst[-1] # 4  
lst[-2] # 3  
lst[-5] # IndexError: list index out of range
```

This is functionally equivalent to

```
lst[len(lst)-1] # 4
```

Lists allow to use *slice notation* as `lst[start:end:step]`. The output of the slice notation is a new list containing elements from index start to end-1. If options are omitted start defaults to beginning of list, end to end of list and step to 1:

```
lst[1:] # [2, 3, 4]  
lst[:3] # [1, 2, 3]  
lst[::2] # [1, 3]  
lst[::-1] # [4, 3, 2, 1]  
lst[-1:0:-1] # [4, 3, 2]  
lst[5:8] # [] since starting index is greater than length of lst, returns empty list  
lst[1:10] # [2, 3, 4] same as omitting ending index
```

With this in mind, you can print a reversed version of the list by calling

```
lst[::-1] # [4, 3, 2, 1]
```

When using step lengths of negative amounts, the starting index has to be greater than the ending index otherwise the result will be an empty list.

```
lst[3:1:-1] # [4, 3]
```

Using negative step indices are equivalent to the following code:

```
reversed(lst)[0:2] # 0 = 1 -1  
# 2 = 3 -1
```

The indices used are 1 less than those used in negative indexing and are reversed.

## 高级切片

当列表被切片时，会调用列表对象的`__getitem__()`方法，传入一个slice对象。Python内置了一个slice方法来生成切片对象。我们可以使用它来存储一个切片，并在以后重用，如下所示，

```
data = 'chandan purohit    22 2000' #假设数据字段长度固定
name_slice = slice(0,19)
age_slice = slice(19,21)
salary_slice = slice(22,None)

#现在我们可以使用更易读的切片
print(data[name_slice]) #chandan purohit
print(data[age_slice]) #'22'
print(data[salary_slice]) #'2000'
```

通过在我们的类中重写`__getitem__`方法，这对于为我们的对象提供切片功能非常有用。

## 第20.3节：检查列表是否为空

列表的空状态对应布尔值`False`，因此你不必检查`len(lst) == 0`，只需检查`lst`或`not lst`

```
lst = []
if not lst:
    print("list is empty")

# 输出: list is empty
```

## 第20.4节：遍历列表

Python支持直接对列表使用for循环：

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)

# 输出: foo
# 输出: bar
# 输出: baz
```

你也可以同时获取每个元素的位置：

```
for (index, item) in enumerate(my_list):
    print('位置为 {} 的项目是 : {}'.format(index, item))

# 输出: 位置为 0 的项目是 : foo
# 输出: 位置为 1 的项目是 : bar
# 输出: 位置为 2 的项目是 : baz
```

基于索引值迭代列表的另一种方式：

```
for i in range(0,len(my_list)):
    print(my_list[i])

#输出:
>>>
foo
bar
```

## Advanced slicing

When lists are sliced the `__getitem__()` method of the list object is called, with a `slice` object. Python has a builtin slice method to generate slice objects. We can use this to store a slice and reuse it later like so,

```
data = 'chandan purohit    22 2000' #assuming data fields of fixed length
name_slice = slice(0,19)
age_slice = slice(19,21)
salary_slice = slice(22,None)

#now we can have more readable slices
print(data[name_slice]) #chandan purohit
print(data[age_slice]) #'22'
print(data[salary_slice]) #'2000'
```

This can be of great use by providing slicing functionality to our objects by overriding `__getitem__` in our class.

## Section 20.3: Checking if list is empty

The emptiness of a list is associated to the boolean `False`, so you don't have to check `len(lst) == 0`, but just `lst` or `not lst`

```
lst = []
if not lst:
    print("list is empty")

# Output: list is empty
```

## Section 20.4: Iterating over a list

Python supports using a `for` loop directly on a list:

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)

# Output: foo
# Output: bar
# Output: baz
```

You can also get the position of each item at the same time:

```
for (index, item) in enumerate(my_list):
    print('The item in position {} is: {}'.format(index, item))

# Output: The item in position 0 is: foo
# Output: The item in position 1 is: bar
# Output: The item in position 2 is: baz
```

The other way of iterating a list based on the index value:

```
for i in range(0,len(my_list)):
    print(my_list[i])

#output:
>>>
foo
bar
```

注意，在迭代列表时更改列表中的项目可能会产生意想不到的结果：

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)
```

```
# 输出: foo
# 输出: baz
```

在最后这个例子中，我们在第一次迭代时删除了第一个元素，但这导致bar被跳过。

## 第20.5节：检查某个元素是否在列表中

Python 使得检查某个元素是否在列表中非常简单。只需使用in操作符。

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# 输出: True
```

```
'toast' in lst
# 输出: False
```

注意：集合上的in操作符在渐进时间复杂度上比列表更快。如果你需要在可能很大的列表上多次使用它，可能需要将你的list转换为set，并在set上测试元素的存在性。

```
slist = set(lst)
'test' in slist
# 输出: True
```

## 第20.6节：Any 和 All

你可以使用all()来判断可迭代对象中的所有值是否都为真

```
nums = [1, 1, 0, 1]
all(nums)
# 假
chars = ['a', 'b', 'c', 'd']
all(chars)
# 真
```

同样，any() 用于判断可迭代对象中是否有一个或多个值为真

```
nums = [1, 1, 0, 1]
any(nums)
# 真
vals = [None, None, None, False]
any(vals)
# 假
```

虽然此示例使用的是列表，但需要注意这些内置函数适用于任何可迭代对象，包括生成器。

Note that changing items in a list while iterating on it may have unexpected results:

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)
```

```
# Output: foo
# Output: baz
```

In this last example, we deleted the first item at the first iteration, but that caused bar to be skipped.

## Section 20.5: Checking whether an item is in a list

Python makes it very simple to check whether an item is in a list. Simply use the `in` operator.

```
lst = ['test', 'twest', 'tweast', 'treast']

'test' in lst
# Out: True
```

```
'toast' in lst
# Out: False
```

Note: the `in` operator on sets is asymptotically faster than on lists. If you need to use it many times on potentially large lists, you may want to convert your `list` to a `set`, and test the presence of elements on the `set`.

```
slist = set(lst)
'test' in slist
# Out: True
```

## Section 20.6: Any and All

You can use `all()` to determine if all the values in an iterable evaluate to True

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

Likewise, `any()` determines if one or more values in an iterable evaluate to True

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

While this example uses a list, it is important to note these built-ins work with any iterable, including generators.

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

## 第20.7节：反转列表元素

你可以使用reversed函数，它返回一个指向反转列表的迭代器：

```
In [3]: rev = reversed(numbers)

In [4]: rev
输出[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

请注意，列表“numbers”在此操作中保持不变，且顺序与原来相同。

要原地反转，也可以使用reverse方法。

你也可以通过使用切片语法反转列表（实际上是获得一个副本，原列表不受影响），将第三个参数（步长）设置为 -1：

```
In [1]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: numbers[::-1]
输出[2]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

## 第20.8节：连接和合并列表

### 1. 连接list1和list2的最简单方法：

```
merged = list1 + list2
```

### 2. zip 返回一个元组列表，其中第 i 个元组包含来自每个参数的第 i 个元素序列或可迭代对象：

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3']

for a, b in zip(alist, blist):
    print(a, b)

# 输出：
# a1 b1
# a2 b2
# a3 b3
```

如果列表长度不同，则结果只包含与最短列表长度相同的元素：

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3', 'b4']
for a, b in zip(alist, blist):
    print(a, b)

# 输出：
# a1 b1
```

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

## Section 20.7: Reversing list elements

You can use the `reversed` function which returns an iterator to the reversed list:

```
In [3]: rev = reversed(numbers)

In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Note that the list "numbers" remains unchanged by this operation, and remains in the same order it was originally.

To reverse in place, you can also use the `reverse` method.

You can also reverse a list (actually obtaining a copy, the original list is unaffected) by using the slicing syntax, setting the third argument (the step) as -1:

```
In [1]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: numbers[::-1]
Out[2]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

## Section 20.8: Concatenate and Merge lists

### 1. The simplest way to concatenate list1 and list2:

```
merged = list1 + list2
```

### 2. zip returns a list of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables:

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3']

for a, b in zip(alist, blist):
    print(a, b)

# Output:
# a1 b1
# a2 b2
# a3 b3
```

If the lists have different lengths then the result will include only as many elements as the shortest one:

```
alist = ['a1', 'a2', 'a3']
blist = ['b1', 'b2', 'b3', 'b4']
for a, b in zip(alist, blist):
    print(a, b)

# Output:
# a1 b1
```

```
# a2 b2
# a3 b3

alist = []
len(list(zip(alist, blist)))

# 输出:
# 0
```

对于用None填充长度不等的列表，使其与最长列表长度一致，使用`itertools.zip_longest`  
(Python 2中为`itertools.izip_longest`)

```
alist = ['a1', 'a2', 'a3']
blist = ['b1']
clist = ['c1', 'c2', 'c3', 'c4']

for a,b,c in itertools.zip_longest(alist, blist, clist):
    print(a, b, c)

# 输出:
# a1 b1 c1
# a2 None c2
# a3 None c3
# None None c4
```

### 3. 向特定索引插入值：

```
alist = [123, 'xyz', 'zara', 'abc']
alist.insert(3, [2009])
print("最终列表 :", alist)
```

输出：

```
最终列表 : [123, 'xyz', 'zara', 2009, 'abc']
```

## 第20.9节：列表的长度

使用`len()`获取列表的一维长度。

```
len(['one', 'two']) # 返回2
len(['one', [2, 3], 'four']) # 返回3, 而不是4
```

`len()`也适用于字符串、字典以及其他类似列表的数据结构。

注意`len()`是内置函数，而不是列表对象的方法。

还要注意，`len()`的时间复杂度是O(1)，意味着获取列表长度所需时间与列表长度无关。

## 第20.10节：移除列表中的重复值

移除列表中的重复值可以通过将列表转换为set（即无序且不重复的对象集合）来实现。如果需要list数据结构，则可以使用函数`list()`将集合转换回列表：

```
# a2 b2
# a3 b3

alist = []
len(list(zip(alist, blist)))

# Output:
# 0
```

For padding lists of unequal length to the longest one with Nones use `itertools.zip_longest`  
(`itertools.izip_longest` in Python 2)

```
alist = ['a1', 'a2', 'a3']
blist = ['b1']
clist = ['c1', 'c2', 'c3', 'c4']

for a,b,c in itertools.zip_longest(alist, blist, clist):
    print(a, b, c)

# Output:
# a1 b1 c1
# a2 None c2
# a3 None c3
# None None c4
```

### 3. Insert to a specific index values:

```
alist = [123, 'xyz', 'zara', 'abc']
alist.insert(3, [2009])
print("Final List :", alist)
```

Output:

```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

## Section 20.9: Length of a list

Use `len()` to get the one-dimensional length of a list.

```
len(['one', 'two']) # returns 2
len(['one', [2, 3], 'four']) # returns 3, not 4
```

`len()` also works on strings, dictionaries, and other data structures similar to lists.

Note that `len()` is a built-in function, not a method of a list object.

Also note that the cost of `len()` is O(1), meaning it will take the same amount of time to get the length of a list regardless of its length.

## Section 20.10: Remove duplicate values in list

Removing duplicate values in a list can be done by converting the list to a set (that is an unordered collection of distinct objects). If a list data structure is needed, then the set can be converted back to a list using the function `list()`:

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# 输出: ['duke', 'tofp', 'aixk', 'edik']
```

注意，通过将列表转换为集合，原有的顺序会丢失。

为了保持列表的顺序，可以使用OrderedDict

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# 输出: ['aixk', 'duke', 'edik', 'tofp']
```

## 第20.11节：列表比较

可以使用比较运算符按字典序比较列表和其他序列。两个操作数必须是相同类型。

```
[1, 10, 100] < [2, 10, 100]
# True, 因为 1 < 2
[1, 10, 100] < [1, 10, 100]
# False, 因为列表相等
[1, 10, 100] <= [1, 10, 100]
# True, 因为列表相等
[1, 10, 100] < [1, 10, 101]
# True, 因为 100 < 101
[1, 10, 100] < [0, 10, 100]
# False, 因为 0 < 1
```

如果一个列表包含在另一个列表的开头，较短的列表获胜。

```
[1, 10] < [1, 10, 100]
# 真
```

## 第20.12节：访问嵌套列表中的值

从一个三维列表开始：

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]]
```

访问列表中的元素：

```
print(alist[0][0][1])
#2
#访问第一个列表中的第一个列表的第二个元素

print(alist[1][1][2])
#10
#访问第二个列表中的第二个列表的第三个元素
```

执行支持操作：

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#将11追加到第一个列表中的第一个列表的末尾
```

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Out: ['duke', 'tofp', 'aixk', 'edik']
```

Note that by converting a list to a set the original ordering is lost.

To preserve the order of the list one can use an OrderedDict

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Out: ['aixk', 'duke', 'edik', 'tofp']
```

## Section 20.11: Comparison of lists

It's possible to compare lists and other sequences lexicographically using comparison operators. Both operands must be of the same type.

```
[1, 10, 100] < [2, 10, 100]
# True, because 1 < 2
[1, 10, 100] < [1, 10, 100]
# False, because the lists are equal
[1, 10, 100] <= [1, 10, 100]
# True, because the lists are equal
[1, 10, 100] < [1, 10, 101]
# True, because 100 < 101
[1, 10, 100] < [0, 10, 100]
# False, because 0 < 1
```

If one of the lists is contained at the start of the other, the shortest list wins.

```
[1, 10] < [1, 10, 100]
# True
```

## Section 20.12: Accessing values in nested list

Starting with a three-dimensional list:

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10], [12, 13, 14]]]
```

Accessing items in the list:

```
print(alist[0][0][1])
#2
#Accesses second element in the first list in the first list

print(alist[1][1][2])
#10
#Accesses the third element in the second list in the second list
```

Performing support operations:

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#Appends 11 to the end of the first list in the first list
```

使用嵌套的for循环打印列表：

```
for row in alist: #遍历嵌套列表的一种方法
    for col in row:
        print(col)
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#[12, 13, 14]
```

注意，这个操作也可以用于列表推导式，甚至作为生成器以提高效率，例如：

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

外层列表中的元素不一定全部是列表：

```
alist[1].insert(2, 15)
#在第二个列表的第三个位置插入15
```

另一种使用嵌套for循环的方法。另一种方法更好，但我有时需要用这种：

```
for row in range(len(alist)): #遍历列表的一种不太Pythonic的方式
    for col in range(len(alist[row])):
        print(alist[row][col])
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#15
#[12, 13, 14]
```

在嵌套列表中使用切片：

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#切片仍然有效
```

最终名单：

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

## 第20.13节：将列表初始化为固定数量的元素

对于不可变元素（例如None，字符串字面量等）：

```
my_list = [None] * 10
my_list = ['test'] * 10
```

对于可变元素，相同的构造会导致列表中的所有元素都引用同一个对象，例如，对于集合：

```
>>> my_list=[{1}] * 10
```

Using nested for loops to print the list:

```
for row in alist: #One way to loop through nested lists
    for col in row:
        print(col)
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#[12, 13, 14]
```

Note that this operation can be used in a list comprehension or even as a generator to produce efficiencies, e.g.:

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

Not all items in the outer lists have to be lists themselves:

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

Another way to use nested for loops. The other way is better but I've needed to use this on occasion:

```
for row in range(len(alist)): #A less Pythonic way to loop through lists
    for col in range(len(alist[row])):
        print(alist[row][col])
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#15
#[12, 13, 14]
```

Using slices in nested list:

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
#Slices still work
```

The final list:

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

## Section 20.13: Initializing a List to a Fixed Number of Elements

For **immutable** elements (e.g. `None`, string literals etc.):

```
my_list = [None] * 10
my_list = ['test'] * 10
```

For **mutable** elements, the same construct will result in all elements of the list referring to the same object, for example, for a set:

```
>>> my_list=[{1}] * 10
```

```
>>> print(my_list)
[{}, {}, {}, {}, {}, {}, {}, {}]
>>> my_list[0].add(2)
>>> print(my_list)
[{}, 2}, {}, 2}, {}, 2}, {}, 2}, {}, 2}, {}, 2}]
```

相反，要用固定数量的不同的可变对象初始化列表，应使用：

```
my_list=[{} for _ in range(10)]
```

```
>>> print(my_list)
[{}, {}, {}, {}, {}, {}, {}, {}, {}]
>>> my_list[0].add(2)
>>> print(my_list)
[{}, 2}, {}, 2}, {}, 2}, {}, 2}, {}, 2}, {}, 2}]
```

Instead, to initialize the list with a fixed number of **different mutable** objects, use:

```
my_list=[{} for _ in range(10)]
```

# 第21章：列表推导式

Python中的列表推导式是一种简洁的语法结构。它们可以通过对列表中的每个元素应用函数，从其他列表生成新的列表。以下部分解释并演示了这些表达式的用法。

## 第21.1节：列表推导式

一个列表推导式通过对可迭代对象的每个元素应用表达式来创建一个新的列表。最基本的形式是：

```
[ <表达式> for <元素> in <可迭代对象> ]
```

还有一个可选的“if”条件：

```
[ <表达式> for <元素> in <可迭代对象> if <条件> ]
```

如果（可选的）<条件>为真，则将<可迭代对象>中的每个<元素>代入<表达式>。所有结果一次性返回到新的列表中。生成器表达式是惰性求值的，但列表推导式会立即计算整个迭代器——消耗与迭代器长度成比例的内存。

创建一个整数平方的列表：

```
squares = [x * x for x in (1, 2, 3, 4)]  
# squares: [1, 4, 9, 16]
```

该for表达式依次将x设置为(1, 2, 3, 4)中的每个值。表达式x \* x的结果被追加到内部列表中。完成后，内部列表被赋值给变量squares。

除了速度提升（如此处所述），列表推导大致等同于以下的for循环：

```
squares = []  
for x in (1, 2, 3, 4):  
    squares.append(x * x)  
# squares: [1, 4, 9, 16]
```

可以对每个元素应用任意复杂的表达式：

```
# 从字符串中获取大写字母列表  
[s.upper() for s in "Hello World"]  
# ['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']  
  
# 去除列表中字符串末尾的逗号  
[w.strip(',') for w in ['these,', 'words,,', 'mostly', 'have,commas,']]  
# ['these', 'words', 'mostly', 'have,commas']  
  
# 更合理地组织单词中的字母——按字母顺序排列  
sentence = "Beautiful is better than ugly"  
["".join(sorted(word, key = lambda x: x.lower())) for word in sentence.split()]  
# ['aBefiltuu', 'is', 'beertt', 'ahnt', 'gluy']
```

else

else 可以用于列表推导式，但要注意语法。if/else 子句应

# Chapter 21: List comprehensions

List comprehensions in Python are concise, syntactic constructs. They can be utilized to generate lists from other lists by applying functions to each element in the list. The following section explains and demonstrates the use of these expressions.

## Section 21.1: List Comprehensions

A [list comprehension](#) creates a new [list](#) by applying an expression to each element of an iterable. The most basic form is:

```
[ <expression> for <element> in <iterable> ]
```

There's also an optional 'if' condition:

```
[ <expression> for <element> in <iterable> if <condition> ]
```

Each [`<element>`](#) in the [`<iterable>`](#) is plugged in to the [`<expression>`](#) if the (optional) [`<condition>`](#) [evaluates to true](#). All results are returned at once in the new list. Generator expressions are evaluated lazily, but list comprehensions evaluate the entire iterator immediately - consuming memory proportional to the iterator's length.

To create a [list](#) of squared integers:

```
squares = [x * x for x in (1, 2, 3, 4)]  
# squares: [1, 4, 9, 16]
```

The [`for`](#) expression sets x to each value in turn from (1, 2, 3, 4). The result of the expression x \* x is appended to an internal [list](#). The internal [list](#) is assigned to the variable squares when completed.

Besides a [speed increase](#) (as explained [here](#)), a list comprehension is roughly equivalent to the following for-loop:

```
squares = []  
for x in (1, 2, 3, 4):  
    squares.append(x * x)  
# squares: [1, 4, 9, 16]
```

The expression applied to each element can be as complex as needed:

```
# Get a list of uppercase characters from a string  
[s.upper() for s in "Hello World"]  
# ['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']  
  
# Strip off any commas from the end of strings in a list  
[w.strip(',') for w in ['these,', 'words,,', 'mostly', 'have,commas,']]  
# ['these', 'words', 'mostly', 'have,commas']  
  
# Organize letters in words more reasonably - in an alphabetical order  
sentence = "Beautiful is better than ugly"  
["".join(sorted(word, key = lambda x: x.lower())) for word in sentence.split()]  
# ['aBefiltuu', 'is', 'beertt', 'ahnt', 'gluy']
```

else

[`else`](#) can be used in List comprehension constructs, but be careful regarding the syntax. The if/else clauses should

在 for 循环之前使用，而不是之后：

```
# 创建一个字符串'apple'中的字符列表，将非元音字母替换为'*'  
# 例如 - 'apple' --> ['a', '*', '*', '*', 'e']  
  
[x for x in 'apple' if x in 'aeiou' else '*']  
# 语法错误: invalid syntax  
  
# 当同时使用 if/else 时，应在循环之前使用  
[x if x in 'aeiou' else '*' for x in 'apple']  
#[a', '*', '*', '*', 'e']
```

注意，这里使用的是另一种语言结构，称为条件表达式，它本身不是推导式语法的一部分。而 for...in 之后的 if 是列表推导式的一部分，用于过滤源可迭代对象中的元素。

be used before **for** loop, not after:

```
# create a list of characters in apple, replacing non vowels with '*'  
# Ex - 'apple' --> ['a', '*', '*', '*', 'e']  
  
[x for x in 'apple' if x in 'aeiou' else '*']  
#SyntaxError: invalid syntax  
  
# When using if/else together use them before the loop  
[x if x in 'aeiou' else '*' for x in 'apple']  
#[a', '*', '*', '*', 'e']
```

Note this uses a different language construct, a [conditional expression](#), which itself is not part of the [comprehension syntax](#). Whereas the if after the **for...in** is a part of list comprehensions and used to *filter* elements from the source iterable.

## 双重迭代

双重迭代的顺序[... for x in ... for y in ...]可以是自然的也可以是反直觉的。经验法则是遵循等效的for循环：

```
def foo(i):  
    return i, i + 0.5  
  
for i in range(3):  
    for x in foo(i):  
        yield str(x)
```

这变成了：

```
[str(x)  
    for i in range(3)  
        for x in foo(i)  
]
```

这可以压缩成一行，即[str(x) for i in range(3) for x in foo(i)]

## 就地变异和其他副作用

在使用列表推导式之前，先了解因副作用调用的函数（变异或原地函数）通常返回None，与返回有意义值的函数之间的区别。

许多函数（尤其是纯函数）只是接受一个对象并返回某个对象。一个原地函数会修改现有对象，这称为副作用。其他例子包括输入输出操作，如打印。

`list.sort()` 会原地排序列表（意味着它修改原始列表）并返回值None。因此，它在列表推导式中不会按预期工作：

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]  
# [None, None, None]
```

相反，`sorted()` 返回一个排序后的列表，而不是原地排序：

## Double Iteration

Order of double iteration [... **for** x **in** ... **for** y **in** ...] is either natural or counter-intuitive. The rule of thumb is to follow an equivalent **for** loop:

```
def foo(i):  
    return i, i + 0.5  
  
for i in range(3):  
    for x in foo(i):  
        yield str(x)
```

This becomes:

```
[str(x)  
    for i in range(3)  
        for x in foo(i)  
]
```

This can be compressed into one line as [str(x) **for** i **in** range(3) **for** x **in** foo(i)]

## In-place Mutation and Other Side Effects

Before using list comprehension, understand the difference between functions called for their side effects (*mutating*, or [in-place](#) functions) which usually return [None](#), and functions that return an interesting value.

Many functions (especially [pure](#) functions) simply take an object and return some object. An *in-place* function modifies the existing object, which is called a *side effect*. Other examples include input and output operations such as printing.

`list.sort()` sorts a list *in-place* (meaning that it modifies the original list) and returns the value [None](#). Therefore, it won't work as expected in a list comprehension:

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]  
# [None, None, None]
```

Instead, `sorted()` returns a sorted [list](#) rather than sorting *in-place*:

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]  
# [[1, 2], [3, 4], [0, 1]]]
```

使用推导式来执行副作用是可能的，比如I/O或原地函数。但通常for循环更易读。在Python 3中，这样写是可行的：

```
[print(x) for x in (1, 2, 3)]
```

更好的写法是：

```
for x in (1, 2, 3):  
    print(x)
```

在某些情况下，带有副作用的函数适合用于列表推导式。`random.randrange()`具有改变随机数生成器状态的副作用，但它也返回一个有趣的值。此外，`next()`可以在迭代器上调用。

以下随机值生成器不是纯粹的，但由于每次计算表达式时都会重置随机生成器，因此是合理的：

```
from random import randrange  
[randrange(1, 7) for _ in range(10)]  
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

### 列表推导式中的空白符

更复杂的列表推导式可能会变得过长或可读性降低。虽然在示例中较少见，但可以将列表推导式拆分成多行，如下所示：

```
[  
    x for x  
        in 'foo'  
        if x not in 'bar'  
]
```

## 第21.2节：条件列表推导式

给定一个列表推导式，你可以附加一个或多个if条件来过滤值。

```
[<expression> for <element> in <iterable> if <condition>]
```

对于 `<element>` 在 `<iterable>` 中的每个元素；如果 `<condition>` 计算结果为 `True`，则将 `<expression>` (通常是 `<element>` 的函数) 添加到返回的列表中。

例如，这可以用来从整数序列中提取所有偶数：

```
[x for x in range(10) if x % 2 == 0]  
# 输出: [0, 2, 4, 6, 8]
```

### 实时演示

上述代码等价于：

```
even_numbers = []
```

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]  
# [[1, 2], [3, 4], [0, 1]]]
```

Using comprehensions for side-effects is possible, such as I/O or in-place functions. Yet a for loop is usually more readable. While this works in Python 3:

```
[print(x) for x in (1, 2, 3)]
```

Instead use:

```
for x in (1, 2, 3):  
    print(x)
```

In some situations, side effect functions *are* suitable for list comprehension. `random.randrange()` has the side effect of changing the state of the random number generator, but it also returns an interesting value. Additionally, `next()` can be called on an iterator.

The following random value generator is not pure, yet makes sense as the random generator is reset every time the expression is evaluated:

```
from random import randrange  
[randrange(1, 7) for _ in range(10)]  
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

### Whitespace in list comprehensions

More complicated list comprehensions can reach an undesired length, or become less readable. Although less common in examples, it is possible to break a list comprehension into multiple lines like so:

```
[  
    x for x  
        in 'foo'  
        if x not in 'bar'  
]
```

## Section 21.2: Conditional List Comprehensions

Given a [list comprehension](#) you can append one or more if conditions to filter values.

```
[<expression> for <element> in <iterable> if <condition>]
```

For each `<element> in <iterable>`; if `<condition>` evaluates to `True`, add `<expression>` (usually a function of `<element>`) to the returned list.

For example, this can be used to extract only even numbers from a sequence of integers:

```
[x for x in range(10) if x % 2 == 0]  
# Out: [0, 2, 4, 6, 8]
```

### Live demo

The above code is equivalent to:

```
even_numbers = []
```

```
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(even_numbers)
# 输出: [0, 2, 4, 6, 8]
```

此外，形式为[e **for** x **in** y **if** c]的条件列表推导式（其中e和c是关于x的表达式）等价于list(filter(lambda x: c, map(lambda x: e, y)))。

尽管结果相同，但请注意前一个示例的速度几乎是后一个的两倍。对于好奇的人来说，this 是对原因的一个很好的解释。

请注意，这与 ... **if** ... **else** ... 条件表达式（有时称为三元表达式）有很大不同，后者可以用于列表推导式的 <expression> 部分。考虑以下示例：

```
[x if x % 2 == 0 else None for x in range(10)]
# 输出: [0, None, 2, None, 4, None, 6, None, 8, None]
```

### 实时演示

这里的条件表达式不是过滤器，而是决定列表项所用值的运算符：

<value-if-condition-is-true> **if** <condition> **else** <value-if-condition-is-false>

如果将其与其他运算符结合使用，这一点会更加明显：

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in range(10)]
# 输出: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

### 实时演示

如果你使用的是 Python 2.7，`xrange` 可能比 `range` 更好，原因在 `xrange` 文档中有描述。

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in xrange(10)]
# 输出: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

上述代码等价于：

```
numbers = []
for x in range(10):
    if x % 2 == 0:
        temp = x
    else:
        temp = -1
    numbers.append(2 * temp + 1)
print(numbers)
# 输出: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

可以将三元表达式和if条件结合使用。三元运算符作用于过滤后的结果：

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# 输出: ['*', '*', 4, 6, 8]
```

```
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(even_numbers)
# Out: [0, 2, 4, 6, 8]
```

Also, a conditional list comprehension of the form [e **for** x **in** y **if** c] (where e and c are expressions in terms of x) is equivalent to list(filter(lambda x: c, map(lambda x: e, y))).

Despite providing the same result, pay attention to the fact that the former example is almost 2x faster than the latter one. For those who are curious, [this](#) is a nice explanation of the reason why.

Note that this is quite different from the ... **if** ... **else** ... conditional expression (sometimes known as a ternary expression) that you can use for the <expression> part of the list comprehension. Consider the following example:

```
[x if x % 2 == 0 else None for x in range(10)]
# Out: [0, None, 2, None, 4, None, 6, None, 8, None]
```

### Live demo

Here the conditional expression isn't a filter, but rather an operator determining the value to be used for the list items:

<value-if-condition-is-true> **if** <condition> **else** <value-if-condition-is-false>

This becomes more obvious if you combine it with other operators:

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in range(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

### Live demo

If you are using Python 2.7, `xrange` may be better than `range` for several reasons as described in the [xrange documentation](#).

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in xrange(10)]
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

The above code is equivalent to:

```
numbers = []
for x in range(10):
    if x % 2 == 0:
        temp = x
    else:
        temp = -1
    numbers.append(2 * temp + 1)
print(numbers)
# Out: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

One can combine ternary expressions and if conditions. The ternary operator works on the filtered result:

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Out: ['*', '*', 4, 6, 8]
```

仅用三元运算符无法实现相同效果：

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# 输出:['*', '*', '*', 4, '*', 6, '*', 8, '*']
```

另见：过滤器（Filters），它们通常是条件列表推导的充分替代方案。

## 第21.3节：使用 条件子句避免重复和昂贵的操作

考虑以下列表推导式：

```
>>> def f(x):
...     import time
...     time.sleep(.1)      # 模拟耗时函数
...     return x**2

>>> [f(x) for x in range(1000) if f(x) > 10]
[16, 25, 36, ...]
```

这导致对 1,000 个 x 值调用了两次 f(x)：一次用于生成值，另一次用于检查 if 条件。如果 f(x) 是一个特别耗时的操作，这可能会对性能产生显著影响。更糟的是，如果调用 f() 有副作用，可能会产生意想不到的结果。

相反，你应该通过生成一个中间可迭代对象（生成器表达式）来确保每个 x 值只计算一次耗时操作，方法如下：

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

或者，使用内置的 map 等价写法：

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

另一种可能使代码更易读的方法是将部分结果（例如前面例子中的v）放入一个可迭代对象（如列表或元组）中，然后对其进行迭代。由于v将是该可迭代对象中的唯一元素，结果就是我们现在拥有对慢速函数输出的引用，该输出只计算了一次：

```
>>> [对于 x 在范围(1000) 内的 v 在[f(x)] 中 如果 v > 10]
[16, 25, 36, ...]
```

然而，在实际操作中，代码的逻辑可能更加复杂，保持代码的可读性非常重要。通常，建议使用单独的生成器函数，而不是复杂的一行代码：

```
>>> def process_prime_numbers(iterable):
...     对于 x 在 iterable 中：
...         如果 是素数(x):
...             产量 f(x)
...
>>> [ x 用于 x 在 process_prime_numbers(范围(1000)) 中 如果 x > 10]
[11, 13, 17, 19, ...]
```

防止多次计算f(x)的另一种方法是对f(x)使用@functools.lru\_cache()（Python 3.2及以上版本）装饰器。这样，由于输入 x 对应的 f 的输出已经计算过一次，第二次

The same couldn't have been achieved just by ternary operator only:

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Out:['*', '*', '*', 4, '*', 6, '*', 8, '*']
```

See also: Filters, which often provide a sufficient alternative to conditional list comprehensions.

## Section 21.3: Avoid repetitive and expensive operations using conditional clause

Consider the below list comprehension:

```
>>> def f(x):
...     import time
...     time.sleep(.1)      # Simulate expensive function
...     return x**2

>>> [f(x) for x in range(1000) if f(x) > 10]
[16, 25, 36, ...]
```

This results in two calls to f(x) for 1,000 values of x: one call for generating the value and the other for checking the if condition. If f(x) is a particularly expensive operation, this can have significant performance implications. Worse, if calling f() has side effects, it can have surprising results.

Instead, you should evaluate the expensive operation only once for each value of x by generating an intermediate iterable (generator expression) as follows:

```
>>> [v for v in (f(x) for x in range(1000)) if v > 10]
[16, 25, 36, ...]
```

Or, using the builtin map equivalent:

```
>>> [v for v in map(f, range(1000)) if v > 10]
[16, 25, 36, ...]
```

Another way that could result in a more readable code is to put the partial result (v in the previous example) in an iterable (such as a list or a tuple) and then iterate over it. Since v will be the only element in the iterable, the result is that we now have a reference to the output of our slow function computed only once:

```
>>> [v for x in range(1000) for v in [f(x)] if v > 10]
[16, 25, 36, ...]
```

However, in practice, the logic of code can be more complicated and it's important to keep it readable. In general, a separate generator function is recommended over a complex one-liner:

```
>>> def process_prime_numbers(iterable):
...     for x in iterable:
...         if is_prime(x):
...             yield f(x)
...
>>> [x for x in process_prime_numbers(range(1000)) if x > 10]
[11, 13, 17, 19, ...]
```

Another way to prevent computing f(x) multiple times is to use the @functools.lru\_cache()(Python 3.2+) decorator on f(x). This way since the output of f for the input x has already been computed once, the second

原始列表推导式的函数调用速度将与字典查找一样快。这种方法使用[记忆化](#)来提高效率，其效果可与使用生成器表达式相媲美。

假设你需要将一个列表展平

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

一些方法可能是：

```
reduce(lambda x, y: x+y, l)  
sum(l, [])  
list(itertools.chain(*l))
```

然而，列表推导式会提供最佳的时间复杂度。

```
[item for sublist in l for item in sublist]
```

基于 `+` 的快捷方式（包括 `sum` 中的隐式使用）在有  $L$  个子列表时，时间复杂度必然是  $O(L^2)$ ——因为中间结果列表不断变长，每一步都会分配一个新的中间结果列表对象，且必须将之前中间结果中的所有元素复制过去（以及在末尾添加一些新元素）。所以（为简化且不失一般性）假设你有  $L$  个子列表，每个子列表有  $I$  个元素：第一个  $I$  个元素被复制了  $L-1$  次，第二个  $I$  个元素被复制了  $L-2$  次，依此类推；复制总次数是  $I$  乘以从 1 到  $L-1$  的和，即  $I * (L**2)/2$ 。

列表推导式只生成一个列表，一次性完成，并且每个元素只复制一次（从其原始位置到结果列表）。

## 第21.4节：字典推导式

字典推导式类似于列表推导式，只不过它生成的是字典对象而不是列表。

一个基本示例：

```
Python 2.x 版本 ≥ 2.7  
{x: x * x for x in (1, 2, 3, 4)}  
# 输出: {1: 1, 2: 4, 3: 9, 4: 16}
```

这只是另一种写法：

```
dict((x, x * x) for x in (1, 2, 3, 4))  
# 输出: {1: 1, 2: 4, 3: 9, 4: 16}
```

与列表推导式一样，我们可以在字典推导式中使用条件语句，只生成满足某些条件的字典元素。

```
Python 2.x 版本 ≥ 2.7  
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}  
# 输出: {'Exchange': 8, 'Overflow': 8}
```

或者，使用生成器表达式重写。

function invocation of the original list comprehension will be as fast as a dictionary lookup. This approach uses [memoization](#) to improve efficiency, which is comparable to using generator expressions.

Say you have to flatten a list

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

Some of the methods could be:

```
reduce(lambda x, y: x+y, l)  
sum(l, [])  
list(itertools.chain(*l))
```

However list comprehension would provide the best time complexity.

```
[item for sublist in l for item in sublist]
```

The shortcuts based on `+` (including the implied use in `sum`) are, of necessity,  $O(L^2)$  when there are  $L$  sublists -- as the intermediate result list keeps getting longer, at each step a new intermediate result list object gets allocated, and all the items in the previous intermediate result must be copied over (as well as a few new ones added at the end). So (for simplicity and without actual loss of generality) say you have  $L$  sublists of  $I$  items each: the first  $I$  items are copied back and forth  $L-1$  times, the second  $I$  items  $L-2$  times, and so on; total number of copies is  $I$  times the sum of  $x$  for  $x$  from 1 to  $L$  excluded, i.e.,  $I * (L**2)/2$ .

The list comprehension just generates one list, once, and copies each item over (from its original place of residence to the result list) also exactly once.

## Section 21.4: Dictionary Comprehensions

A [dictionary comprehension](#) is similar to a list comprehension except that it produces a dictionary object instead of a list.

A basic example:

```
Python 2.x 版本 ≥ 2.7  
{x: x * x for x in (1, 2, 3, 4)}  
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

which is just another way of writing:

```
dict((x, x * x) for x in (1, 2, 3, 4))  
# Out: {1: 1, 2: 4, 3: 9, 4: 16}
```

As with a list comprehension, we can use a conditional statement inside the dict comprehension to produce only the dict elements meeting some criterion.

```
Python 2.x 版本 ≥ 2.7  
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}  
# Out: {'Exchange': 8, 'Overflow': 8}
```

Or, rewritten using a generator expression.

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# 输出: {'Exchange': 8, 'Overflow': 8}
```

### 从一个字典开始，使用字典推导式作为键值对过滤器

Python 2.x 版本 ≥ 2.7

```
initial_dict = {'x': 1, 'y': 2}
{key: value for key, value in initial_dict.items() if key == 'x'}
# 输出: {'x': 1}
```

### 交换字典的键和值（字典反转）

如果你有一个包含简单可哈希值的字典（重复值可能导致意外结果）：

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

如果你想交换键和值，可以根据你的编码风格采取多种方法：

- swapped = {v: k for k, v in my\_dict.items()}
- swapped = dict((v, k) for k, v in my\_dict.iteritems())
- swapped = dict(zip(my\_dict.values(), my\_dict))
- swapped = dict(zip(my\_dict.values(), my\_dict.keys()))
- swapped = dict(map(reversed, my\_dict.items()))

```
print(swapped)
# 输出: {a: 1, b: 2, c: 3}
```

Python 2.x 版本 ≥ 2.3

如果你的字典很大，考虑导入`itertools`并使用`izip`或`imap`。

### 合并字典

合并字典，并可选择使用嵌套字典推导式覆盖旧值。

```
dict1 = {'w': 1, 'x': 1}
dict2 = {'x': 2, 'y': 2, 'z': 2}

{k: v for d in [dict1, dict2] for k, v in d.items()}
# 输出: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

然而，字典解包（PEP 448）可能是更优选。

Python 3.x 版本 ≥ 3.5

```
{**dict1, **dict2}
# 输出: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

**注意:** 字典推导式是在 Python 3.0 中添加的，并向 2.7+ 版本回移植，不同于列表推导式，列表推导式是在 2.0 中添加的。低于 2.7 的版本可以使用生成器表达式和内置的 `dict()` 来模拟字典推导式的行为。

## 第21.5节：带嵌套循环的列表推导式

列表推导式可以使用嵌套的 `for` 循环。你可以在列表推导式中编写任意数量的嵌套 `for` 循环，且每个 `for` 循环可以有一个可选的 `if` 条件。当这样做时，`for` 的顺序

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# Out: {'Exchange': 8, 'Overflow': 8}
```

### Starting with a dictionary and using dictionary comprehension as a key-value pair filter

Python 2.x 版本 ≥ 2.7

```
initial_dict = {'x': 1, 'y': 2}
{key: value for key, value in initial_dict.items() if key == 'x'}
# Out: {'x': 1}
```

### Switching key and value of dictionary (invert dictionary)

If you have a dict containing simple *hashable* values (duplicate values may have unexpected results):

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

and you wanted to swap the keys and values you can take several approaches depending on your coding style:

- swapped = {v: k for k, v in my\_dict.items()}
- swapped = dict((v, k) for k, v in my\_dict.iteritems())
- swapped = dict(zip(my\_dict.values(), my\_dict))
- swapped = dict(zip(my\_dict.values(), my\_dict.keys()))
- swapped = dict(map(reversed, my\_dict.items()))

```
print(swapped)
# Out: {a: 1, b: 2, c: 3}
```

Python 2.x 版本 ≥ 2.3

If your dictionary is large, consider *importing* `itertools` and utilize `izip` or `imap`.

### Merging Dictionaries

Combine dictionaries and optionally override old values with a nested dictionary comprehension.

```
dict1 = {'w': 1, 'x': 1}
dict2 = {'x': 2, 'y': 2, 'z': 2}

{k: v for d in [dict1, dict2] for k, v in d.items()}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

However, dictionary unpacking (PEP 448) may be a preferred.

```
Python 3.x 版本 ≥ 3.5
{**dict1, **dict2}
# Out: {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

**Note:** [dictionary comprehensions](#) were added in Python 3.0 and backported to 2.7+, unlike list comprehensions, which were added in 2.0. Versions < 2.7 can use generator expressions and the `dict()` builtin to simulate the behavior of dictionary comprehensions.

## Section 21.5: List Comprehensions with Nested Loops

[List Comprehensions](#) can use nested `for` loops. You can code any number of nested for loops within a list comprehension, and each `for` loop may have an optional associated `if` test. When doing so, the order of the `for`

构造的顺序与编写一系列嵌套的for语句时相同。列表推导式的一般结构如下：

```
[ 表达式 for 目标1 in 可迭代对象1 [if 条件1]
    for 目标2 in 可迭代对象2 [if 条件2]...
    for 目标N in 可迭代对象N [if 条件N] ]
```

例如，以下代码使用多个for语句将列表的列表展平：

```
data = [[1, 2], [3, 4], [5, 6]]
output = []
for each_list in data:
    for element in each_list:
        output.append(element)
print(output)
# 输出: [1, 2, 3, 4, 5, 6]
```

可以等价地写成带有多个for结构的列表推导式：

```
data = [[1, 2], [3, 4], [5, 6]]
output = [element for each_list in data for element in each_list]
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

## 实时演示

在展开形式和列表推导式中，外层循环（第一个for语句）都是先出现的。

除了更简洁之外，嵌套推导式的速度也明显更快。

```
In [1]: data = [[1, 2], [3, 4], [5, 6]]
In [2]: def f():
...:     output=[]
...:     for each_list in data:
...:         for element in each_list:
...:             output.append(element)
...:     return output
在 [3]: timeit f()
1000000 循环, 3次中最佳: 每次循环 1.37 微秒
在 [4]: timeit [内层 for 外层 中 数据 for 内层 中 外层]
1000000 循环, 3次中最佳: 每次循环 632 纳秒
```

上述函数调用的开销约为140纳秒。

内联if语句以类似方式嵌套，且可出现在第一个for之后的任意位置：

```
数据 = [[1], [2, 3], [4, 5]]
输出 = [元素 for 每个列表 在 数据中
        如果 len(每个列表) == 2
        对于 元素 在 每个列表中
        如果 元素 != 5]
打印(输出)
# 输出: [2, 3, 4]
```

## 实时演示

为了提高可读性，您应该考虑使用传统的for循环。尤其是在嵌套超过2层，和/或推导式的逻辑过于复杂时更是如此。多重嵌套循环列表

constructs is the same order as when writing a series of nested `for` statements. The general structure of list comprehensions looks like this:

```
[ expression for target1 in iterable1 [if condition1]
    for target2 in iterable2 [if condition2]...
    for targetN in iterableN [if conditionN] ]
```

For example, the following code flattening a list of lists using multiple `for` statements:

```
data = [[1, 2], [3, 4], [5, 6]]
output = []
for each_list in data:
    for element in each_list:
        output.append(element)
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

can be equivalently written as a list comprehension with multiple `for` constructs:

```
data = [[1, 2], [3, 4], [5, 6]]
output = [element for each_list in data for element in each_list]
print(output)
# Out: [1, 2, 3, 4, 5, 6]
```

## Live Demo

In both the expanded form and the list comprehension, the outer loop (first for statement) comes first.

In addition to being more compact, the nested comprehension is also significantly faster.

```
In [1]: data = [[1, 2], [3, 4], [5, 6]]
In [2]: def f():
...:     output=[]
...:     for each_list in data:
...:         for element in each_list:
...:             output.append(element)
...:     return output
In [3]: timeit f()
1000000 loops, best of 3: 1.37 µs per loop
In [4]: timeit [inner for outer in data for inner in outer]
1000000 loops, best of 3: 632 ns per loop
```

The overhead for the function call above is about 140ns.

Inline ifs are nested similarly, and may occur in any position after the first `for`:

```
data = [[1, 2, 3], [4, 5]]
output = [element for each_list in data
        if len(each_list) == 2
        for element in each_list
        if element != 5]
print(output)
# Out: [2, 3, 4]
```

## Live Demo

For the sake of readability, however, you should consider using traditional `for-loops`. This is especially true when nesting is more than 2 levels deep, and/or the logic of the comprehension is too complex. multiple nested loop list

推导式可能容易出错或产生意外结果。

## 第21.6节：生成器表达式

生成器表达式与列表推导式非常相似。主要区别在于它不会一次性创建完整的结果集；而是创建一个生成器对象，之后可以对其进行迭代。

例如，看看下面代码中的区别：

```
# 列表推导式  
[x**2 for x in range(10)]  
# 输出: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
  
Python 2.x 版本 ≥ 2.4  
  
# 生成器推导式  
(x**2 for x in xrange(10))  
# 输出: <generator object <genexpr> at 0x11b4b7c80>
```

这两个是非常不同的物体：

- 列表推导式返回一个列表对象，而生成器推导式返回一个生成器。
- 生成器对象不能被索引，并且使用next函数按顺序获取元素。

**注意：**我们使用 `xrange`，因为它也创建一个生成器对象。如果使用`range`，则会创建一个列表。此外，`xrange`仅存在于Python 2的后期版本中。在Python 3中，`range`直接返回一个生成器。更多信息，请参见`range`和`xrange`函数的区别示例。

```
Python 2.x 版本 ≥ 2.4  
  
g = (x**2 for x in xrange(10))  
print(g[0])  
  
回溯（最近一次调用最后）：  
文件 "<stdin>", 第 1 行, 在 <module>  
TypeError: 'generator' 对象没有属性 '__getitem__'
```

```
g.next() # 0  
g.next() # 1  
g.next() # 4  
...  
g.next() # 81  
  
g.next() # 抛出 StopIteration 异常  
  
Traceback (most recent call last):  
文件 "<stdin>", 第 1 行, 在 <模块>  
StopIteration  
  
Python 3.x 版本 ≥ 3.0
```

**注意：**函数 `g.next()` 应替换为 `next(g)`，`xrange` 应替换为 `range`，因为 `Iterator.next()` 和 `xrange()` 在 Python 3 中不存在。

虽然这两者都可以用类似的方式迭代：

comprehension could be error prone or it gives unexpected result.

## Section 21.6: Generator Expressions

Generator expressions are very similar to list comprehensions. The main difference is that it does not create a full set of results at once; it creates a generator object which can then be iterated over.

For instance, see the difference in the following code:

```
# list comprehension  
[x**2 for x in range(10)]  
# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
  
Python 2.x Version ≥ 2.4  
  
# generator comprehension  
(x**2 for x in xrange(10))  
# Output: <generator object <genexpr> at 0x11b4b7c80>
```

These are two very different objects:

- the list comprehension returns a `list` object whereas the generator comprehension returns a generator.
- generator objects cannot be indexed and makes use of the `next` function to get items in order.

**Note:** We use `xrange` since it too creates a generator object. If we would use `range`, a list would be created. Also, `xrange` exists only in later version of python 2. In python 3, `range` just returns a generator. For more information, see the *Differences between range and xrange functions* example.

```
Python 2.x Version ≥ 2.4  
  
g = (x**2 for x in xrange(10))  
print(g[0])  
  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: 'generator' object has no attribute '__getitem__'
```

```
g.next() # 0  
g.next() # 1  
g.next() # 4  
...  
g.next() # 81  
  
g.next() # Throws StopIteration Exception  
  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
StopIteration  
  
Python 3.x Version ≥ 3.0
```

**NOTE:** The function `g.next()` should be substituted by `next(g)` and `xrange` with `range` since `Iterator.next()` and `xrange()` do not exist in Python 3.

Although both of these can be iterated in a similar way:

```
for i in [x**2 for x in range(10)]:  
    print(i)  
  
'''  
输出：  
0  
1  
4  
.  
8  
1  
4  
.  
*  
py  
thon  
2  
x  
版本      ≥ 2.4
```

```
for i in (x**2 for x in xrange(10)):  
    print(i)  
  
'''
```

输出：

```
for i in [x**2 for x in range(10)]:  
    print(i)
```

```
'''  
Out:  
0  
1  
4  
.  
81  
'''
```

Python 2.x Version ≥ 2.4

```
for i in (x**2 for x in xrange(10)):  
    print(i)
```

```
'''  
Out:  
0  
1  
4  
.  
.  
.  
81  
'''
```

## Use cases

Generator expressions are lazily evaluated, which means that they generate and return each value only when the generator is iterated. This is often useful when iterating through large datasets, avoiding the need to create a duplicate of the dataset in memory:

```
for square in (x**2 for x in range(1000000)):  
    #do something
```

Another common use case is to avoid iterating over an entire iterable if doing so is not necessary. In this example, an item is retrieved from a remote API with each iteration of `get_objects()`. Thousands of objects may exist, must be retrieved one-by-one, and we only need to know if an object matching a pattern exists. By using a generator expression, when we encounter an object matching the pattern,

```
def get_objects():  
    """Gets objects from an API one by one"""  
    while True:  
        yield get_next_item()  
  
def object_matches_pattern(obj):  
    # perform potentially complex calculation  
    return matches_pattern  
  
def right_item_exists():  
    items = (object_matched_pattern(each) for each in get_objects())  
    for item in items:  
        if item.is_the_right_one:  
  
            return True  
    return False
```

## 第21.7节：集合推导式

集合推导式类似于列表和字典推导式，但它生成一个[集合](#)，这是一个无序的唯一元素集合。

Python 2.x 版本 ≥ 2.7

```
# 一个包含 range(5) 中所有值的集合：  
{x for x in range(5)}  
# 输出: {0, 1, 2, 3, 4}  
  
# 1 到 10 之间的偶数集合：  
{x for x in range(1, 11) if x % 2 == 0}  
# 输出: {2, 4, 6, 8, 10}  
  
# 字符串中唯一的字母字符：  
text = "When in the Course of human events it becomes necessary for one people..."  
{ch.lower() for ch in text if ch.isalpha()}  
# 输出: set(['a', 'c', 'b', 'e', 'f', 'i', 'h', 'm', 'l', 'o',  
#           'n', 'p', 's', 'r', 'u', 't', 'w', 'v', 'y'])
```

[实时演示](#)

请记住，集合是无序的。这意味着集合中结果的顺序可能与上述示例中显示的顺序不同。

**注意：**集合推导式自 Python 2.7+ 起可用，不同于列表推导式，它是在 2.0 版本中添加的。在 Python 2.2 到 Python 2.6 中，可以使用[set\(\)](#)函数配合生成器表达式来产生相同的结果：

Python 2.x 版本 ≥ 2.2

```
set(x for x in range(5))  
# 输出: {0, 1, 2, 3, 4}
```

## 第21.8节：将filter和map重构为列表推导式

filter或map函数通常应被列表推导式替代。Guido Van Rossum在2005年的一封公开信中对此有很好的描述：

filter( $P, S$ )几乎总是可以更清晰地写成 $[x \text{ for } x \text{ in } S \text{ if } P(x)]$ ，这有一个巨大的优势，即最常见的用法涉及的是比较谓词，例如 $x==42$ ，定义一个lambda函数对读者来说需要更多的努力（而且lambda比列表推导式更慢）。对于 $\text{map}(F, S)$ 更是如此，它变成了 $[F(x) \text{ for } x \text{ in } S]$ 。当然，在许多情况下你也可以使用生成器表达式。

以下代码行被认为是“不符合Python风格”的，并且会在许多Python代码检查工具中报错。

```
filter(lambda x: x % 2 == 0, range(10)) # 小于10的偶数  
map(lambda x: 2*x, range(10)) # 每个数字乘以二  
reduce(lambda x,y: x+y, range(10)) # 列表中所有元素的和
```

结合前面引用的内容，我们可以将这些filter和map表达式拆解成等价的列表推导式；同时去除每个表达式中的lambda函数——使代码更易读。

## Section 21.7: Set Comprehensions

Set comprehension is similar to list and dictionary comprehension, but it produces a [set](#), which is an unordered collection of unique elements.

Python 2.x 版本 ≥ 2.7

```
# A set containing every value in range(5):  
{x for x in range(5)}  
# Out: {0, 1, 2, 3, 4}  
  
# A set of even numbers between 1 and 10:  
{x for x in range(1, 11) if x % 2 == 0}  
# Out: {2, 4, 6, 8, 10}  
  
# Unique alphabetic characters in a string of text:  
text = "When in the Course of human events it becomes necessary for one people..."  
{ch.lower() for ch in text if ch.isalpha()}  
# Out: set(['a', 'c', 'b', 'e', 'f', 'i', 'h', 'm', 'l', 'o',  
#           'n', 'p', 's', 'r', 'u', 't', 'w', 'v', 'y'])
```

[Live Demo](#)

Keep in mind that sets are unordered. This means that the order of the results in the set may differ from the one presented in the above examples.

**Note:** Set comprehension is available since python 2.7+, unlike list comprehensions, which were added in 2.0. In Python 2.2 to Python 2.6, the [set\(\)](#) function can be used with a generator expression to produce the same result:

Python 2.x 版本 ≥ 2.2

```
set(x for x in range(5))  
# Out: {0, 1, 2, 3, 4}
```

## Section 21.8: Refactoring filter and map to list comprehensions

The [filter](#) or [map](#) functions should often be replaced by [list comprehensions](#). Guido Van Rossum describes this well in an [open letter in 2005](#):

`filter( $P, S$ )` is almost always written clearer as  $[x \text{ for } x \text{ in } S \text{ if } P(x)]$ , and this has the huge advantage that the most common usages involve predicates that are comparisons, e.g.  $x==42$ , and defining a lambda for that just requires much more effort for the reader (plus the lambda is slower than the list comprehension). Even more so for `map( $F, S$ )` which becomes  $[F(x) \text{ for } x \text{ in } S]$ . Of course, in many cases you'd be able to use generator expressions instead.

The following lines of code are considered "not pythonic" and will raise errors in many python linters.

```
filter(lambda x: x % 2 == 0, range(10)) # even numbers < 10  
map(lambda x: 2*x, range(10)) # multiply each number by two  
reduce(lambda x,y: x+y, range(10)) # sum of all elements in list
```

Taking what we have learned from the previous quote, we can break down these `filter` and `map` expressions into their equivalent [list comprehensions](#); also removing the `lambda` functions from each - making the code more readable in the process.

```
# 过滤：  
# P(x) = x % 2 == 0  
# S = range(10)  
[x for x in range(10) if x % 2 == 0]
```

```
# 映射  
# F(x) = 2*x  
# S = range(10)  
[2*x 对于 x 在 range(10)]
```

在处理链式函数时，可读性变得更加明显。由于可读性的原因，一个 map 或 filter 函数的结果应作为下一个函数的输入；在简单情况下，这些可以用单个列表推导式替代。此外，我们可以很容易地从列表推导式中看出处理的结果，而在推理链式的 Map 和 Filter 过程时，认知负担更重。

```
# Map & Filter  
filtered = filter(lambda x: x % 2 == 0, range(10))  
results = map(lambda x: 2*x, filtered)  
  
# 列表推导式  
results = [2*x for x in range(10) if x % 2 == 0]
```

## 重构 - 快速参考

- Map

```
map(F, S) == [F(x) for x in S]
```

- Filter

```
filter(P, S) == [x for x in S if P(x)]
```

其中  $F$  和  $P$  分别是转换输入值和返回布尔值的函数

## 第21.9节：涉及元组的推导式

列表推导式的for子句可以指定多个变量：

```
[x + y for x, y in [(1, 2), (3, 4), (5, 6)]]  
# 输出: [3, 7, 11]
```

```
[x + y for x, y in zip([1, 3, 5], [2, 4, 6])]  
# 输出: [3, 7, 11]
```

这就像普通的for循环：

```
for x, y in [(1,2), (3,4), (5,6)]:  
    print(x+y)  
# 3  
# 7  
# 11
```

但请注意，如果推导式开头的表达式是一个元组，则必须加括号：

```
[x, y 对于 x, y 属于[(1, 2), (3, 4), (5, 6)]]
```

```
# Filter:  
# P(x) = x % 2 == 0  
# S = range(10)  
[x for x in range(10) if x % 2 == 0]  
  
# Map  
# F(x) = 2*x  
# S = range(10)  
[2*x for x in range(10)]
```

Readability becomes even more apparent when dealing with chaining functions. Where due to readability, the results of one map or filter function should be passed as a result to the next; with simple cases, these can be replaced with a single list comprehension. Further, we can easily tell from the list comprehension what the outcome of our process is, where there is more cognitive load when reasoning about the chained Map & Filter process.

```
# Map & Filter  
filtered = filter(lambda x: x % 2 == 0, range(10))  
results = map(lambda x: 2*x, filtered)  
  
# List comprehension  
results = [2*x for x in range(10) if x % 2 == 0]
```

## Refactoring - Quick Reference

- Map

```
map(F, S) == [F(x) for x in S]
```

- Filter

```
filter(P, S) == [x for x in S if P(x)]
```

where  $F$  and  $P$  are functions which respectively transform input values and return a bool

## Section 21.9: Comprehensions involving tuples

The for clause of a [list comprehension](#) can specify more than one variable:

```
[x + y for x, y in [(1, 2), (3, 4), (5, 6)]]  
# Out: [3, 7, 11]
```

```
[x + y for x, y in zip([1, 3, 5], [2, 4, 6])]  
# Out: [3, 7, 11]
```

This is just like regular for loops:

```
for x, y in [(1,2), (3,4), (5,6)]:  
    print(x+y)  
# 3  
# 7  
# 11
```

Note however, if the expression that begins the comprehension is a tuple then it must be parenthesized:

```
[x, y for x, y in [(1, 2), (3, 4), (5, 6)]]
```

```
# 语法错误：无效的语法
```

```
[(x, y) for x, y in [(1, 2), (3, 4), (5, 6)]]
# 输出: [(1, 2), (3, 4), (5, 6)]
```

## 第21.10节：使用推导式计数出现次数

当我们想要计算可迭代对象中满足某个条件的元素数量时，可以使用推导式来生成一种惯用语法：

```
# 计算 `range(1000)` 中既是偶数又包含数字 '9' 的数字个数：
print (sum(
    1 for x in range(1000)
    if x % 2 == 0 and
    '9' in str(x)
))
# 输出: 95
```

基本概念可以总结为：

1. 遍历范围为1000的元素。
2. 连接所有需要的if条件。
3. 使用1作为表达式，为每个满足条件的项返回1。
4. 将所有的1相加，以确定满足条件的项的数量。

注意：这里我们不是将1收集到列表中（注意没有方括号），而是直接将这些1传递给sum函数进行求和。这称为生成器表达式，类似于推导式。

## 第21.11节：列表中类型的转换

定量数据通常以字符串形式读取，必须在处理前转换为数值类型。所有列表项的类型可以通过列表推导式或map()函数进行转换。

```
# 将字符串列表转换为整数列表。
items = ["1", "2", "3", "4"]
[int(item) for item in items]
# 输出: [1, 2, 3, 4]
```

```
# 将字符串列表转换为浮点数列表。
items = ["1", "2", "3", "4"]
map(float, items)
# Out:[1.0, 2.0, 3.0, 4.0]
```

## 第21.12节：嵌套列表推导式

嵌套列表推导式，不同于带有嵌套循环的列表推导式，是列表推导式中的列表推导式。初始表达式可以是任意表达式，包括另一个列表推导式。

```
# 带嵌套循环的列表推导式
[x + y for x in [1, 2, 3] for y in [3, 4, 5]]
#输出: [4, 5, 6, 5, 6, 7, 6, 7, 8]
```

```
# 嵌套列表推导式
[[x + y for x in [1, 2, 3]] for y in [3, 4, 5]]
#输出: [[4, 5, 6], [5, 6, 7], [6, 7, 8]]
```

```
# SyntaxError: invalid syntax
```

```
[(x, y) for x, y in [(1, 2), (3, 4), (5, 6)]]
# Out: [(1, 2), (3, 4), (5, 6)]
```

## Section 21.10: Counting Occurrences Using Comprehension

When we want to count the number of items in an iterable, that meet some condition, we can use comprehension to produce an idiomatic syntax:

```
# Count the numbers in `range(1000)` that are even and contain the digit '9':
print (sum(
    1 for x in range(1000)
    if x % 2 == 0 and
    '9' in str(x)
))
# Out: 95
```

The basic concept can be summarized as:

1. Iterate over the elements in `range(1000)`.
2. Concatenate all the needed `if` conditions.
3. Use 1 as *expression* to return a 1 for each item that meets the conditions.
4. Sum up all the 1s to determine number of items that meet the conditions.

**Note:** Here we are not collecting the 1s in a list (note the absence of square brackets), but we are passing the ones directly to the `sum` function that is summing them up. This is called a *generator expression*, which is similar to a Comprehension.

## Section 21.11: Changing Types in a List

Quantitative data is often read in as strings that must be converted to numeric types before processing. The types of all list items can be converted with either a List Comprehension or the `map()` function.

```
# Convert a list of strings to integers.
items = ["1", "2", "3", "4"]
[int(item) for item in items]
# Out: [1, 2, 3, 4]
```

```
# Convert a list of strings to float.
items = ["1", "2", "3", "4"]
map(float, items)
# Out:[1.0, 2.0, 3.0, 4.0]
```

## Section 21.12: Nested List Comprehensions

Nested list comprehensions, unlike list comprehensions with nested loops, are List comprehensions within a list comprehension. The initial expression can be any arbitrary expression, including another list comprehension.

```
#List Comprehension with nested loop
[x + y for x in [1, 2, 3] for y in [3, 4, 5]]
#Out: [4, 5, 6, 5, 6, 7, 6, 7, 8]
```

```
#Nested List Comprehension
[[x + y for x in [1, 2, 3]] for y in [3, 4, 5]]
#Out: [[4, 5, 6], [5, 6, 7], [6, 7, 8]]
```

该嵌套示例等价于

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

嵌套推导式的一个例子是用来转置矩阵。

```
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]

[[[row[i] for row in matrix] for i in range(len(matrix))]]
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

像嵌套的for循环一样，推导式的嵌套深度没有限制。

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# 输出: [[[1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

## 第21.13节：在列表推导式中同时迭代两个或多个列表

在列表推导式中同时迭代两个以上的列表，可以使用zip()，示例如下：

```
>>> list_1 = [1, 2, 3, 4]
>>> list_2 = ['a', 'b', 'c', 'd']
>>> list_3 = ['6', '7', '8', '9']

# 两个列表
>>> [(i, j) for i, j in zip(list_1, list_2)]
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# 三个列表
>>> [(i, j, k) for i, j, k in zip(list_1, list_2, list_3)]
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]

# 依此类推...
```

The Nested example is equivalent to

```
l = []
for y in [3, 4, 5]:
    temp = []
    for x in [1, 2, 3]:
        temp.append(x + y)
    l.append(temp)
```

One example where a nested comprehension can be used it to transpose a matrix.

```
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]

[[[row[i] for row in matrix] for i in range(len(matrix))]]
# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Like nested for loops, there is no limit to how deep comprehensions can be nested.

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Out: [[[1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

## Section 21.13: Iterate two or more list simultaneously within list comprehension

For iterating more than two lists simultaneously within list comprehension, one may use [zip\(\)](#) as:

```
>>> list_1 = [1, 2, 3, 4]
>>> list_2 = ['a', 'b', 'c', 'd']
>>> list_3 = ['6', '7', '8', '9']

# Two lists
>>> [(i, j) for i, j in zip(list_1, list_2)]
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# Three lists
>>> [(i, j, k) for i, j, k in zip(list_1, list_2, list_3)]
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]

# so on ...
```

# 第22章：列表切片（选择列表的部分内容）

## 第22.1节：使用第三个“步长”参数

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
lst[::2]
# 输出: ['a', 'c', 'e', 'g']

lst[::3]
# 输出: ['a', 'd', 'g']
```

## 第22.2节：从列表中选择子列表

```
lst = ['a', 'b', 'c', 'd', 'e']
lst[2:4]
# 输出: ['c', 'd']

lst[2:]
# 输出: ['c', 'd', 'e']

lst[:4]
# 输出: ['a', 'b', 'c', 'd']
```

## 第22.3节：使用切片反转列表

```
a = [1, 2, 3, 4, 5]
# 以步长-1反向遍历列表
b = a[::-1]

# 内置列表方法反转'a'
a.reverse()

if a == b:
    print(True)

print(b)

# 输出:
# 真
# [5, 4, 3, 2, 1]
```

## 第22.4节：使用切片移动列表

```
def shift_list(array, s):
    将列表中的元素向左或向右移动。

参数：
array - 要移动的列表
s - 移动列表的数量 ('+': 右移, '-': 左移)

返回：
shifted_array - 移动后的列表
```

# Chapter 22: List slicing (selecting parts of lists)

## Section 22.1: Using the third "step" argument

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
lst[::2]
# Output: ['a', 'c', 'e', 'g']

lst[::3]
# Output: ['a', 'd', 'g']
```

## Section 22.2: Selecting a sublist from a list

```
lst = ['a', 'b', 'c', 'd', 'e']
lst[2:4]
# Output: ['c', 'd']

lst[2:]
# Output: ['c', 'd', 'e']

lst[:4]
# Output: ['a', 'b', 'c', 'd']
```

## Section 22.3: Reversing a list with slicing

```
a = [1, 2, 3, 4, 5]
# steps through the list backwards (step=-1)
b = a[::-1]

# built-in list method to reverse 'a'
a.reverse()

if a == b:
    print(True)

print(b)

# Output:
# True
# [5, 4, 3, 2, 1]
```

## Section 22.4: Shifting a list using slicing

```
def shift_list(array, s):
    """Shifts the elements of a list to the left or right.

Args:
    array - the list to shift
    s - the amount to shift the list ('+': right-shift, '-': left-shift)

Returns:
    shifted_array - the shifted list
```

```
    """  
    # 计算实际的移动量 (例如, 如果数组长度为5, 11 -> 1)  
    s %= len(array)
```

```
    # 反转移动方向以使其更直观  
    s *= -1
```

```
    # 使用列表切片移动数组  
shifted_array = array[s:] + array[:s]
```

```
return shifted_array
```

```
my_array = [1, 2, 3, 4, 5]
```

```
# 负数  
shift_list(my_array, -7)  
>>> [3, 4, 5, 1, 2]
```

```
# 数组大小相等时不进行移动  
shift_list(my_array, 5)  
>>> [1, 2, 3, 4, 5]
```

```
# 适用于正数  
shift_list(my_array, 3)  
>>> [3, 4, 5, 1, 2]
```

```
    """  
    # calculate actual shift amount (e.g., 11 -> 1 if length of the array is 5)  
    s %= len(array)
```

```
    # reverse the shift direction to be more intuitive  
    s *= -1
```

```
    # shift array with list slicing  
shifted_array = array[s:] + array[:s]
```

```
return shifted_array
```

```
my_array = [1, 2, 3, 4, 5]
```

```
# negative numbers  
shift_list(my_array, -7)  
>>> [3, 4, 5, 1, 2]
```

```
# no shift on numbers equal to the size of the array  
shift_list(my_array, 5)  
>>> [1, 2, 3, 4, 5]
```

```
# works on positive numbers  
shift_list(my_array, 3)  
>>> [3, 4, 5, 1, 2]
```

# 第23章：groupby()

参数	详情
可迭代对象	任何Python可迭代对象
键	用于对可迭代对象进行分组的函数（条件）

在 Python 中，`itertools.groupby()` 方法允许开发者根据指定的属性将可迭代类的值分组为另一个可迭代的值集合。

## 第23.1节：示例4

在本例中，我们将看到使用不同类型的可迭代对象时会发生什么。

```
things = [("动物", "熊"), ("动物", "鸭子"), ("植物", "仙人掌"), ("交通工具", "哈雷"), \
          ("交通工具", "快艇"), ("交通工具", "校车")]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

结果为

```
{'动物': [('动物', '熊'), ('动物', '鸭')], \
  '植物': [('植物', '仙人掌')], \
  'vehicle': [('vehicle', '哈雷'), \
               ('vehicle', '快艇'), \
               ('vehicle', '校车')]}}
```

下面的这个例子本质上与上面的例子相同。唯一的区别是我将所有的元组改成了列表。

```
things = [[("动物", "熊"), ["动物", "鸭子"], ["车辆", "哈雷"], ["植物", "仙人掌"], \
           ["车辆", "快艇"], ["车辆", "校车"]]]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

结果

```
{'动物': [[('动物', '熊'), ['动物', '鸭子']]], \
  '植物': [[('植物', '仙人掌')]], \
  '车辆': [[('车辆', '哈雷'), \
             ['车辆', '快艇'], \
             ['车辆', '校车']]}}
```

## 第23.2节：示例2

这个例子说明了如果我们不指定任何内容，默认键是如何被选择的

```
c = groupby(['goat', 'dog', 'cow', 1, 1, 2, 3, 11, 10, ('persons', 'man', 'woman')])
dic = {}
for k, v in c:
```

# Chapter 23: groupby()

Parameter	Details
iterable	Any python iterable
key	Function(criteria) on which to group the iterable

In Python, the `itertools.groupby()` method allows developers to group values of an iterable class based on a specified property into another iterable set of values.

## Section 23.1: Example 4

In this example we see what happens when we use different types of iterable.

```
things = [("animal", "bear"), ("animal", "duck"), ("plant", "cactus"), ("vehicle", "harley"), \
          ("vehicle", "speed boat"), ("vehicle", "school bus")]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Results in

```
{'animal': [('animal', 'bear'), ('animal', 'duck')], \
  'plant': [('plant', 'cactus')], \
  'vehicle': [('vehicle', 'harley'), \
               ('vehicle', 'speed boat'), \
               ('vehicle', 'school bus')]}
```

This example below is essentially the same as the one above it. The only difference is that I have changed all the tuples to lists.

```
things = [[("animal", "bear"), ["animal", "duck"], ["vehicle", "harley"], ["plant", "cactus"], \
           ["vehicle", "speed boat"], ["vehicle", "school bus"]]]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Results

```
{'animal': [[('animal', 'bear'), ['animal', 'duck']]], \
  'plant': [[('plant', 'cactus')]], \
  'vehicle': [[('vehicle', 'harley'), \
               ('vehicle', 'speed boat'), \
               ('vehicle', 'school bus')]]}
```

## Section 23.2: Example 2

This example illustrates how the default key is chosen if we do not specify any

```
c = groupby(['goat', 'dog', 'cow', 1, 1, 2, 3, 11, 10, ('persons', 'man', 'woman')])
dic = {}
for k, v in c:
```

```
dic[k] = list(v)
dic
```

结果为

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

请注意，这个元组整体作为此列表中的一个键

## 第23.3节：示例3

注意在此示例中，mulato和camel没有出现在我们的结果中。只有最后一个具有指定键的元素会显示。对于c的最后一个结果实际上覆盖了之前的两个结果。但请看我将数据先按相同键排序后的新版本。

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
c = groupby(list_things, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

结果为

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

排序版本

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
sorted_list = sorted(list_things, key = lambda x: x[0])
print(sorted_list)
print()
c = groupby(sorted_list, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

结果为

```
['牛', '猫', '骆驼', '狗', '驴', '山羊', '混血儿', '猫鼬', '马鲁', ('人', '男人', '女人'), '袋熊']
```

```
dic[k] = list(v)
dic
```

Results in

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

Notice here that the tuple as a whole counts as one key in this list

## Section 23.3: Example 3

Notice in this example that mulato and camel don't show up in our result. Only the last element with the specified key shows up. The last result for c actually wipes out two previous results. But watch the new version where I have the data sorted first on same key.

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
c = groupby(list_things, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Results in

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Sorted Version

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
sorted_list = sorted(list_things, key = lambda x: x[0])
print(sorted_list)
print()
c = groupby(sorted_list, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Results in

```
['cow', 'cat', 'camel', 'dog', 'donkey', 'goat', 'mulato', 'mongoose', 'malloo', ('persons', 'man', 'woman'), 'wombat']
```

```
{'c': ['牛', '猫', '骆驼'],
'd': ['狗', '驴'],
'g': ['山羊'],
'm': ['混血儿', '猫鼬', '马鲁'],
'persons': [('人', '男人', '女人')],
'w': ['袋熊']}
```

```
{'c': ['cow', 'cat', 'camel'],
'd': ['dog', 'donkey'],
'g': ['goat'],
'm': ['mulato', 'mongoose', 'malloo'],
'persons': [('persons', 'man', 'woman')],
'w': ['wombat']}
```

# 第24章：链表

链表是一组节点的集合，每个节点由一个引用和一个值组成。节点通过它们的引用串联成一个序列。链表可以用来实现更复杂的数据结构，如列表、栈、队列和关联数组。

## 第24.1节：单链表示例

本例实现了一个链表，具有许多与内置列表对象相同的方法。

```
类 Node:
    def __init__(self, val):
        self.data = val
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, val):
        self.data = val

    def setNext(self, val):
        self.next = val

class LinkedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        """检查列表是否为空"""
        return self.head is None

    def add(self, item):
        """向列表中添加元素"""
        new_node = Node(item)
        new_node.setNext(self.head)
        self.head = new_node

    def size(self):
        """返回列表的长度/大小"""
        count = 0
        current = self.head
        while current is not None:
            count += 1
            current = current.getNext()
        return count

    def search(self, item):
        """在列表中搜索项目。如果找到，返回True。如果未找到，返回False"""
        current = self.head
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()

        return found
```

# Chapter 24: Linked lists

A linked list is a collection of nodes, each made up of a reference and a value. Nodes are strung together into a sequence using their references. Linked lists can be used to implement more complex data structures like lists, stacks, queues, and associative arrays.

## Section 24.1: Single linked list example

This example implements a linked list with many of the same methods as that of the built-in list object.

```
class Node:
    def __init__(self, val):
        self.data = val
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, val):
        self.data = val

    def setNext(self, val):
        self.next = val

class LinkedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        """Check if the list is empty"""
        return self.head is None

    def add(self, item):
        """Add the item to the list"""
        new_node = Node(item)
        new_node.setNext(self.head)
        self.head = new_node

    def size(self):
        """Return the length/size of the list"""
        count = 0
        current = self.head
        while current is not None:
            count += 1
            current = current.getNext()
        return count

    def search(self, item):
        """Search for item in list. If found, return True. If not found, return False"""
        current = self.head
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()

        return found
```

```

    return found

def remove(self, item):
    """从列表中移除项目。如果列表中未找到该项目，则抛出ValueError异常"""
    current = self.head
previous = None
found = False
    while current is not None and not found:
        if current.getData() is item:
            found = True
        else:
            previous = current
            current = current.getNext()
    if found:
        if previous is None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
    else:
        raise ValueError
        print 'Value not found.'

```

```

def insert(self, position, item):
    """
在指定位置插入项目。如果指定的位置超出范围，则引发 IndexError
    """

```

```

    如果 位置 > self.size() - 1:
        引发 索引错误
        打印 "索引超出范围。"
    当前 = self.头节点
    前一个 = 无
    位置 = 0
    如果 位置 是 0:
        self.添加(项目)
    else:
        新节点 = 节点(项目)
        当 位置 < 目标位置:
            位置 += 1
            前一个 = 当前
            当前 = 当前.获取下一个()
            前一个.设置下一个(新节点)
            新节点.设置下一个(当前)

```

```

    定义 索引(self, 项目):
    """

```

```

    返回找到项目的索引。
    如果未找到项目，返回 None。
    """

```

```

current = self.head
pos = 0
found = False
    while current is not None and not found:
        if current.getData() is item:
            found = True
        else:
            current = current.getNext()
            pos += 1
    if found:
        pass
    else:
        pos = None

```

```

    return found

def remove(self, item):
    """Remove item from list. If item is not found in list, raise ValueError"""
    current = self.head
previous = None
found = False
    while current is not None and not found:
        if current.getData() is item:
            found = True
        else:
            previous = current
            current = current.getNext()
    if found:
        if previous is None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
    else:
        raise ValueError
        print 'Value not found.'

```

```

def insert(self, position, item):
    """
Insert item at position specified. If position specified is
out of bounds, raise IndexError
    """
    if position > self.size() - 1:
        raise IndexError
        print "Index out of bounds."
    current = self.head
    previous = None
    pos = 0
    if position is 0:
        self.add(item)
    else:
        new_node = Node(item)
        while pos < position:
            pos += 1
            previous = current
            current = current.getNext()
        previous.setNext(new_node)
        new_node.setNext(current)

```

```

def index(self, item):
    """
Return the index where item is found.
If item is not found, return None.
    """
    current = self.head
    pos = 0
    found = False
    while current is not None and not found:
        if current.getData() is item:
            found = True
        else:
            current = current.getNext()
            pos += 1
    if found:
        pass
    else:
        pos = None

```

```
return pos
```

```
def pop(self, position = None):
```

```
    """
```

```
如果未提供参数, 返回并移除头部的项目。
```

```
如果提供了位置参数, 返回并移除该位置的项目。
```

```
如果索引超出范围, 抛出 IndexError
```

```
"""
```

```
if position > self.size():
```

```
    print '索引超出范围'
```

```
    raise IndexError
```

```
current = self.head
```

```
    if position is None:
```

```
        ret = current.getData()
```

```
        self.head = current.getNext()
```

```
    else:
```

```
pos = 0
```

```
    previous = None
```

```
    当 位置 < 目标位置:
```

```
previous = current
```

```
    current = current.getNext()
```

```
    pos += 1
```

```
ret = current.getData()
```

```
    previous.setNext(current.getNext())
```

```
print ret
```

```
return ret
```

```
def append(self, item):
```

```
    """将元素追加到列表末尾"""
    current = self.head
```

```
    previous = None
```

```
pos = 0
```

```
length = self.size()
```

```
    while pos < length:
```

```
previous = current
```

```
    current = current.getNext()
```

```
    pos += 1
```

```
new_node = Node(item)
```

```
    if previous is None:
```

```
new_node.setNext(current)
```

```
    self.head = new_node
```

```
else:
```

```
    previous.setNext(new_node)
```

```
def printList(self):
```

```
    """打印链表"""
    current = self.head
```

```
    while current is not None:
```

```
        print current.getData()
```

```
        current = current.getNext()
```

用法与内置列表的用法非常相似。

```
ll = LinkedList()
```

```
ll.add('l')
```

```
ll.add('H')
```

```
ll.insert(1, 'e')
```

```
ll.append('l')
```

```
ll.append('o')
```

```
ll.printList()
```

```
return pos
```

```
def pop(self, position = None):
```

```
    """
    If no argument is provided, return and remove the item at the head.
```

```
    If position is provided, return and remove the item at that position.
```

```
    If index is out of bounds, raise IndexError
```

```
    """

```

```
if position > self.size():
```

```
    print 'Index out of bounds'
```

```
    raise IndexError
```

```
current = self.head
```

```
if position is None:
```

```
    ret = current.getData()
```

```
    self.head = current.getNext()
```

```
else:
```

```
    pos = 0
```

```
    previous = None
```

```
    while pos < position:
```

```
        previous = current
```

```
        current = current.getNext()
```

```
        pos += 1
```

```
        ret = current.getData()
```

```
        previous.setNext(current.getNext())
```

```
print ret
```

```
return ret
```

```
def append(self, item):
```

```
    """Append item to the end of the list"""
    current = self.head
```

```
    previous = None
```

```
pos = 0
```

```
length = self.size()
```

```
while pos < length:
```

```
    previous = current
```

```
    current = current.getNext()
```

```
    pos += 1
```

```
new_node = Node(item)
```

```
if previous is None:
```

```
    new_node.setNext(current)
```

```
    self.head = new_node
```

```
else:
```

```
    previous.setNext(new_node)
```

```
def printList(self):
```

```
    """Print the list"""
    current = self.head
```

```
    while current is not None:
```

```
        print current.getData()
```

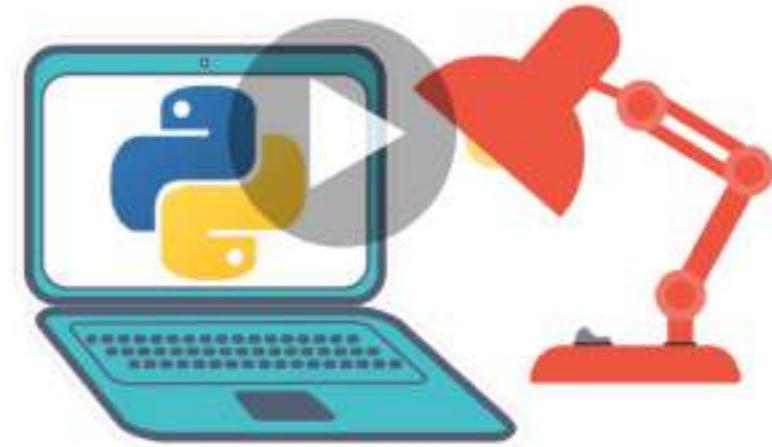
```
        current = current.getNext()
```

H  
e  
l  
l  
o

# 视频：完整的Python 训练营：从零开始 成为Python 3高手

像专业人士一样学习Python！从基础开始，直到  
创建你自己的应用程序和游戏！

## COMPLETE PYTHON 3 BOOTCAMP



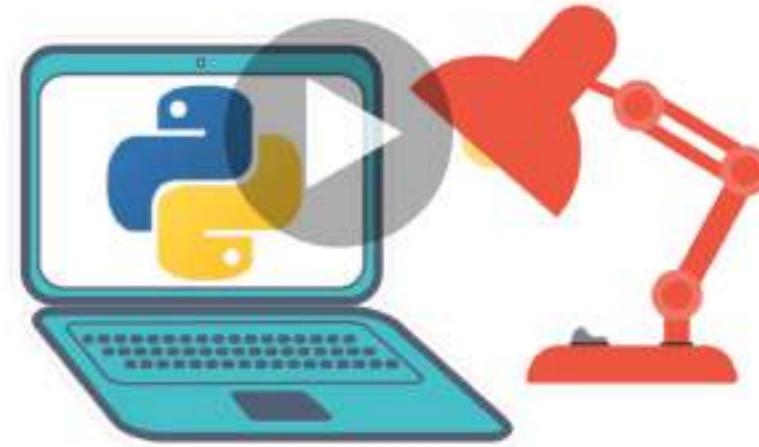
- ✓ 学习专业使用Python，掌握Python 2和Python 3！
- ✓ 用Python创建游戏，比如井字棋和二十一点！
- ✓ 学习高级Python功能，如collections模块和时间戳的使用！
- ✓ 学习使用面向对象编程和类！
- ✓ 理解复杂主题，如装饰器。
- ✓ 理解如何使用Jupyter Notebook并创建.py文件
- ✓ 了解如何在Jupyter Notebook系统中创建图形用户界面（GUI）！
- ✓ 从零开始全面理解Python！

今天观看 →

# VIDEO: Complete Python Bootcamp: Go from zero to hero in Python 3

Learn Python like a Professional! Start from the basics and go all the way to creating your own applications and games!

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ Learn to use Python professionally, learning both Python 2 and Python 3!
- ✓ Create games with Python, like Tic Tac Toe and Blackjack!
- ✓ Learn advanced Python features, like the collections module and how to work with timestamps!
- ✓ Learn to use Object Oriented Programming with classes!
- ✓ Understand complex topics, like decorators.
- ✓ Understand how to use both the Jupyter Notebook and create .py files
- ✓ Get an understanding of how to create GUIs in the Jupyter Notebook system!
- ✓ Build a complete understanding of Python from the ground up!

Watch Today →

# 第25章：链表节点

## 第25.1节：用Python编写一个简单的链表节点

链表是以下之一：

- 空链表，用None表示，或者
- 包含一个数据对象和指向链表的引用的节点。

```
#! /usr/bin/env python
```

```
类 Node:  
    def __init__(self, cargo=None, next=None):  
        self.car = 货物  
        self.cdr = 下一个  
    def __str__(self):  
        返回 str(self.car)  
  
    定义 display(lst):  
        如果 lst:  
            w("%s " % lst)  
            display(lst.cdr)  
        else:  
            w("nil")
```

# Chapter 25: Linked List Node

## Section 25.1: Write a simple Linked List Node in python

A linked list is either:

- the empty list, represented by None, or
- a node that contains a cargo object and a reference to a linked list.

```
#! /usr/bin/env python
```

```
class Node:  
    def __init__(self, cargo=None, next=None):  
        self.car = cargo  
        self.cdr = next  
    def __str__(self):  
        return str(self.car)  
  
    def display(lst):  
        if lst:  
            w("%s " % lst)  
            display(lst.cdr)  
        else:  
            w("nil\n")
```

# 第26章：过滤器

参数	详情
函数	可调用对象 用于确定条件，或None则使用恒等函数进行过滤（仅限位置参数）
可迭代对象	将被过滤的可迭代对象（仅限位置参数）

## 第26.1节：滤波器的基本使用

根据某些条件过滤序列中的元素：

```
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5

Python 2.x 版本 ≥ 2.0
filter(long_name, names)
# 输出: ['Barney']

[name for name in names if len(name) > 5] # 等价的列表推导式
# 输出: ['Barney']
```

```
from itertools import ifilter
ifilter(long_name, names)      # 作为生成器（类似于 Python 3.x 内置的 filter）
# 输出: <itertools.ifilter at 0x4197e10>
list(ifilter(long_name, names)) # 等同于使用列表的filter
# 输出: ['Barney']

(name for name in names if len(name) > 5) # 等价的生成器表达式
# 输出: <generator object <genexpr> at 0x000000003FD5D38>
```

```
Python 2.x 版本 ≥ 2.6
# 除了适用于旧版python 2.x的选项外，还有一个future_builtin函数：
from future_builtins import filter
filter(long_name, names)      # 与itertools.ifilter相同
# 输出: <itertools.ifilter at 0x3eb0ba8>
```

```
Python 3.x 版本 ≥ 3.0
filter(long_name, names)      # 返回一个生成器
# 输出: <filter at 0x1fc6e443470>
list(filter(long_name, names)) # 转换为列表
# 输出: ['Barney']

(name for name in names if len(name) > 5) # 等价的生成器表达式
# 输出: <generator object <genexpr> at 0x000001C6F49BF4C0>
```

## 第26.2节：无函数的filter

如果函数参数是None，则将使用恒等函数：

```
list(filter(None, [1, 0, 2, [], '', 'a'])) # 丢弃0、[]和''
# 输出: [1, 2, 'a']

Python 2.x 版本 ≥ 2.0.1
[i for i in [1, 0, 2, [], '', 'a'] if i] # 等价的列表推导式

Python 3.x 版本 ≥ 3.0.0
```

# Chapter 26: Filter

Parameter	Details
function	callable that determines the condition or <code>None</code> then use the identity function for filtering ( <i>positional-only</i> )
iterable	iterable that will be filtered ( <i>positional-only</i> )

## Section 26.1: Basic use of filter

To `filter` discards elements of a sequence based on some criteria:

```
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5

Python 2.x Version ≥ 2.0
filter(long_name, names)
# Out: ['Barney']

[name for name in names if len(name) > 5] # equivalent list comprehension
# Out: ['Barney']
```

```
from itertools import ifilter
ifilter(long_name, names)      # as generator (similar to python 3.x filter builtin)
# Out: <itertools.ifilter at 0x4197e10>
list(ifilter(long_name, names)) # equivalent to filter with lists
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x000000003FD5D38>
```

```
Python 2.x Version ≥ 2.6
# Besides the options for older python 2.x versions there is a future_builtin function:
from future_builtins import filter
filter(long_name, names)      # identical to itertools.ifilter
# Out: <itertools.ifilter at 0x3eb0ba8>
```

```
Python 3.x Version ≥ 3.0
filter(long_name, names)      # returns a generator
# Out: <filter at 0x1fc6e443470>
list(filter(long_name, names)) # cast to list
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x000001C6F49BF4C0>
```

## Section 26.2: Filter without function

If the function parameter is `None`, then the identity function will be used:

```
list(filter(None, [1, 0, 2, [], '', 'a'])) # discards 0, [] and ''
# Out: [1, 2, 'a']

Python 2.x Version ≥ 2.0.1
[i for i in [1, 0, 2, [], '', 'a'] if i] # equivalent list comprehension

Python 3.x Version ≥ 3.0.0
```

```
(i for i in [1, 0, 2, [], '', 'a']) if i) # 等价的生成器表达式
```

## 第26.3节：作为短路检查的filter

filter (python 3.x) 和 ifilter (python 2.x) 返回生成器，因此在创建类似 or 或 and 的短路测试时非常方便：

Python 2.x 版本 ≥ 2.0.1

```
# 实际使用中不推荐，但保持示例简短：  
from itertools import ifilter as filter
```

Python 2.x 版本 ≥ 2.6.1

```
from future_builtins import filter
```

查找第一个小于100的元素：

```
car_shop = [('丰田', 1000), ('矩形轮胎', 80), ('保时捷', 5000)]  
def find_something_smaller_than(name_value_tuple):  
    print('检查 {0}, {1}'.format(*name_value_tuple))  
    return name_value_tuple[1] < 100  
next(filter(find_something_smaller_than, car_shop))  
# 输出: 检查丰田, 1000$  
#       检查矩形轮胎, 80$  
# 结果: ('矩形轮胎', 80)
```

函数 next 返回序列中的下一个元素（此处为第一个元素），因此它是短路操作的原因。

## 第26.4节：补充函数：filterfalse, ifilterfalse

在 itertools 模块中有一个与 filter 互补的函数：

Python 2.x 版本 ≥ 2.0.1

```
# 在实际使用中不推荐，但保持示例在python 2.x和python 3.x中有效  
from itertools import ifilterfalse as filterfalse
```

Python 3.x 版本 ≥ 3.0.0

```
from itertools import filterfalse
```

其工作方式与generatorfilter完全相同，但只保留False的元素：

```
# 不使用函数 (None) 的用法：  
list(filterfalse(None, [1, 0, 2, [], '', 'a'])) # 丢弃 1、2、'a'  
# 输出: [0, [], '']
```

```
# 使用函数的用法  
names = ['Fred', 'Wilma', 'Barney']  
  
def long_name(name):  
    return len(name) > 5  
  
list(filterfalse(long_name, names))  
# 输出: ['Fred', 'Wilma']
```

```
# 使用 next 的短路用法：  
car_shop = [('丰田', 1000), ('矩形轮胎', 80), ('保时捷', 5000)]  
def find_something_smaller_than(name_value_tuple):
```

```
(i for i in [1, 0, 2, [], '', 'a']) if i) # equivalent generator expression
```

## Section 26.3: Filter as short-circuit check

filter (python 3.x) and ifilter (python 2.x) return a generator so they can be very handy when creating a short-circuit test like or or and:

Python 2.x 版本 ≥ 2.0.1

```
# not recommended in real use but keeps the example short:  
from itertools import ifilter as filter
```

Python 2.x 版本 ≥ 2.6.1

```
from future_builtins import filter
```

To find the first element that is smaller than 100:

```
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]  
def find_something_smaller_than(name_value_tuple):  
    print('Check {0}, {1}'.format(*name_value_tuple))  
    return name_value_tuple[1] < 100  
next(filter(find_something_smaller_than, car_shop))  
# Print: Check Toyota, 1000$  
#       Check rectangular tire, 80$  
# Out: ('rectangular tire', 80)
```

The next-function gives the next (in this case first) element of and is therefore the reason why it's short-circuit.

## Section 26.4: Complementary function: filterfalse, ifilterfalse

There is a complementary function for filter in the itertools-module:

Python 2.x 版本 ≥ 2.0.1

```
# not recommended in real use but keeps the example valid for python 2.x and python 3.x  
from itertools import ifilterfalse as filterfalse
```

Python 3.x 版本 ≥ 3.0.0

```
from itertools import filterfalse
```

which works exactly like the generator filter but keeps only the elements that are False:

```
# Usage without function (None):  
list(filterfalse(None, [1, 0, 2, [], '', 'a'])) # discards 1, 2, 'a'  
# Out: [0, [], '']
```

```
# Usage with function  
names = ['Fred', 'Wilma', 'Barney']  
  
def long_name(name):  
    return len(name) > 5  
  
list(filterfalse(long_name, names))  
# Out: ['Fred', 'Wilma']
```

```
# Short-circuit usage with next:  
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]  
def find_something_smaller_than(name_value_tuple):
```

```
print('检查 {0}, {1}'.format(*name_value_tuple))
    return name_value_tuple[1] < 100
next(filterfalse(find_something_smaller_than, car_shop))
# 打印: 检查 Toyota, 1000$
# 输出: ('Toyota', 1000)
```

```
# 使用等效生成器:
car_shop = [('丰田', 1000), ('矩形轮胎', 80), ('保时捷', 5000)]
generator = (car for car in car_shop if not car[1] < 100)
next(generator)
```

```
print('Check {0}, {1}'.format(*name_value_tuple))
    return name_value_tuple[1] < 100
next(filterfalse(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
# Out: ('Toyota', 1000)
```

```
# Using an equivalent generator:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
generator = (car for car in car_shop if not car[1] < 100)
next(generator)
```

## 第27章：Heapq（堆队列）

### 第27.1节：集合中的最大和最小项

要查找集合中的最大项，heapq模块有一个名为nlargest的函数，我们传递两个参数，第一个是我们想要检索的项数，第二个是集合名称：

```
import heapq
```

```
numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers)) # [200, 150, 100, 50]
```

同样地，要查找集合中的最小项，我们使用nsmallest函数：

```
print(heapq.nsmallest(4, numbers)) # [1, 2, 4, 8]
```

Both nlargest and nsmallest 函数都接受一个可选参数（key 参数）用于复杂数据结构。以下示例展示了如何使用 age 属性从 people 字典中获取年龄最大和最小的人：

```
people = [
    {'firstname': 'John', 'lastname': 'Doe', 'age': 30},
    {'firstname': 'Jane', 'lastname': 'Doe', 'age': 25},
    {'firstname': 'Janie', 'lastname': 'Doe', 'age': 10},
    {'firstname': 'Jane', 'lastname': 'Roe', 'age': 22},
    {'firstname': 'Johnny', 'lastname': 'Doe', 'age': 12},
    {'firstname': 'John', 'lastname': 'Roe', 'age': 45}
]

oldest = heapq.nlargest(2, people, key=lambda s: s['age'])
print(oldest)
# 输出: [{"firstname": "John", "age": 45, "lastname": "Roe"}, {"firstname": "John", "age": 30, "lastname": "Doe"}]

youngest = heapq.nsmallest(2, people, key=lambda s: s['age'])
print(youngest)
# 输出: [{"firstname": "Janie", "age": 10, "lastname": "Doe"}, {"firstname": "Johnny", "age": 12, "lastname": "Doe"}]
```

### 第27.2节：集合中的最小项

堆（heap）最有趣的性质是其最小的元素总是第一个元素：heap[0]

```
import heapq
```

```
numbers = [10, 4, 2, 100, 20, 50, 32, 200, 150, 8]

heapq.heapify(numbers)
print(numbers)
# 输出: [2, 4, 10, 100, 8, 50, 32, 200, 150, 20]

heapq.heappop(numbers) # 2
print(numbers)
# 输出: [4, 8, 10, 100, 20, 50, 32, 200, 150]
```

## Chapter 27: Heapq

### Section 27.1: Largest and smallest items in a collection

To find the largest items in a collection, `heapq` module has a function called `nlargest`, we pass it two arguments, the first one is the number of items that we want to retrieve, the second one is the collection name:

```
import heapq
```

```
numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers)) # [200, 150, 100, 50]
```

Similarly, to find the smallest items in a collection, we use `nsmallest` function:

```
print(heapq.nsmallest(4, numbers)) # [1, 2, 4, 8]
```

Both `nlargest` and `nsmallest` functions take an optional argument (key parameter) for complicated data structures. The following example shows the use of `age` property to retrieve the oldest and the youngest people from `people` dictionary:

```
people = [
    {'firstname': 'John', 'lastname': 'Doe', 'age': 30},
    {'firstname': 'Jane', 'lastname': 'Doe', 'age': 25},
    {'firstname': 'Janie', 'lastname': 'Doe', 'age': 10},
    {'firstname': 'Jane', 'lastname': 'Roe', 'age': 22},
    {'firstname': 'Johnny', 'lastname': 'Doe', 'age': 12},
    {'firstname': 'John', 'lastname': 'Roe', 'age': 45}
]

oldest = heapq.nlargest(2, people, key=lambda s: s['age'])
print(oldest)
# Output: [{"firstname": "John", "age": 45, "lastname": "Roe"}, {"firstname": "John", "age": 30, "lastname": "Doe"}]

youngest = heapq.nsmallest(2, people, key=lambda s: s['age'])
print(youngest)
# Output: [{"firstname": "Janie", "age": 10, "lastname": "Doe"}, {"firstname": "Johnny", "age": 12, "lastname": "Doe"}]
```

### Section 27.2: Smallest item in a collection

The most interesting property of a heap is that its smallest element is always the first element: `heap[0]`

```
import heapq
```

```
numbers = [10, 4, 2, 100, 20, 50, 32, 200, 150, 8]

heapq.heapify(numbers)
print(numbers)
# Output: [2, 4, 10, 100, 8, 50, 32, 200, 150, 20]

heapq.heappop(numbers) # 2
print(numbers)
# Output: [4, 8, 10, 100, 20, 50, 32, 200, 150]
```

```
heapq.heappop(numbers) # 4  
print(numbers)  
# 输出: [8, 20, 10, 100, 150, 50, 32, 200]
```

```
heapq.heappop(numbers) # 4  
print(numbers)  
# Output: [8, 20, 10, 100, 150, 50, 32, 200]
```

# 第28章：元组

元组是不可变的值列表。元组是Python中最简单且最常见的集合类型之一，可以通过逗号操作符创建 (value = 1, 2, 3)。

## 第28.1节：元组

从语法上讲，元组是由逗号分隔的值列表：

```
t = 'a', 'b', 'c', 'd', 'e'
```

虽然不是必须的，但通常会将元组用括号括起来：

```
t = ('a', 'b', 'c', 'd', 'e')
```

用括号创建一个空元组：

```
t0 = ()  
type(t0)      # <type 'tuple'>
```

要创建一个只有一个元素的元组，必须在元素后面加上逗号：

```
t1 = 'a'  
type(t1)      # <type 'tuple'>
```

注意，单个值用括号括起来并不是元组：

```
t2 = ('a')  
type(t2)      # <type 'str'>
```

要创建单元素元组，必须有一个尾随逗号。

```
t2 = ('a',)  
type(t2)      # <type 'tuple'>
```

注意，对于单元素元组，建议使用括号（参见[PEP8关于尾随逗号的规定](#)）。此外，尾随逗号后不应有空白（参见[PEP8关于空白的规定](#)）。

```
t2 = ('a',)      # 符合PEP8规范  
t2 = 'a',        # PEP8不推荐这种写法  
t2 = ('a', )     # PEP8不推荐这种写法
```

创建元组的另一种方法是使用内置函数tuple。

```
t = tuple('lupins')  
print(t)          # ('l', 'u', 'p', 'i', 'n', 's')  
t = tuple(range(3))  
print(t)          # (0, 1, 2)
```

这些例子基于艾伦·B·道尼 (Allen B. Downey) 所著的《Think Python》一书的内容。

# Chapter 28: Tuple

A tuple is an immutable list of values. Tuples are one of Python's simplest and most common collection types, and can be created with the comma operator (value = 1, 2, 3).

## Section 28.1: Tuple

Syntactically, a tuple is a comma-separated list of values:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Although not necessary, it is common to enclose tuples in parentheses:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Create an empty tuple with parentheses:

```
t0 = ()  
type(t0)      # <type 'tuple'>
```

To create a tuple with a single element, you have to include a final comma:

```
t1 = 'a',  
type(t1)      # <type 'tuple'>
```

Note that a single value in parentheses is not a tuple:

```
t2 = ('a')  
type(t2)      # <type 'str'>
```

To create a singleton tuple it is necessary to have a trailing comma.

```
t2 = ('a',)  
type(t2)      # <type 'tuple'>
```

Note that for singleton tuples it's recommended (see [PEP8 on trailing commas](#)) to use parentheses. Also, no white space after the trailing comma (see [PEP8 on whitespaces](#))

```
t2 = ('a',)      # PEP8-compliant  
t2 = 'a',        # this notation is not recommended by PEP8  
t2 = ('a', )     # this notation is not recommended by PEP8
```

Another way to create a tuple is the built-in function tuple.

```
t = tuple('lupins')  
print(t)          # ('l', 'u', 'p', 'i', 'n', 's')  
t = tuple(range(3))  
print(t)          # (0, 1, 2)
```

These examples are based on material from the book [Think Python](#) by Allen B. Downey.

## 第28.2节：元组是不可变的

Python中列表（list）和元组（tuple）的主要区别之一是元组是不可变的，也就是说，元组初始化后不能添加或修改元素。例如：

```
>>> t = (1, 4, 9)
>>> t[0] = 2
回溯 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 模块中
TypeError: 'tuple' 对象不支持项赋值
```

同样，元组没有.append和.extend方法，而列表有。使用+=是可能的，但它改变的是变量的绑定，而不是元组本身：

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

在将可变对象（例如列表）放入元组时要小心。这可能会导致在修改它们时出现非常混乱的结果。  
例如：

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

这两者都会引发错误并改变元组中列表的内容：

```
TypeError: 'tuple' 对象不支持项赋值
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

你可以使用+=运算符来“追加”到元组——这是通过创建一个包含你“追加”的新元素的新元组，并将其赋值给当前变量来实现的；旧元组不会被修改，而是被替换！

这避免了转换为列表和从列表转换回来，但这很慢且是不好的做法，尤其是当你要多次追加时。

## 第28.3节：元组的打包与解包

Python中的元组是由逗号分隔的值。输入元组时括号是可选的，因此以下两种赋值

```
a = 1, 2, 3 # a 是元组 (1, 2, 3)
```

和

```
a = (1, 2, 3) # a 是元组 (1, 2, 3)
```

是等价的。赋值 a = 1, 2, 3 也称为打包，因为它将值打包成一个元组。

注意，一个只有一个值的元组也是元组。要告诉 Python 一个变量是元组而不是单个值，可以使用

## Section 28.2: Tuples are immutable

One of the main differences between `lists` and `tuples` in Python is that tuples are immutable, that is, one cannot add or modify items once the tuple is initialized. For example:

```
>>> t = (1, 4, 9)
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Similarly, tuples don't have `.append` and `.extend` methods as `list` does. Using `+=` is possible, but it changes the binding of the variable, and not the tuple itself:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

Be careful when placing mutable objects, such as `lists`, inside tuples. This may lead to very confusing outcomes when changing them. For example:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

Will **both** raise an error and change the contents of the list within the tuple:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

You can use the `+=` operator to "append" to a tuple - this works by creating a new tuple with the new element you "appended" and assign it to its current variable; the old tuple is not changed, but replaced!

This avoids converting to and from a list, but this is slow and is a bad practice, especially if you're going to append multiple times.

## Section 28.3: Packing and Unpacking Tuples

Tuples in Python are values separated by commas. Enclosing parentheses for inputting tuples are optional, so the two assignments

```
a = 1, 2, 3 # a is the tuple (1, 2, 3)
```

and

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

are equivalent. The assignment `a = 1, 2, 3` is also called *packing* because it packs values together in a tuple.

Note that a one-value tuple is also a tuple. To tell Python that a variable is a tuple and not a single value you can use

## 尾随逗号

```
a = 1 # a 是值 1  
a = 1, # a 是元组 (1,)
```

如果使用括号，也需要逗号

```
a = (1,) # a 是元组 (1,)  
a = (1) # a 是值 1, 不是元组
```

要从元组中解包值并进行多重赋值，使用

```
# 解包, 也称多重赋值  
x, y, z = (1, 2, 3)  
# x == 1  
# y == 2  
# z == 3
```

符号 `_` 可以用作一次性变量名，如果只需要元组中的某些元素，作为一个占位符：

```
a = 1, 2, 3, 4  
_, x, y, _ = a  
# x == 2  
# y == 3
```

单元素元组：

```
x, = 1, # x 是值 1  
x = 1, # x 是元组 (1,)
```

在 Python 3 中，带有 `*` 前缀的目标变量可以用作捕获所有变量（参见可迭代对象拆包）：

```
Python 3.x 版本 ≥ 3.0  
first, *more, last = (1, 2, 3, 4, 5)  
# first == 1  
# more == [2, 3, 4]  
# last == 5
```

## 第28.4节：内置元组函数

元组支持以下内置函数

### 比较

如果元素类型相同，Python 会执行比较并返回结果。如果元素类型不同，则检查它们是否为数字。

- 如果是数字，则执行比较。
- 如果任一元素是数字，则返回另一个元素。
- 否则，类型按字母顺序排序。

如果到达其中一个列表的末尾，较长的列表被视为“较大”。如果两个列表相同，则返回0。

```
tuple1 = ('a', 'b', 'c', 'd', 'e')  
tuple2 = ('1', '2', '3')
```

### a trailing comma

```
a = 1 # a is the value 1  
a = 1, # a is the tuple (1,)
```

A comma is needed also if you use parentheses

```
a = (1,) # a is the tuple (1,)  
a = (1) # a is the value 1 and not a tuple
```

To unpack values from a tuple and do multiple assignments use

```
# unpacking AKA multiple assignment  
x, y, z = (1, 2, 3)  
# x == 1  
# y == 2  
# z == 3
```

The symbol `_` can be used as a disposable variable name if one only needs some elements of a tuple, acting as a placeholder:

```
a = 1, 2, 3, 4  
_, x, y, _ = a  
# x == 2  
# y == 3
```

Single element tuples:

```
x, = 1, # x is the value 1  
x = 1, # x is the tuple (1,)
```

In Python 3 a target variable with a `*` prefix can be used as a [catch-all](#) variable (see Unpacking Iterables):

```
Python 3.x 版本 ≥ 3.0  
first, *more, last = (1, 2, 3, 4, 5)  
# first == 1  
# more == [2, 3, 4]  
# last == 5
```

## Section 28.4: Built-in Tuple Functions

Tuples support the following build-in functions

### Comparison

If elements are of the same type, python performs the comparison and returns the result. If elements are different types, it checks whether they are numbers.

- If numbers, perform comparison.
- If either element is a number, then the other element is returned.
- Otherwise, types are sorted alphabetically .

If we reached the end of one of the lists, the longer list is "larger." If both list are same it returns 0.

```
tuple1 = ('a', 'b', 'c', 'd', 'e')  
tuple2 = ('1', '2', '3')
```

```
tuple3 = ('a', 'b', 'c', 'd', 'e')
```

```
cmp(tuple1, tuple2)
```

输出: 1

```
cmp(tuple2, tuple1)
```

输出: -1

```
cmp(tuple1, tuple3)
```

输出: 0

## 元组长度

函数len返回元组的总长度

```
len(tuple1)
```

输出: 5

## 元组的最大值

函数max返回元组中最大值的元素

```
max(tuple1)
```

输出: 'e'

```
max(tuple2)
```

输出: '3'

## 元组的最小值

函数min返回元组中最小值的元素

```
min(tuple1)
```

输出: 'a'

```
min(tuple2)
```

输出: '1'

## 将列表转换为元组

内置函数 tuple 可以将列表转换为元组。

```
list = [1, 2, 3, 4, 5]
```

```
tuple(list)
```

输出: (1, 2, 3, 4, 5)

## 元组连接

使用+连接两个元组

```
tuple1 + tuple2
```

输出: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')

## 第28.5节：元组的元素可逐个进行哈希和比较

```
hash( (1, 2) ) # 可以
```

```
hash( [], {"hello"}) # 不行, 因为列表和集合不可哈希
```

因此，只有当元组的每个元素都可哈希时，元组才能放入集合中或作为字典的键。

```
{ (1, 2) } # 可以
```

```
tuple3 = ('a', 'b', 'c', 'd', 'e')
```

```
cmp(tuple1, tuple2)
```

Out: 1

```
cmp(tuple2, tuple1)
```

Out: -1

```
cmp(tuple1, tuple3)
```

Out: 0

## Tuple Length

The function len returns the total length of the tuple

```
len(tuple1)
```

Out: 5

## Max of a tuple

The function max returns item from the tuple with the max value

```
max(tuple1)
```

Out: 'e'

```
max(tuple2)
```

Out: '3'

## Min of a tuple

The function min returns the item from the tuple with the min value

```
min(tuple1)
```

Out: 'a'

```
min(tuple2)
```

Out: '1'

## Convert a list into tuple

The built-in function tuple converts a list into a tuple.

```
list = [1, 2, 3, 4, 5]
```

```
tuple(list)
```

Out: (1, 2, 3, 4, 5)

## Tuple concatenation

Use + to concatenate two tuples

```
tuple1 + tuple2
```

Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')

## Section 28.5: Tuple Are Element-wise Hashable and Equatable

```
hash( (1, 2) ) # ok
```

```
hash( ([], {"hello"})) # not ok, since lists and sets are not hashable
```

Thus a tuple can be put inside a set or as a key in a dict only if each of its elements can.

```
{ (1, 2) } # ok
```

```
{ (), {"hello"}) # 不可以
```

## 第28.6节：元组索引

```
x = (1, 2, 3)
x[0] # 1
x[1] # 2
x[2] # 3
x[3] # IndexError: 元组索引超出范围
```

使用负数索引将从最后一个元素开始，索引为 -1：

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

索引一系列元素

```
print(x[:-1]) # (1, 2)
print(x[-1:]) # (3,)
print(x[1:3]) # (2, 3)
```

## 第28.7节：反转元素

反转元组中的元素

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# 颜色: ("blue", "green", "red")
```

或者使用 reversed (reversed 返回一个可迭代对象，可以转换为元组)：

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# 颜色: ("blue", "green", "red")
```

```
{ (), {"hello"}) # not ok
```

## Section 28.6: Indexing Tuples

```
x = (1, 2, 3)
x[0] # 1
x[1] # 2
x[2] # 3
x[3] # IndexError: tuple index out of range
```

Indexing with negative numbers will start from the last element as -1:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

Indexing a range of elements

```
print(x[:-1]) # (1, 2)
print(x[-1:]) # (3,)
print(x[1:3]) # (2, 3)
```

## Section 28.7: Reversing Elements

Reverse elements within a tuple

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Or using reversed (reversed gives an iterable which is converted to a tuple):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

# 第29章：基本输入和输出

## 第29.1节：使用 print 函数

Python 3.x 版本 ≥ 3.0

在 Python 3 中，print 功能以函数的形式存在：

```
print("这个字符串将显示在输出中")
# 这个字符串将显示在输出中

print("你也可以打印 转义字符。")# 你也可以打印转义字符。
```

Python 2.x 版本 ≥ 2.3

在 Python 2 中，print 最初是一个语句，如下所示。

```
print "该字符串将显示在输出中"
# 这个字符串将显示在输出中

print "你也可以打印 转义字符。"# 你也可以打印转义字符。
```

注意：在 Python 2 中使用 `from __future__ import print_function` 可以让用户像 Python 3 代码一样使用 `print()` 函数。此功能仅在 Python 2.6 及以上版本可用。

## 第29.2节：从文件输入

输入也可以从文件中读取。文件可以使用内置函数 `open` 打开。使用 `with <command> as <name>` 语法（称为“上下文管理器”）使得使用 `open` 并获取文件句柄变得非常简单：

```
with open('somefile.txt', 'r') as fileobj:
    # 在这里使用 fileobj 编写代码
```

这确保了当代码执行离开该代码块时，文件会自动关闭。

文件可以以不同模式打开。在上述示例中，文件以只读方式打开。要以只读方式打开现有文件，使用 `r`。如果想以字节形式读取该文件，使用 `rb`。要向现有文件追加数据，使用 `a`。使用 `w` 可以创建文件或覆盖同名的现有文件。你可以使用 `r+` 同时打开文件进行读写。`open()` 的第一个参数是文件名，第二个是模式。如果模式留空，默认是 `r`。

```
# 让我们创建一个示例文件：
使用open('shoppinglist.txt', 'w') 作为 fileobj:
fileobj.write('tomato\tpasta\ngarlic')
```

```
使用open('shoppinglist.txt', 'r') 作为 fileobj:
    # 该方法将文件的每一行作为列表中的一个元素
    #
lines = fileobj.readlines()
```

```
print(lines)
# ['tomato', 'pasta', 'garlic']使用open('sho
pinglist.txt', 'r') 作为 fileobj:
```

# Chapter 29: Basic Input and Output

## Section 29.1: Using the print function

Python 3.x Version ≥ 3.0

In Python 3, print functionality is in the form of a function:

```
print("This string will be displayed in the output")
# This string will be displayed in the output

print("You can print \n escape characters too.")
# You can print escape characters too.
```

Python 2.x Version ≥ 2.3

In Python 2, print was originally a statement, as shown below.

```
print "This string will be displayed in the output"
# This string will be displayed in the output

print "You can print \n escape characters too."
# You can print escape characters too.
```

Note: using `from __future__ import print_function` in Python 2 will allow users to use the `print()` function the same as Python 3 code. This is only available in Python 2.6 and above.

## Section 29.2: Input from a File

Input can also be read from files. Files can be opened using the built-in function `open`. Using a `with <command> as <name>` syntax (called a 'Context Manager') makes using `open` and getting a handle for the file super easy:

```
with open('somefile.txt', 'r') as fileobj:
    # write code here using fileobj
```

This ensures that when code execution leaves the block the file is automatically closed.

Files can be opened in different modes. In the above example the file is opened as read-only. To open an existing file for reading only use `r`. If you want to read that file as bytes use `rb`. To append data to an existing file use `a`. Use `w` to create a file or overwrite any existing files of the same name. You can use `r+` to open a file for both reading and writing. The first argument of `open()` is the filename, the second is the mode. If mode is left blank, it will default to `r`.

```
# let's create an example file:
with open('shoppinglist.txt', 'w') as fileobj:
    fileobj.write('tomato\tpasta\ngarlic')
```

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method makes a list where each line
    # of the file is an element in the list
    lines = fileobj.readlines()
```

```
print(lines)
# ['tomato\n', 'pasta\n', 'garlic']
```

```
with open('shoppinglist.txt', 'r') as fileobj:
```

```
# 这里我们将整个内容读取为一个字符串：  
content = fileobj.read()  
# 获取一行列表，就像前面的例子一样：  
lines = content.split()  
  
print(lines)  
# ['tomato', 'pasta', 'garlic']
```

如果文件大小很小，将整个文件内容读入内存是安全的。如果文件非常大，通常更好逐行或分块读取，并在同一个循环中处理输入。为此：

```
使用open('shoppinglist.txt', 'r') 作为 fileobj:  
# 这种方法逐行读取：  
lines = []  
for line in fileobj:  
    lines.append(line.strip())
```

读取文件时，要注意操作系统特定的换行符。虽然 `for line in fileobj` 会自动去除它们，但调用 `strip()` 来处理读取的行总是安全的，如上所示。

打开的文件（上述示例中的 `fileobj`）总是指向文件中的特定位置。文件首次打开时，文件指针指向文件的开头，即位置 0。文件指针可以用 `tell` 显示其当前位置：

```
fileobj = open('shoppinglist.txt', 'r')  
pos = fileobj.tell()  
print('我们在位置 %u.' % pos) # 我们在位置 0。
```

读取所有内容后，文件句柄的位置将指向文件末尾：

```
content = fileobj.read()  
end = fileobj.tell()  
print('该文件长度为 %u 个字符.' % end)  
# 该文件长度为 22 个字符。  
fileobj.close()
```

文件句柄的位置可以设置为所需的任意位置：

```
fileobj = open('shoppinglist.txt', 'r')  
fileobj.seek(7)  
pos = fileobj.tell()  
print('我们在第 %#u 个字符处.' % pos)
```

你也可以在一次调用中读取文件内容的任意长度。为此，向 `read()` 传递一个参数。当 `read()` 被调用且无参数时，它将读取直到文件末尾。如果传入参数，它将读取该字节数或字符数，具体取决于模式（分别为 `rb` 和 `r`）：

```
# 读取当前位置开始的下一个4个字符  
  
next4 = fileobj.read(4)  
# 我们得到了什么？  
print(next4) # 'cucu'  
# 我们现在在哪里？  
pos = fileobj.tell()  
print('我们现在在 %u.' % pos) # 我们在11，因为之前在7，读取了4个字符。  
  
fileobj.close()
```

```
# here we read the whole content into one string:  
content = fileobj.read()  
# get a list of lines, just like in the previous example:  
lines = content.split('\n')  
  
print(lines)  
# ['tomato', 'pasta', 'garlic']
```

If the size of the file is tiny, it is safe to read the whole file contents into memory. If the file is very large it is often better to read line-by-line or by chunks, and process the input in the same loop. To do that:

```
with open('shoppinglist.txt', 'r') as fileobj:  
    # this method reads line by line:  
    lines = []  
    for line in fileobj:  
        lines.append(line.strip())
```

When reading files, be aware of the operating system-specific line-break characters. Although `for line in fileobj` automatically strips them off, it is always safe to call `strip()` on the lines read, as it is shown above.

Opened files (`fileobj` in the above examples) always point to a specific location in the file. When they are first opened the file handle points to the very beginning of the file, which is the position 0. The file handle can display its current position with `tell`:

```
fileobj = open('shoppinglist.txt', 'r')  
pos = fileobj.tell()  
print('We are at %u.' % pos) # We are at 0.
```

Upon reading all the content, the file handler's position will be pointed at the end of the file:

```
content = fileobj.read()  
end = fileobj.tell()  
print('This file was %u characters long.' % end)  
# This file was 22 characters long.  
fileobj.close()
```

The file handler position can be set to whatever is needed:

```
fileobj = open('shoppinglist.txt', 'r')  
fileobj.seek(7)  
pos = fileobj.tell()  
print('We are at character %#u.' % pos)
```

You can also read any length from the file content during a given call. To do this pass an argument for `read()`. When `read()` is called with no argument it will read until the end of the file. If you pass an argument it will read that number of bytes or characters, depending on the mode (`rb` and `r` respectively):

```
# reads the next 4 characters  
# starting at the current position  
next4 = fileobj.read(4)  
# what we got?  
print(next4) # 'cucu'  
# where we are now?  
pos = fileobj.tell()  
print('We are at %u.' % pos) # We are at 11, as we was at 7, and read 4 chars.  
  
fileobj.close()
```

演示字符和字节之间的区别：

```
使用open('shoppinglist.txt', 'r') 作为 fileobj:  
print(type(fileobj.read())) # <class 'str'>  
  
with open('shoppinglist.txt', 'rb') as fileobj:  
print(type(fileobj.read())) # <class 'bytes'>
```

## 第29.3节：从标准输入读取

Python 程序可以从unix 管道读取。下面是一个如何从stdin读取的简单示例：

```
import sys  
  
for line in sys.stdin:  
    print(line)
```

请注意，sys.stdin 是一个流。这意味着 for 循环只有在流结束时才会终止。

你现在可以将另一个程序的输出通过管道传入你的 python 程序，方法如下：

```
$ cat myfile | python myprogram.py
```

在此示例中，cat myfile 可以是任何输出到stdout的 unix 命令。

或者，使用fileinput 模块也很方便：

```
import fileinput  
for line in fileinput.input():  
    process(line)
```

## 第29.4节：使用input()和raw\_input()

Python 2.x 版本 ≥ 2.3

raw\_input会等待用户输入文本，然后将结果作为字符串返回。

```
foo = raw_input("在此处放置提示用户输入的消息")
```

在上述示例中，foo将存储用户提供的任何输入。

Python 3.x 版本 ≥ 3.0

input会等待用户输入文本，然后将结果作为字符串返回。

```
foo = input("在此处放置提示用户输入的消息")
```

在上述示例中，foo将存储用户提供的任何输入。

## 第29.5节：提示用户输入数字的函数

```
def input_number(msg, err_msg=None):  
    while True:  
        try:            pass
```

To demonstrate the difference between characters and bytes:

```
with open('shoppinglist.txt', 'r') as fileobj:  
    print(type(fileobj.read())) # <class 'str'>  
  
with open('shoppinglist.txt', 'rb') as fileobj:  
    print(type(fileobj.read())) # <class 'bytes'>
```

## Section 29.3: Read from stdin

Python programs can read from [unix pipelines](#). Here is a simple example how to read from [stdin](#):

```
import sys  
  
for line in sys.stdin:  
    print(line)
```

Be aware that `sys.stdin` is a stream. It means that the for-loop will only terminate when the stream has ended.

You can now pipe the output of another program into your python program as follows:

```
$ cat myfile | python myprogram.py
```

In this example `cat myfile` can be any unix command that outputs to `stdout`.

Alternatively, using the [fileinput module](#) can come in handy:

```
import fileinput  
for line in fileinput.input():  
    process(line)
```

## Section 29.4: Using input() and raw\_input()

Python 2.x Version ≥ 2.3

`raw_input` will wait for the user to enter text and then return the result as a string.

```
foo = raw_input("Put a message here that asks the user for input")
```

In the above example `foo` will store whatever input the user provides.

Python 3.x Version ≥ 3.0

`input` will wait for the user to enter text and then return the result as a string.

```
foo = input("Put a message here that asks the user for input")
```

In the above example `foo` will store whatever input the user provides.

## Section 29.5: Function to prompt user for a number

```
def input_number(msg, err_msg=None):  
    while True:  
        try:            pass
```

```

    return float(raw_input(msg))
except ValueError:
    if err_msg is not None:
        print(err_msg)

def input_number(msg, err_msg=None):
    while True:
        try:
            return float(input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)

```

并且这样使用它：

```
user_number = input_number("输入一个数字: ", "这不是一个数字 !")
```

或者，如果你不想要“错误信息”：

```
user_number = input_number("输入一个数字: ")
```

## 第29.6节：打印字符串时不换行

Python 2.x 版本 ≥ 2.3

在Python 2.x中，要让print语句继续在同一行，需在print语句末尾加逗号。它会自动添加一个空格。

```

print "Hello, "
print "World!"
# Hello, World!

```

Python 3.x 版本 ≥ 3.0

在Python 3.x中，print函数有一个可选的end参数，用于指定打印给定字符串后结尾的内容。默认情况下它是一个换行符，因此等同于以下内容：

```

print("Hello, ", end="")
print("World!")
# Hello,
# World!

```

但是你可以传入其他字符串

```

print("Hello, ", end="")
print("World!")
# Hello, World!

print("Hello, ", end=<br>)
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!

```

如果你想更好地控制输出，可以使用 sys.stdout.write：

```

    return float(raw_input(msg))
except ValueError:
    if err_msg is not None:
        print(err_msg)

def input_number(msg, err_msg=None):
    while True:
        try:
            return float(input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)

```

And to use it:

```
user_number = input_number("input a number: ", "that's not a number!")
```

Or, if you do not want an "error message":

```
user_number = input_number("input a number: ")
```

## Section 29.6: Printing a string without a newline at the end

Python 2.x Version ≥ 2.3

In Python 2.x, to continue a line with **print**, end the **print** statement with a comma. It will automatically add a space.

```

print "Hello, "
print "World!"
# Hello, World!

```

Python 3.x Version ≥ 3.0

In Python 3.x, the **print** function has an optional end parameter that is what it prints at the end of the given string. By default it's a newline character, so equivalent to this:

```

print("Hello, ", end="\n")
print("World!")
# Hello,
# World!

```

But you could pass in other strings

```

print("Hello, ", end="")
print("World!")
# Hello, World!

```

```

print("Hello, ", end=<br>)
print("World!")
# Hello, <br>World!

```

```

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!

```

If you want more control over the output, you can use **sys.stdout.write**:

```
import sys
```

```
sys.stdout.write("Hello, ")  
sys.stdout.write("World!")  
# Hello, World!
```

```
import sys
```

```
sys.stdout.write("Hello, ")  
sys.stdout.write("World!")  
# Hello, World!
```

# 视频：Python 数据 科学与机器 学习训练营

学习如何使用 NumPy、Pandas、Seaborn、  
Matplotlib、Plotly、Scikit-Learn、机器学习、  
Tensorflow 等！



- ✓ 使用 Python 进行数据科学和机器学习
- ✓ 使用 Spark 进行大数据分析
- ✓ 实现机器学习算法
- ✓ 学习使用 NumPy 处理数值数据
- ✓ 学习使用 Pandas 进行数据分析
- ✓ 学习使用 Matplotlib 进行 Python 绘图
- ✓ 学习使用 Seaborn 进行统计图表绘制
- ✓ 使用 Plotly 进行交互式动态可视化
- ✓ 使用 SciKit-Learn 进行机器学习任务
- ✓ K-均值聚类
- ✓ 逻辑回归
- ✓ 线性回归
- ✓ 随机森林和决策树
- ✓ 神经网络
- ✓ 支持向量机今天观看→

# VIDEO: Python for Data Science and Machine Learning Bootcamp

Learn how to use NumPy, Pandas, Seaborn,  
Matplotlib , Plotly, Scikit-Learn , Machine Learning,  
Tensorflow, and more!



- ✓ Use Python for Data Science and Machine Learning
- ✓ Use Spark for Big Data Analysis
- ✓ Implement Machine Learning Algorithms
- ✓ Learn to use NumPy for Numerical Data
- ✓ Learn to use Pandas for Data Analysis
- ✓ Learn to use Matplotlib for Python Plotting
- ✓ Learn to use Seaborn for statistical plots
- ✓ Use Plotly for interactive dynamic visualizations
- ✓ Use SciKit-Learn for Machine Learning Tasks
- ✓ K-Means Clustering
- ✓ Logistic Regression
- ✓ Linear Regression
- ✓ Random Forest and Decision Trees
- ✓ Neural Networks
- ✓ Support Vector Machines

**Watch Today →**

# 第30章：文件与文件夹输入输出

## 参数

	详情
文件名	文件的路径，或者如果文件在工作目录中，则为文件名
access_mode	一个字符串值，用于确定文件的打开方式
buffering	用于可选行缓冲的整数值

在存储、读取或传输数据时，操作系统文件的处理既必要又简单，使用Python尤为如此。与其他语言中需要复杂的读写对象来进行文件输入输出不同，Python简化了这一过程，只需使用打开、读写和关闭文件的命令。本节介绍了Python如何与操作系统上的文件进行交互。

## 第30.1节：文件模式

你可以使用不同的模式打开文件，这些模式由mode参数指定。包括以下几种：

- 'r' - 读取模式。默认模式。它只允许你读取文件，不能修改文件。使用此模式时，文件必须存在。
- 'w' - 写入模式。如果文件不存在，将创建新文件；否则会清空文件并允许你写入内容。
- 'a' - 追加模式。它会将数据写入文件末尾。不会清空文件，且文件必须存在才能使用此模式。
- 'rb' - 二进制读取模式。类似于 r，但强制以二进制模式读取。这也是默认选择。
- 'r+' - 读写模式。允许你同时读写文件，无需分别使用 r 和 w。
- 'rb+' - 二进制读写模式。与 r+ 相同，但数据以二进制形式处理。
- 'wb' - 二进制写入模式。与 w 相同，但数据以二进制形式处理。
- 'w+' - 读写模式。与 r+ 完全相同，但如果文件不存在，会创建新文件。  
否则，文件将被覆盖。
- 'wb+' - 二进制读写模式。与 w+ 相同，但数据以二进制形式处理。
- 'ab' - 二进制追加模式。类似于 a，但数据以二进制形式处理。
- 'a+' - 追加和读取模式。类似于 w+，因为如果文件不存在，它会创建一个新文件。  
否则，如果文件存在，文件指针将位于文件末尾。
- 'ab+' - 以二进制追加和读取模式。与 a+ 相同，只是数据是二进制的。

```
with open(filename, 'r') as f:  
    f.read()  
with open(filename, 'w') as f:  
    f.write(filedata)  
with open(filename, 'a') as f:  
    f.write('\n' + newdata)
```

	r	r+	w	w+	a	a+
读取	✓	✓	X	✓	X	✓

# Chapter 30: Files & Folders I/O

## Parameter

	Details
filename	the path to your file or, if the file is in the working directory, the filename of your file
access_mode	a string value that determines how the file is opened

buffering an integer value used for optional line buffering

When it comes to storing, reading, or communicating data, working with the files of an operating system is both necessary and easy with Python. Unlike other languages where file input and output requires complex reading and writing objects, Python simplifies the process only needing commands to open, read/write and close the file. This topic explains how Python can interface with files on the operating system.

## Section 30.1: File modes

There are different modes you can open a file with, specified by the mode parameter. These include:

- 'r' - reading mode. The default. It allows you only to read the file, not to modify it. When using this mode the file must exist.
- 'w' - writing mode. It will create a new file if it does not exist, otherwise will erase the file and allow you to write to it.
- 'a' - append mode. It will write data to the end of the file. It does not erase the file, and the file must exist for this mode.
- 'rb' - reading mode in binary. This is similar to r except that the reading is forced in binary mode. This is also a default choice.
- 'r+' - reading mode plus writing mode at the same time. This allows you to read and write into files at the same time without having to use r and w.
- 'rb+' - reading and writing mode in binary. The same as r+ except the data is in binary
- 'wb' - writing mode in binary. The same as w except the data is in binary.
- 'w+' - writing and reading mode. The exact same as r+ but if the file does not exist, a new one is made. Otherwise, the file is overwritten.
- 'wb+' - writing and reading mode in binary mode. The same as w+ but the data is in binary.
- 'ab' - appending in binary mode. Similar to a except that the data is in binary.
- 'a+' - appending and reading mode. Similar to w+ as it will create a new file if the file does not exist. Otherwise, the file pointer is at the end of the file if it exists.
- 'ab+' - appending and reading mode in binary. The same as a+ except that the data is in binary.

```
with open(filename, 'r') as f:  
    f.read()  
with open(filename, 'w') as f:  
    f.write(filedata)  
with open(filename, 'a') as f:  
    f.write('\n' + newdata)
```

	r	r+	w	w+	a	a+
Read	✓	✓	X	✓	X	✓

写入	x	✓	✓	✓	✓	✓	✓
创建文件	x	x	✓	✓	✓	✓	✓
删除文件	x	x	✓	✓	x	x	
初始位置	开始	开始	开始	开始	结束	结束	

Python 3 增加了一种新的独占创建模式，这样你就不会意外截断或覆盖已有文件。

- 'x' - 以独占创建模式打开，如果文件已存在则会引发FileExistsError
- 'xb' - 以独占创建的二进制写入模式打开。与 x 相同，只是数据为二进制。
- 'x+' - 读写模式。类似于 w+，因为如果文件不存在会创建新文件。否则，会引发FileExistsError。
- 'xb+' - 写入和读取模式。与 x+ 完全相同，但数据为二进制。

	x	x+
读取	x	✓
写入	✓	✓
创建文件	✓	✓
删除文件	x	x
初始位置	起始	起始

允许你以更符合 Python 风格的方式编写文件打开代码：

Python 3.x 版本 ≥ 3.3

```
try:
    with open("fname", "r") as fout:
        # 处理已打开的文件
except FileExistsError:
    # 这里处理你的错误
```

在 Python 2 中，你会这样做

Python 2.x 版本 ≥ 2.0

```
import os.path
if os.path.isfile(fname):
    with open("fname", "w") as fout:
        # 处理已打开的文件
else:
    # 这里处理你的错误
```

## 第30.2节：逐行读取文件

逐行遍历文件的最简单方法：

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

readline() 允许对逐行迭代进行更细粒度的控制。下面的示例与上面的示例等价：

```
with open('myfile.txt', 'r') as fp:
    while True:
        cur_line = fp.readline()
```

Write	x	✓	✓	✓	✓	✓
Creates file	x	x	✓	✓	✓	✓
Erases file	x	x	✓	✓	x	x
Initial position	Start	Start	Start	Start	End	End

Python 3 added a new mode for exclusive creation so that you will not accidentally truncate or overwrite an existing file.

- 'x' - open for exclusive creation, will raise FileExistsError if the file already exists
- 'xb' - open for exclusive creation writing mode in binary. The same as x except the data is in binary.
- 'x+' - reading and writing mode. Similar to w+ as it will create a new file if the file does not exist. Otherwise, will raise FileExistsError.
- 'xb+' - writing and reading mode. The exact same as x+ but the data is binary

	x	x+
Read	x	✓
Write	✓	✓
Creates file	✓	✓
Erases file	x	x
Initial position	Start	Start

Allow one to write your file open code in a more pythonic manner:

Python 3.x Version ≥ 3.3

```
try:
    with open("fname", "r") as fout:
        # Work with your open file
except FileExistsError:
    # Your error handling goes here
```

In Python 2 you would have done something like

Python 2.x Version ≥ 2.0

```
import os.path
if os.path.isfile(fname):
    with open("fname", "w") as fout:
        # Work with your open file
else:
    # Your error handling goes here
```

## Section 30.2: Reading a file line-by-line

The simplest way to iterate over a file line-by-line:

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

readline() allows for more granular control over line-by-line iteration. The example below is equivalent to the one above:

```
with open('myfile.txt', 'r') as fp:
    while True:
        cur_line = fp.readline()
```

```
# 如果结果是空字符串
if cur_line == '':
    # 我们已经到达文件末尾
    break
print(cur_line)
```

同时使用 for 循环迭代器和 readline() 被认为是不好的做法。

更常见的是使用readlines()方法来存储文件行的可迭代集合：

```
with open("myfile.txt", "r") as fp:
lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

这将打印以下内容：

```
第0行：你好
第1行：世界
```

## 第30.3节：迭代文件（递归）

要迭代所有文件，包括子目录中的文件，使用 os.walk：

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print(root, filename)
```

root\_dir 可以是 ".", 表示从当前目录开始，或者任何其他路径作为起点。

Python 3.x 版本 ≥ 3.5

如果你还想获取文件的信息，可以使用更高效的方法 os.scandir，示例如下：

```
for entry in os.scandir(path):
    if not entry.name.startswith('.') and entry.is_file():
        print(entry.name)
```

## 第30.4节：获取文件的全部内容

文件输入输出的首选方法是使用with关键字。这将确保在读取或写入完成后文件句柄被关闭。

```
with open('myfile.txt') as in_file:
content = in_file.read()

print(content)
```

或者，为了手动关闭文件，你可以不使用with，而是自己调用close：

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
```

```
# If the result is an empty string
if cur_line == '':
    # We have reached the end of the file
    break
print(cur_line)
```

Using the for loop iterator and readline() together is considered bad practice.

More commonly, the readlines() method is used to store an iterable collection of the file's lines:

```
with open("myfile.txt", "r") as fp:
lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

This would print the following:

```
Line 0: hello
Line 1: world
```

## Section 30.3: Iterate files (recursively)

To iterate all files, including in sub directories, use os.walk:

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print(root, filename)
```

root\_dir can be "." to start from current directory, or any other path to start from.

Python 3.x Version ≥ 3.5

If you also wish to get information about the file, you may use the more efficient method [os.scandir](#) like so:

```
for entry in os.scandir(path):
    if not entry.name.startswith('.') and entry.is_file():
        print(entry.name)
```

## Section 30.4: Getting the full contents of a file

The preferred method of file i/o is to use the [with](#) keyword. This will ensure the file handle is closed once the reading or writing has been completed.

```
with open('myfile.txt') as in_file:
content = in_file.read()

print(content)
```

or, to handle closing the file manually, you can forgo [with](#) and simply call close yourself:

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
```

```
in_file.close()
```

请记住，如果不使用with语句，遇到意外

异常时，可能会意外地保持文件打开，情况如下：

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close() # 这行代码永远不会被执行
```

## 第30.5节：写入文件

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

如果你打开myfile.txt，你会看到文件内容是：

Line 1Line 2Line 3Line 4

Python 不会自动添加换行符，你需要手动添加：

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

Line 1  
Line 2  
Line 3  
Line 4

写入以文本模式（默认）打开的文件时，不要使用os.linesep作为行终止符；应使用代替。

如果你想指定编码，只需在open函数中添加encoding参数：

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

也可以使用 print 语句写入文件。Python 2 和 Python 3 的实现方式不同，但概念是相同的，即你可以将本来输出到屏幕的内容发送到文件中。

Python 3.x 版本 ≥ 3.0

```
with open('fred.txt', 'w') as outfile:
    s = "我还没死！"
    print(s) # 输出到标准输出
    print(s, file = outfile) # 输出到 outfile

#注意：可以同时指定 file 参数并输出到屏幕
#方法是确保 file 最终为 None 值，直接或通过变量实现
```

```
in_file.close()
```

Keep in mind that without using a **with** statement, you might accidentally keep the file open in case an unexpected exception arises like so:

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close() # This will never be called
```

## Section 30.5: Writing to a file

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

If you open myfile.txt, you will see that its contents are:

Line 1Line 2Line 3Line 4

Python doesn't automatically add line breaks, you need to do that manually:

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1\n")
    f.write("Line 2\n")
    f.write("Line 3\n")
    f.write("Line 4\n")
```

Line 1  
Line 2  
Line 3  
Line 4

Do not use **os.linesep** as a line terminator when writing files opened in text mode (the default); use **\n** instead.

If you want to specify an encoding, you simply add the **encoding** parameter to the **open** function:

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

It is also possible to use the **print** statement to write to a file. The mechanics are different in Python 2 vs Python 3, but the concept is the same in that you can take the output that would have gone to the screen and send it to a file instead.

Python 3.x Version ≥ 3.0

```
with open('fred.txt', 'w') as outfile:
    s = "I'm Not Dead Yet!"
    print(s) # writes to stdout
    print(s, file = outfile) # writes to outfile
```

#Note: it is possible to specify the file parameter AND write to the screen  
#by making sure file ends up with a None value either directly or via a variable

```
myfile = None
print(s, file = myfile) # 输出到标准输出
print(s, file = None)    # 输出到标准输出
```

在 Python 2 中，你会这样做

```
Python 2.x 版本 ≥ 2.0
outfile = open('fred.txt', 'w')
s = "我还没死！"
print s    # 输出到标准输出
print >> outfile, s  # 输出到 outfile
```

与使用 write 函数不同，print 函数会自动添加换行符。

## 第30.6节：检查文件或路径是否存在

采用EAFP编码风格并尝试打开它。

```
import errno

try:
    with open(path) as f:
        # 文件存在
except IOError as e:
    # 如果不是 ENOENT (没有此文件或目录)，则抛出异常
    if e.errno != errno.ENOENT:
        raise
    # 没有此文件或目录
```

如果另一个进程在检查和使用之间删除了文件，这也将避免竞态条件。此竞态条件可能发生在以下情况：

- 使用os模块：

```
import os
os.path.isfile('/path/to/some/file.txt')
```

Python 3.x 版本 ≥ 3.4

- 使用 pathlib:

```
import pathlib
path = pathlib.Path('/path/to/some/file.txt')
if path.is_file():
    ...
```

要检查给定路径是否存在，可以按照上述EAFP方法，或者显式检查路径：

```
import os
path = "/home/myFiles/directory1"

if os.path.exists(path):
    ## 执行操作
```

```
myfile = None
print(s, file = myfile) # writes to stdout
print(s, file = None)    # writes to stdout
```

In Python 2 you would have done something like

```
Python 2.x Version ≥ 2.0
outfile = open('fred.txt', 'w')
s = "I'm Not Dead Yet!"
print s    # writes to stdout
print >> outfile, s  # writes to outfile
```

Unlike using the write function, the print function does automatically add line breaks.

## Section 30.6: Check whether a file or path exists

Employ the [EAFP](#) coding style and `try` to open it.

```
import errno

try:
    with open(path) as f:
        # File exists
except IOError as e:
    # Raise the exception if it is not ENOENT (No such file or directory)
    if e.errno != errno.ENOENT:
        raise
    # No such file or directory
```

This will also avoid race-conditions if another process deleted the file between the check and when it is used. This race condition could happen in the following cases:

- Using the os module:

```
import os
os.path.isfile('/path/to/some/file.txt')
```

Python 3.x Version ≥ 3.4

- Using pathlib:

```
import pathlib
path = pathlib.Path('/path/to/some/file.txt')
if path.is_file():
    ...
```

To check whether a given path exists or not, you can follow the above EAFP procedure, or explicitly check the path:

```
import os
path = "/home/myFiles/directory1"

if os.path.exists(path):
    ## Do stuff
```

## 第30.7节：使用mmap进行随机文件访问

使用mmap模块允许用户通过将文件映射到内存中，随机访问文件中的位置。这是使用普通文件操作的替代方法。

```
import mmap

使用open('filename.ext', 'r') 作为fd:
    # 0 : 映射整个文件
mm = mmap.mmap(fd.fileno(), 0)

# 打印索引5到10的字符
print mm[5:10]

# 打印从mm当前位置开始的一行
print mm.readline()

# 向第5个索引写入一个字符
mm[5] = 'a'

# 将mm的位置返回到文件开头
mm.seek(0)

# 关闭 mmap 对象
mm.close()
```

## 第30.8节：替换文件中的文本

```
import fileinput

replacements = {'查找1': '替换1',
                '查找2': '替换2'}

for line in fileinput.input('filename.txt', inplace=True):
    for search_for in replacements:
        replace_with = replacements[search_for]
        line = line.replace(search_for, replace_with)
    print(line, end="")
```

## 第30.9节：检查文件是否为空

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

或者

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

但是，如果文件不存在，这两种方法都会抛出异常。为了避免必须捕获此类错误，可以这样做：

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

这将返回一个bool值。

## Section 30.7: Random File Access Using mmap

Using the `mmap` module allows the user to randomly access locations in a file by mapping the file into memory. This is an alternative to using normal file operations.

```
import mmap

with open('filename.ext', 'r') as fd:
    # 0: map the whole file
    mm = mmap.mmap(fd.fileno(), 0)

    # print characters at indices 5 through 10
    print mm[5:10]

    # print the line starting from mm's current position
    print mm.readline()

    # write a character to the 5th index
    mm[5] = 'a'

    # return mm's position to the beginning of the file
    mm.seek(0)

    # close the mmap object
    mm.close()
```

## Section 30.8: Replacing text in a file

```
import fileinput

replacements = {'Search1': 'Replace1',
                'Search2': 'Replace2'}

for line in fileinput.input('filename.txt', inplace=True):
    for search_for in replacements:
        replace_with = replacements[search_for]
        line = line.replace(search_for, replace_with)
    print(line, end="")
```

## Section 30.9: Checking if a file is empty

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

or

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

However, both will throw an exception if the file does not exist. To avoid having to catch such an error, do this:

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

which will return a `bool` value.

## 第30.10节：读取文件中某个范围的行

假设你只想遍历文件中的某些特定行

你可以利用`itertools`来实现

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # 在这里执行某些操作
```

这将读取第13行到第20行，因为Python的索引从0开始。所以第1行的索引是0

也可以通过使用`next()`关键字读取一些额外的行。

当你将文件对象作为可迭代对象使用时，请不要使用`readline()`语句，因为这两种遍历文件的技术不能混用

## Section 30.10: Read a file between a range of lines

So let's suppose you want to iterate only between some specific lines of a file

You can make use of `itertools` for that

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # do something here
```

This will read through the lines 13 to 20 as in python indexing starts from 0. So line number 1 is indexed as 0

As can also read some extra lines by making use of the `next()` keyword here.

And when you are using the file object as an iterable, please don't use the `readline()` statement here as the two techniques of traversing a file are not to be mixed together

## 第30.11节：复制目录树

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

目标目录必须不存在。

## 第30.12节：将一个文件的内容复制到另一个文件

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- 使用`shutil`模块：

```
import shutil
shutil.copyfile(src, dst)
```

## Section 30.11: Copy a directory tree

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

The destination directory **must not exist** already.

## Section 30.12: Copying contents of one file to a different file

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- Using the `shutil` module:

```
import shutil
shutil.copyfile(src, dst)
```

# 第31章：os.path

该模块实现了一些对路径名有用的函数。路径参数可以作为字符串或字节传递。建议应用程序将文件名表示为（Unicode）字符字符串。

## 第31.1节：连接路径

要连接两个或多个路径组件，首先导入Python的os模块，然后使用以下方法：

```
import os  
os.path.join('a', 'b', 'c')
```

使用os.path的优点是它允许代码在所有操作系统上保持兼容，因为它使用适合运行平台的分隔符。

例如，该命令在Windows上的结果将是：

```
>>> os.path.join('a', 'b', 'c')  
'a\b\c'
```

在 Unix 操作系统中：

```
>>> os.path.join('a', 'b', 'c')  
'a/b/c'
```

## 第31.2节：路径组件操作

要从路径中分离出一个组件：

```
>>> p = os.path.join(os.getcwd(), 'foo.txt')  
>>> p  
'/Users/csaftoiu/tmp/foo.txt'  
>>> os.path.dirname(p)  
'/Users/csaftoiu/tmp'  
>>> os.path.basename(p)  
'foo.txt'  
>>> os.path.split(os.getcwd())  
('/Users/csaftoiu/tmp', 'foo.txt')  
>>> os.path.splitext(os.path.basename(p))  
('foo', '.txt')
```

## 第31.3节：获取父目录

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

## 第31.4节：如果给定路径存在

检查给定路径是否存在

```
path = '/home/john/temp'  
os.path.exists(path)  
#如果路径不存在或路径是一个损坏的符号链接，则返回False
```

# Chapter 31: os.path

This module implements some useful functions on pathnames. The path parameters can be passed as either strings, or bytes. Applications are encouraged to represent file names as (Unicode) character strings.

## Section 31.1: Join Paths

To join two or more path components together, firstly import os module of python and then use following:

```
import os  
os.path.join('a', 'b', 'c')
```

The advantage of using os.path is that it allows code to remain compatible over all operating systems, as this uses the separator appropriate for the platform it's running on.

For example, the result of this command on Windows will be:

```
>>> os.path.join('a', 'b', 'c')  
'a\b\c'
```

In an Unix OS:

```
>>> os.path.join('a', 'b', 'c')  
'a/b/c'
```

## Section 31.2: Path Component Manipulation

To split one component off of the path:

```
>>> p = os.path.join(os.getcwd(), 'foo.txt')  
>>> p  
'/Users/csaftoiu/tmp/foo.txt'  
>>> os.path.dirname(p)  
'/Users/csaftoiu/tmp'  
>>> os.path.basename(p)  
'foo.txt'  
>>> os.path.split(os.getcwd())  
('/Users/csaftoiu/tmp', 'foo.txt')  
>>> os.path.splitext(os.path.basename(p))  
('foo', '.txt')
```

## Section 31.3: Get the parent directory

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

## Section 31.4: If the given path exists

to check if the given path exists

```
path = '/home/john/temp'  
os.path.exists(path)  
#this returns false if path doesn't exist or if the path is a broken symbolic link
```

## 第31.5节：检查给定路径是否是目录、文件、符号链接、挂载点等

检查给定路径是否是目录

```
dirname = '/home/john/python'  
os.path.isdir(dirname)
```

检查给定路径是否是文件

```
filename = dirname + 'main.py'  
os.path.isfile(filename)
```

检查给定路径是否是符号链接

```
symlink = dirname + 'some_sym_link'  
os.path.islink(symlink)
```

检查给定路径是否为挂载点

```
mount_path = '/home'  
os.path.ismount(mount_path)
```

## Section 31.5: check if the given path is a directory, file, symbolic link, mount point etc

to check if the given path is a directory

```
dirname = '/home/john/python'  
os.path.isdir(dirname)
```

to check if the given path is a file

```
filename = dirname + 'main.py'  
os.path.isfile(filename)
```

to check if the given path is [symbolic link](#)

```
symlink = dirname + 'some_sym_link'  
os.path.islink(symlink)
```

to check if the given path is a [mount point](#)

```
mount_path = '/home'  
os.path.ismount(mount_path)
```

## 第31.6节：从相对路径获取绝对路径

使用 `os.path.abspath` :

```
>>> os.getcwd()  
'/Users/csaftoiu/tmp'  
>>> os.path.abspath('foo')  
'/Users/csaftoiu/tmp/foo'  
>>> os.path.abspath('../foo')  
'/Users/csaftoiu/foo'  
>>> os.path.abspath('/foo')  
'/foo'
```

## Section 31.6: Absolute Path from Relative Path

Use `os.path.abspath`:

```
>>> os.getcwd()  
'/Users/csaftoiu/tmp'  
>>> os.path.abspath('foo')  
'/Users/csaftoiu/tmp/foo'  
>>> os.path.abspath('../foo')  

```

# 第32章：可迭代对象和迭代器

## 第32.1节：迭代器 vs 可迭代对象 vs 生成器

可迭代对象是能够返回迭代器的对象。任何具有状态且包含 `__iter__` 方法并返回迭代器的对象都是可迭代的。它也可以是一个无状态且实现了 `__getitem__` 方法的对象。- 该方法可以接受从零开始的索引，当索引不再有效时会抛出 `IndexError` 异常。

Python 的 `str` 类是一个 `__getitem__` 可迭代对象的例子。

迭代器 (Iterator) 是一个对象，当你在某个对象上调用 `next(*object*)` 时，它会产生序列中的下一个值。此外，任何具有 `__next__` 方法的对象都是迭代器。迭代器在耗尽后会引发 `StopIteration` 异常，且此时不能再次使用。

可迭代类：

可迭代类定义了 `__iter__` 和 `__next__` 方法。以下是一个可迭代类的示例：

```
class MyIterable:  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        #代码  
  
#在较早版本的 Python 中的经典可迭代对象，仍然支持 __getitem__ ...  
class MySequence:  
  
    def __getitem__(self, index):  
  
        if (condition):  
            raise IndexError  
        return (item)  
  
#可以通过使用 iter(MySequence()) 生成一个普通的 `iterator` 实例
```

尝试实例化 `collections` 模块中的抽象类以更好地理解这一点。

示例：

```
Python 2.x 版本 ≥ 2.3  
  
import collections  
>>> collections.Iterator()  
>>> TypeError: 无法实例化带有抽象方法 next 的抽象类 Iterator  
  
Python 3.x 版本 ≥ 3.0  
  
>>> TypeError: 无法实例化带有抽象方法 __next__ 的抽象类 Iterator
```

通过以下方式处理 Python 2 中可迭代类的 Python 3 兼容性：

```
Python 2.x 版本 ≥ 2.3  
  
class MyIterable(object): #或者 collections.Iterator, 我推荐使用后者....  
  
....
```

# Chapter 32: Iterables and Iterators

## Section 32.1: Iterator vs Iterable vs Generator

An **iterable** is an object that can return an **iterator**. Any object with state that has an `__iter__` method and returns an iterator is an iterable. It may also be an object *without* state that implements a `__getitem__` method. - The method can take indices (starting from zero) and raise an `IndexError` when the indices are no longer valid.

Python's `str` class is an example of a `__getitem__` iterable.

An **Iterator** is an object that produces the next value in a sequence when you call `next(*object*)` on some object. Moreover, any object with a `__next__` method is an iterator. An iterator raises `StopIteration` after exhausting the iterator and *cannot* be re-used at this point.

**Iterable classes:**

Iterable classes define an `__iter__` and a `__next__` method. Example of an iterable class:

```
class MyIterable:  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        #code  
  
#Classic iterable object in older versions of python, __getitem__ is still supported...  
class MySequence:  
  
    def __getitem__(self, index):  
  
        if (condition):  
            raise IndexError  
        return (item)  
  
#Can produce a plain `iterator` instance by using iter(MySequence())
```

Trying to instantiate the abstract class from the `collections` module to better see this.

Example:

```
Python 2.x Version ≥ 2.3  
  
import collections  
>>> collections.Iterator()  
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods next  
  
Python 3.x Version ≥ 3.0  
  
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods __next__
```

Handle Python 3 compatibility for iterable classes in Python 2 by doing the following:

```
Python 2.x Version ≥ 2.3  
  
class MyIterable(object): #or collections.Iterator, which I'd recommend....  
  
....
```

```

def __iter__(self):
    return self

def next(self): #代码
    _next_ = next

```

这两者现在都是迭代器，可以用循环遍历：

```

ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #代码
for (item) in (ex2): #代码

```

生成器是创建迭代器的简单方法。生成器是迭代器，而迭代器是可迭代对象。

## 第32.2节：逐个提取值

从内置的`iter()`开始获取可迭代对象的`iterator`，使用`next()`逐个获取元素，直到抛出`StopIteration`表示结束：

```

s = {1, 2}    # 或列表或生成器甚至迭代器
i = iter(s)  # 获取迭代器
a = next(i)  # a = 1
b = next(i)  # b = 2
c = next(i)  # 引发 StopIteration

```

## 第32.3节：遍历整个可迭代对象

```

s = {1, 2, 3}

# 获取s中的每个元素
for a in s:
    print a # 依次打印1, 然后2, 然后3

# 复制到列表中
l1 = list(s) # l1 = [1, 2, 3]

# 使用列表推导式
l2 = [a * 2 for a in s if a > 2] # l2 = [6]

```

## 第32.4节：验证可迭代对象中只有一个元素

使用解包提取第一个元素，并确保它是唯一的：

```

a, = 可迭代对象

def foo():
    yield 1

a, = foo() # a = 1

nums = [1, 2, 3]
a, = nums # ValueError: too many values to unpack

```

```

def __iter__(self):
    return self

def next(self): #code
    __next__ = next

```

Both of these are now iterators and can be looped through:

```

ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code

```

**Generators** are simple ways to create iterators. A generator *is* an iterator and an iterator is an iterable.

## Section 32.2: Extract values one by one

Start with `iter()` built-in to get `iterator` over iterable and use `next()` to get elements one by one until `StopIteration` is raised signifying the end:

```

s = {1, 2}    # or list or generator or even iterator
i = iter(s)  # get iterator
a = next(i)  # a = 1
b = next(i)  # b = 2
c = next(i)  # raises StopIteration

```

## Section 32.3: Iterating over entire iterable

```

s = {1, 2, 3}

# get every element in s
for a in s:
    print a # prints 1, then 2, then 3

# copy into list
l1 = list(s) # l1 = [1, 2, 3]

# use list comprehension
l2 = [a * 2 for a in s if a > 2] # l2 = [6]

```

## Section 32.4: Verify only one element in iterable

Use unpacking to extract the first element and ensure it's the only one:

```

a, = iterable

def foo():
    yield 1

a, = foo() # a = 1

nums = [1, 2, 3]
a, = nums # ValueError: too many values to unpack

```

## 第32.5节：什么可以是可迭代对象

可迭代对象可以是任何能够逐个接收元素、只能向前遍历的对象。Python内置集合是可迭代的：

```
[1, 2, 3]      # 列表, 遍历元素  
(1, 2, 3)      # 元组  
{1, 2, 3}      # 集合  
{1: 2, 3: 4}  # 字典, 遍历键
```

生成器返回可迭代对象：

```
def foo(): # foo 还不能迭代...  
    yield 1  
  
res = foo() # ...但 res 已经是可迭代的了
```

## 第32.6节：迭代器不是可重入的！

```
def gen():  
    yield 1  
  
iterable = gen()  
for a in iterable:  
    print a  
  
# iterable 的第一个元素是什么？现在无法获取。  
# 只能获取一个新的迭代器  
gen()
```

## Section 32.5: What can be iterable

**Iterable** can be anything for which items are received *one by one, forward only*. Built-in Python collections are iterable:

```
[1, 2, 3]      # list, iterate over items  
(1, 2, 3)      # tuple  
{1, 2, 3}      # set  
{1: 2, 3: 4}  # dict, iterate over keys
```

Generators return iterables:

```
def foo(): # foo isn't iterable yet...  
    yield 1  
  
res = foo() # ...but res already is
```

## Section 32.6: Iterator isn't reentrant!

```
def gen():  
    yield 1  
  
iterable = gen()  
for a in iterable:  
    print a  
  
# What was the first item of iterable? No way to get it now.  
# Only to get a new iterator  
gen()
```

# 第33章：函数

参数	详细信息
<code>arg1, ..., argN</code>	常规参数
<code>*args</code>	未命名的位置参数
<code>kw1, ..., kwN</code>	仅限关键字参数
<code>**kwargs</code>	其余的关键字参数

Python中的函数提供了有组织、可重用和模块化的代码，用于执行一组特定的操作。函数简化了编码过程，防止冗余逻辑，并使代码更易于理解。本主题介绍了Python中函数的声明和使用。

Python有许多内置函数，如`print()`、`input()`、`len()`。除了内置函数，你还可以创建自己的函数来完成更具体的任务——这些称为用户定义函数。

## 第33.1节：定义和调用简单函数

使用`def`语句是Python中定义函数最常见的方式。该语句是一种所谓的单条款复合语句，语法如下：

```
def 函数名(参数):  
    声明
```

`function_name` 被称为函数的标识符。由于函数定义是一个可执行语句，其执行将函数名绑定到函数对象，之后可以使用该标识符调用该函数对象。

`parameters` 是一个可选的标识符列表，当函数被调用时，这些标识符会绑定到作为参数传入的值。函数可以有任意数量的参数，参数之间用逗号分隔。

`statement(s)` – 也称为函数体–是一个非空的语句序列，每次函数被调用时都会执行。这意味着函数体不能为空，就像任何一个缩进块一样。

这里是一个简单函数定义的示例，其目的是每次调用时打印Hello：

```
def greet():  
    print("Hello")
```

现在让我们调用已定义的`greet()`函数：

```
greet()  
# 输出: Hello
```

这是另一个函数定义的示例，该函数接受一个参数，并在每次调用函数时显示传入的值：

```
def greet_two(greeting):  
    print(greeting)
```

之后必须用一个参数调用`greet_two()`函数：

```
greet_two("Howdy")  
# 输出: Howdy
```

# Chapter 33: Functions

Parameter	Details
<code>arg1, ..., argN</code>	Regular arguments
<code>*args</code>	Unnamed positional arguments
<code>kw1, ..., kwN</code>	Keyword-only arguments
<code>**kwargs</code>	The rest of keyword arguments

Functions in Python provide organized, reusable and modular code to perform a set of specific actions. Functions simplify the coding process, prevent redundant logic, and make the code easier to follow. This topic describes the declaration and utilization of functions in Python.

Python has many *built-in functions* like `print()`, `input()`, `len()`. Besides built-ins you can also create your own functions to do more specific jobs—these are called *user-defined functions*.

## Section 33.1: Defining and calling simple functions

Using the `def` statement is the most common way to define a function in python. This statement is a so called *single clause compound statement* with the following syntax:

```
def function_name(parameters):  
    statement(s)
```

`function_name` is known as the *identifier* of the function. Since a function definition is an executable statement its execution *binds* the function name to the function object which can be called later on using the identifier.

`parameters` is an optional list of identifiers that get bound to the values supplied as arguments when the function is called. A function may have an arbitrary number of arguments which are separated by commas.

`statement(s)` – also known as the *function body* – are a nonempty sequence of statements executed each time the function is called. This means a function body cannot be empty, just like any *indented block*.

Here's an example of a simple function definition which purpose is to print Hello each time it's called:

```
def greet():  
    print("Hello")
```

Now let's call the defined `greet()` function:

```
greet()  
# Out: Hello
```

That's another example of a function definition which takes one single argument and displays the passed in value each time the function is called:

```
def greet_two(greeting):  
    print(greeting)
```

After that the `greet_two()` function must be called with an argument:

```
greet_two("Howdy")  
# Out: Howdy
```

你也可以给该函数参数一个默认值：

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

现在你可以在不传入参数的情况下调用该函数：

```
greet_two()
# Output: Howdy
```

你会注意到，与许多其他语言不同，你不需要显式声明函数的返回类型。

Python 函数可以通过return关键字返回任何类型的值。一个函数可以返回任意数量的不同类型！

```
def many_types(x):
    if x < 0:
        return "Hello!"
    else:
        return 0

print(many_types(1))
print(many_types(-1))
```

```
# Output:
0
Hello!
```

只要调用者正确处理，这段代码就是完全有效的Python代码。

一个没有return语句就执行到末尾的函数，总是返回None：

```
def do_nothing():
    pass

print(do_nothing())
# Output: None
```

如前所述，函数定义必须有函数体，即一段非空的语句序列。

因此，pass语句被用作函数体，它是一个空操作-执行时不会发生任何事情。它的含义就是跳过。当语法上需要语句但不需要执行任何代码时，它作为占位符非常有用。

## 第33.2节：定义带有任意数量参数的函数

任意数量的位置参数：

定义一个能够接受任意数量参数的函数，可以通过在某个参数前加上\*来实现

```
def func(*args):
    # args will be a tuple containing all values that are passed in
    for i in args:
        print(i)

func(1, 2, 3) # Using 3 arguments
```

Also you can give a default value to that function argument:

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

Now you can call the function without giving a value:

```
greet_two()
# Output: Howdy
```

You'll notice that unlike many other languages, you do not need to explicitly declare a return type of the function. Python functions can return values of any type via the `return` keyword. One function can return any number of different types!

```
def many_types(x):
    if x < 0:
        return "Hello!"
    else:
        return 0

print(many_types(1))
print(many_types(-1))

# Output:
0
Hello!
```

As long as this is handled correctly by the caller, this is perfectly valid Python code.

A function that reaches the end of execution without a return statement will always return `None`:

```
def do_nothing():
    pass

print(do_nothing())
# Output: None
```

As mentioned previously a function definition must have a function body, a nonempty sequence of statements. Therefore the `pass` statement is used as function body, which is a null operation – when it is executed, nothing happens. It does what it means, it skips. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed.

## Section 33.2: Defining a function with an arbitrary number of arguments

**Arbitrary number of positional arguments:**

Defining a function capable of taking an arbitrary number of arguments can be done by prefixing one of the arguments with a \*

```
def func(*args):
    # args will be a tuple containing all values that are passed in
    for i in args:
        print(i)

func(1, 2, 3) # Calling it with 3 arguments
```

```

# 输出: 1
#     2
#     3

list_of_arg_values = [1, 2, 3]
func(*list_of_arg_values) # 使用值列表调用, * 展开列表
# 输出: 1
#     2
#     3

func() # 不带参数调用
# 无输出

```

你不能为args提供默认值，例如func(\*args=[1, 2, 3])会引发语法错误（甚至无法编译）。

调用函数时**不能**通过名称提供这些参数，例如func(\*args=[1, 2, 3])会引发 **TypeError**。

但如果你的参数已经在一个数组（或其他Iterable）中，你可以这样调用函数：  
func(\*my\_stuff)。

这些参数 (\*args) 可以通过索引访问，例如args[0]将返回第一个参数

### 任意数量的关键字参数

你可以通过在定义中为参数名前加上两个\*来接收任意数量的命名参数：

```

def func(**kwargs):
    # kwargs 将是一个包含名称作为键和值作为值的字典
    for name, value in kwargs.items():
        print(name, value)

func(value1=1, value2=2, value3=3) # 使用3个参数调用
# 输出: value1 1
#     value2 2
#     value3 3

func() # 不带参数调用
# 无输出

my_dict = {'foo': 1, 'bar': 2}
func(**my_dict) # 使用字典调用
# 输出: foo 1
#     bar 2

```

你不能在没有名字的情况下提供这些，例如func(1, 2, 3)会引发**TypeError**。

`kwargs` 是一个普通的原生 Python 字典。例如，`args['value1']` 将返回参数 `value1` 的值。请事先确认存在该参数，否则会引发 **KeyError** 异常。

### 警告

你可以将这些与其他可选和必需参数混合使用，但定义中的顺序很重要。

位置/关键字参数先出现。（必需参数）。

然后是 任意数量的 `*arg` 参数。（可选）。

```

# Out: 1
#     2
#     3

list_of_arg_values = [1, 2, 3]
func(*list_of_arg_values) # Calling it with list of values, * expands the list
# Out: 1
#     2
#     3

func() # Calling it without arguments
# No Output

```

You **can't** provide a default for args, for example `func(*args=[1, 2, 3])` will raise a syntax error (won't even compile).

You **can't** provide these by name when calling the function, for example `func(*args=[1, 2, 3])` will raise a **TypeError**.

But if you already have your arguments in an array (or any other Iterable), you **can** invoke your function like this:  
func(\*my\_stuff)。

These arguments (\*args) can be accessed by index, for example `args[0]` will return the first argument

### Arbitrary number of keyword arguments

You can take an arbitrary number of arguments with a name by defining an argument in the definition with **two \*** in front of it:

```

def func(**kwargs):
    # kwargs will be a dictionary containing the names as keys and the values as values
    for name, value in kwargs.items():
        print(name, value)

func(value1=1, value2=2, value3=3) # Calling it with 3 arguments
# Out: value1 1
#     value2 2
#     value3 3

func() # Calling it without arguments
# No Out put

my_dict = {'foo': 1, 'bar': 2}
func(**my_dict) # Calling it with a dictionary
# Out: foo 1
#     bar 2

```

You **can't** provide these **without** names, for example `func(1, 2, 3)` will raise a **TypeError**.

`kwargs` is a plain native python dictionary. For example, `args['value1']` will give the value for argument `value1`. Be sure to check beforehand that there is such an argument or a **KeyError** will be raised.

### Warning

You can mix these with other optional and required arguments but the order inside the definition matters.

The **positional/keyword** arguments come first. (Required arguments).

Then comes the **arbitrary** `*arg` arguments. (Optional).

接着是 仅关键字参数。 (必需)。

最后是 任意关键字参数 \*\*kwargs。 (可选)。

```
# /-位置参数-/可选参数---仅关键字参数--/-可选参数-/
def func(arg1, arg2=10, *args, kwarg1, kwarg2=2, **kwargs):
    pass
```

- arg1 必须提供，否则会引发 `TypeError`。它可以作为位置参数 (`func(10)`) 或关键字参数 (`func(arg1=10)`) 提供。
- kwarg1 也必须提供，但只能作为关键字参数提供：`func(kwarg1=10)`。
- arg2 和 kwarg2 是可选的。如果要更改其值，适用与 arg1 (位置参数或关键字参数) 和 kwarg1 (仅关键字参数) 相同的规则。
- \*args 捕获额外的位置参数。但请注意，arg1 和 arg2 必须作为位置参数提供，才能将参数传递给 \*args : `func(1, 1, 1, 1)`。
- \*\*kwargs 捕获所有额外的关键字参数。在这种情况下，任何不是 arg1、arg2、kwarg1 或 kwarg2 的参数都会被捕获。例如：`func(kwarg3=10)`。
- 在 Python 3 中，可以单独使用 \* 来表示所有后续参数必须作为关键字参数指定。  
例如，Python 3.5 及更高版本中的 `math.isclose` 函数定义为 `def math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)`，这意味着前两个参数可以作为位置参数提供，但可选的第三个和第四个参数只能作为关键字参数提供。

Python 2.x 不支持仅关键字参数。此行为可以通过 `kwargs` 来模拟：

```
def func(arg1, arg2=10, **kwargs):
    try:
        kwarg1 = kwargs.pop("kwarg1")
    except KeyError:
        raise TypeError("missing required keyword-only argument: 'kwarg1'")

    kwarg2 = kwargs.pop("kwarg2", 2)
    # 函数体 ...
```

## 命名说明

可选位置参数命名为`args`和可选关键字参数命名为`kwargs`只是一个约定，你可以使用任何你喜欢的名字，但遵循这个约定是有用的，这样别人知道你在做什么，或者即使是你自己以后也能明白，所以请遵守。

## 唯一性说明

任何函数都可以定义`none`或一个`*args`和`none`或一个`**kwargs`，但不能定义多个相同的参数。另外，`*args`必须是最后一个位置参数，`**kwargs`必须是最后一个参数。尝试使用多个相同参数将导致语法错误异常。

## 关于嵌套带可选参数函数的说明

可以嵌套此类函数，通常的约定是移除代码已经处理过的项，但如果要传递参数，需要用\*前缀传递可选位置参数，用\*\*前缀传递可选关键字参数，否则`args`会作为列表或元组传递，`kwargs`会作为单个字典传递。例如：

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)
```

Then **keyword-only** arguments come next. (Required).

Finally the **arbitrary keyword** `**kwargs` come. (Optional).

```
# /-positional-/optional---keyword-only--/-optional-/
def func(arg1, arg2=10, *args, kwarg1, kwarg2=2, **kwargs):
    pass
```

- arg1 必须给定，否则会引发 `TypeError`。它可以作为位置参数 (`func(10)`) 或关键字参数 (`func(arg1=10)`) 提供。
- kwarg1 也必须给定，但只能作为关键字参数提供：`func(kwarg1=10)`。
- arg2 和 kwarg2 是可选的。如果要更改其值，适用与 arg1 (位置参数或关键字参数) 和 kwarg1 (仅关键字参数) 相同的规则。
- \*args 捕获额外的位置参数。但请注意，arg1 和 arg2 必须作为位置参数提供，才能将参数传递给 \*args : `func(1, 1, 1, 1)`。
- \*\*kwargs 捕获所有额外的关键字参数。在这种情况下，任何不是 arg1、arg2、kwarg1 或 kwarg2 的参数都会被捕获。例如：`func(kwarg3=10)`。
- 在 Python 3 中，你可以单独使用 \* 来表示所有后续参数必须作为关键字参数指定。  
例如，Python 3.5 及更高版本中的 `math.isclose` 函数在 Python 3.5 及更高版本中是通过 `def math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)` 定义的，这意味着前两个参数可以作为位置参数提供，但可选的第三个和第四个参数只能作为关键字参数提供。

Python 2.x 不支持仅关键字参数。此行为可以通过 `kwargs` 来模拟：

```
def func(arg1, arg2=10, **kwargs):
    try:
        kwarg1 = kwargs.pop("kwarg1")
    except KeyError:
        raise TypeError("missing required keyword-only argument: 'kwarg1'")

    kwarg2 = kwargs.pop("kwarg2", 2)
    # function body ...
```

## Note on Naming

The convention of naming optional positional arguments `args` and optional keyword arguments `kwargs` is just a convention you **can** use any names you like **but** it is useful to follow the convention so that others know what you are doing, *or even yourself later* so please do.

## Note on Uniqueness

Any function can be defined with **none or one** `*args` and **none or one** `**kwargs` but not with more than one of each. Also `*args` **must** be the last positional argument and `**kwargs` must be the last parameter. Attempting to use more than one of either **will** result in a Syntax Error exception.

## Note on Nesting Functions with Optional Arguments

It is possible to nest such functions and the usual convention is to remove the items that the code has already handled **but** if you are passing down the parameters you need to pass optional positional args with a `*` prefix and optional keyword args with a `**` prefix, otherwise args will be passed as a list or tuple and kwargs as a single dictionary. e.g.:

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)
```

```
def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

### 第33.3节：Lambda（内联/匿名）函数

关键字lambda创建一个包含单个表达式的内联函数。该表达式的值是函数被调用时返回的结果。

考虑以下函数：

```
def greeting():
    return "Hello"
```

当调用时：

```
print(greeting())
```

打印：

```
Hello
```

这可以写成如下的lambda函数：

```
greet_me = lambda: "Hello"
```

请参见本节底部关于将lambda赋值给变量的说明。通常，不要这样做。

这创建了一个名为greet\_me的内联函数，返回Hello。注意，使用lambda创建函数时不需要写return。冒号后面的值会自动返回。

一旦赋值给变量，就可以像普通函数一样使用：

```
print(greet_me())
```

打印：

```
Hello
```

lambda函数也可以接受参数：

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("Hello ")
```

返回字符串：

```
HELLO
```

```
def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

### Section 33.3: Lambda (Inline/Anonymous) Functions

The **lambda** keyword creates an inline function that contains a single expression. The value of this expression is what the function returns when invoked.

Consider the function:

```
def greeting():
    return "Hello"
```

which, when called as:

```
print(greeting())
```

prints:

```
Hello
```

This can be written as a lambda function as follows:

```
greet_me = lambda: "Hello"
```

See note at the bottom of this section regarding the assignment of lambdas to variables. Generally, don't do it.

This creates an inline function with the name greet\_me that returns Hello. Note that you don't write **return** when creating a function with lambda. The value after : is automatically returned.

Once assigned to a variable, it can be used just like a regular function:

```
print(greet_me())
```

prints:

```
Hello
```

lambda can take arguments, too:

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("Hello ")
```

returns the string:

```
HELLO
```

它们也可以像普通函数一样接受任意数量的位置参数和关键字参数。

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

打印：

```
Hello ('world') {'world': 'world'}
```

**lambda**通常用于定义简短的函数，方便在调用点直接定义  
(通常与 `sorted`、`filter`和 `map`一起使用)。

例如，下面这行代码对字符串列表进行排序，忽略大小写以及字符串开头和结尾的空白字符：

```
sorted( [" foo ", " bAR", "BaZ"], key=lambda s: s.strip().upper())
# Out:
# [' bAR', 'BaZ', ' foo ']
```

仅忽略空格排序列表：

```
sorted( [" foo ", " bAR", "BaZ"], key=lambda s: s.strip())
# Out:
# ['BaZ', ' bAR', ' foo ']
```

使用`map`的示例：

```
sorted( map( lambda s: s.strip().upper(), [" foo ", " bAR", "BaZ"]))
# Out:
# ['BAR', 'BAZ', 'FOO']

sorted( map( lambda s: s.strip(), [" foo ", " bAR", "BaZ"]))
# Out:
# ['BaZ', 'bAR', 'foo ']
```

数值列表的示例：

```
my_list = [3, -4, -2, 5, 1, 7]
sorted( my_list, key=lambda x: abs(x))
# Out:
# [1, -2, 3, -4, 5, 7]

list( filter( lambda x: x>0, my_list))
# 输出:
# [3, 5, 1, 7]

list( map( lambda x: abs(x), my_list))
# 输出:
# [3, 4, 2, 5, 1, 7]
```

可以在**lambda**函数内部调用其他函数 (带参数或不带参数)。

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

They can also take arbitrary number of arguments / keyword arguments, like normal functions.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

prints:

```
Hello ('world',) {'world': 'world'}
```

**lambda**s are commonly used for short functions that are convenient to define at the point where they are called (typically with `sorted`, `filter` and `map`).

For example, this line sorts a list of strings ignoring their case and ignoring whitespace at the beginning and at the end:

```
sorted( [" foo ", " bAR", "BaZ"], key=lambda s: s.strip().upper())
# Out:
# [' bAR', 'BaZ', ' foo ']
```

Sort list just ignoring whitespaces:

```
sorted( [" foo ", " bAR", "BaZ"], key=lambda s: s.strip())
# Out:
# ['BaZ', ' bAR', ' foo ']
```

Examples with `map`:

```
sorted( map( lambda s: s.strip().upper(), [" foo ", " bAR", "BaZ"]))
# Out:
# ['BAR', 'BAZ', 'FOO']

sorted( map( lambda s: s.strip(), [" foo ", " bAR", "BaZ"]))
# Out:
# ['BaZ', 'bAR', 'foo ']
```

Examples with numerical lists:

```
my_list = [3, -4, -2, 5, 1, 7]
sorted( my_list, key=lambda x: abs(x))
# Out:
# [1, -2, 3, -4, 5, 7]

list( filter( lambda x: x>0, my_list))
# Out:
# [3, 5, 1, 7]

list( map( lambda x: abs(x), my_list))
# Out:
# [3, 4, 2, 5, 1, 7]
```

One can call other functions (with/without arguments) from inside a **lambda** function.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

```
hello world
```

这很有用，因为lambda只能包含一个表达式，通过使用辅助函数可以执行多个语句。

## 注意

请记住，PEP-8（[官方的Python风格指南](#)）不建议将lambda表达式赋值给变量（如我们在前两个例子中所做的）：

总是使用def语句，而不是直接将lambda表达式绑定到标识符的赋值语句。

是的：

```
def f(x): return 2*x
```

不建议：

```
f = lambda x: 2*x
```

第一种形式意味着生成的函数对象的名称明确为f，而不是通用的<lambda>。这对于追踪错误和字符串表示通常更有用。使用赋值语句消除了lambda表达式相较于显式def语句唯一的优势（即它可以嵌入到更大的表达式中）。

prints:

```
hello world
```

This is useful because **lambda** may contain only one expression and by using a subsidiary function one can run multiple statements.

## NOTE

Bear in mind that [PEP-8](#) (the official Python style guide) does not recommend assigning lambdas to variables (as we did in the first two examples):

Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.

Yes:

```
def f(x): return 2*x
```

No:

```
f = lambda x: 2*x
```

The first form means that the name of the resulting function object is specifically f instead of the generic <lambda>. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit **def** statement (i.e. that it can be embedded inside a larger expression).

## 第33.4节：定义带有可选参数的函数

可选参数可以通过给参数名赋予默认值（使用=）来定义：

```
def make(action='nothing'):
    return action
```

调用此函数有三种不同的方式：

```
make("fun")
# 输出: fun

make(action="sleep")
# 输出: sleep

# 参数是可选的，因此如果未传入参数，函数将使用默认值。

make()
# 输出: nothing
```

## 警告

可变类型（列表、字典、集合等）在作为默认属性时应谨慎对待。对默认参数的任何修改都会永久改变它。详见[定义带有可选可变参数的函数](#)。

## Section 33.4: Defining a function with optional arguments

Optional arguments can be defined by assigning (using =) a default value to the argument-name:

```
def make(action='nothing'):
    return action
```

Calling this function is possible in 3 different ways:

```
make("fun")
# Out: fun

make(action="sleep")
# Out: sleep

# The argument is optional so the function will use the default value if the argument is
# not passed in.
make()
# Out: nothing
```

## Warning

Mutable types ([list](#), [dict](#), [set](#), etc.) should be treated with care when given as **default** attribute. Any mutation of the default argument will change it permanently. See [Defining a function with optional mutable arguments](#).

## 第33.5节：定义带有可选可变参数的函数

使用可选参数与可变默认类型（在定义带有可选参数的函数中描述）时存在一个问题，这可能导致意外的行为。

### 说明

这个问题产生的原因是函数的默认参数只在函数被定义时初始化一次，而不是像许多其他语言那样在函数被调用时初始化。默认值存储在函数对象的`__defaults__`成员变量中。

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# 输出: (42, [])
```

对于不可变类型（参见参数传递和可变性），这不是问题，因为变量无法被修改；它只能被重新赋值，原始值保持不变。因此，后续调用保证具有相同的默认值。然而，对于可变类型，原始值可以通过调用其各种成员函数而发生变化。因此，函数的连续调用不能保证具有初始默认值。

```
def append(elem, to=[]):
    to.append(elem)      # 这次对append()的调用修改了默认变量"to"
    return to

append(1)
# 输出: [1]

append(2)  # 将其追加到内部存储的列表中
# 输出: [1, 2]

append(3, [])  # 使用新创建的列表会得到预期结果
# 输出: [3]

# 再次调用时如果不传参数，将再次追加到内部存储的列表中
append(4)
# 输出: [1, 2, 4]
```

注意：一些IDE如PyCharm在指定可变类型作为默认属性时会发出警告。

### 解决方案

如果你想确保默认参数始终是函数定义中指定的那个，解决方案是始终使用不可变类型作为默认参数。

当需要使用可变类型作为默认值时，一个常见的惯用法是使用`None`（不可变）作为默认参数，然后如果参数变量等于`None`，再将实际的默认值赋给该参数变量。

```
def append(elem, to=None):
    if to is None:
        to = []
```

## Section 33.5: Defining a function with optional mutable arguments

There is a problem when using **optional arguments** with a **mutable default type** (described in Defining a function with optional arguments), which can potentially lead to unexpected behaviour.

### Explanation

This problem arises because a function's default arguments are initialised **once**, at the point when the function is *defined*, and **not** (like many other languages) when the function is *called*. The default values are stored inside the function object's `__defaults__` member variable.

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# Out: (42, [])
```

For **immutable** types (see Argument passing and mutability) this is not a problem because there is no way to mutate the variable; it can only ever be reassigned, leaving the original value unchanged. Hence, subsequent are guaranteed to have the same default value. However, for a **mutable** type, the original value can mutate, by making calls to its various member functions. Therefore, successive calls to the function are not guaranteed to have the initial default value.

```
def append(elem, to=[]):
    to.append(elem)      # This call to append() mutates the default variable "to"
    return to

append(1)
# Out: [1]

append(2)  # Appends it to the internally stored list
# Out: [1, 2]

append(3, [])  # Using a new created list gives the expected result
# Out: [3]

# Calling it again without argument will append to the internally stored list again
append(4)
# Out: [1, 2, 4]
```

**Note:** Some IDEs like PyCharm will issue a warning when a mutable type is specified as a default attribute.

### Solution

If you want to ensure that the default argument is always the one you specify in the function definition, then the solution is to **always** use an immutable type as your default argument.

A common idiom to achieve this when a mutable type is needed as the default, is to use `None` (immutable) as the default argument and then assign the actual default value to the argument variable if it is equal to `None`.

```
def append(elem, to=None):
    if to is None:
        to = []
```

```
to.append(elem)
return to
```

## 第33.6节：参数传递与可变性

首先，一些术语：

- **参数（实际参数）**：传递给函数的实际变量；
- **形参（形式参数）**：函数中使用的接收变量。

在Python中，参数是通过赋值传递的（与其他语言不同，其他语言中参数可以通过值/引用/指针传递）。

- 修改参数会修改实参（如果实参的类型是可变的）。

```
def foo(x):      # 这里x是形参
    x[0] = 9     # 这会修改由x和y共同指向的列表
    print(x)

y = [4, 5, 6]
foo(y)          # 用y作为参数调用foo
# 输出: [9, 5, 6] # 由x标记的列表已被修改
print(y)
# 输出: [9, 5, 6] # 由y标记的列表也已被修改
```

- 重新赋值参数不会重新赋值实参。

```
def foo(x):      # 这里x是参数，当我们调用foo(y)时，将y赋给x
    x[0] = 9     # 这会修改由x和y标记的列表
x = [1, 2, 3]   # x现在标记一个不同的列表(y不受影响)
    x[2] = 8     # 这会修改x的列表，而不是y的列表

y = [4, 5, 6]    # y是实参，x是参数
foo(y)          # 假设我们写了"x = y"，然后执行第1行
y
# 输出: [9, 5, 6]
```

在 Python 中，我们实际上并不将值赋给变量，而是将变量（视为名称）绑定（即赋值、附加）到对象上。

- 不可变类型：整数、字符串、元组等。所有操作都会生成副本。
- 可变的：列表、字典、集合等。操作可能会或可能不会改变其内容。

```
x = [3, 1, 9]
y = x
x.append(5)    # 修改了由x和y指向的列表，x和y都绑定到[3, 1, 9]
x.sort()       # 修改了由x和y指向的列表（原地排序）
x = x + [4]    # 不修改列表（仅为x创建了一个副本，y不变）
z = x          # z指向x([1, 3, 9, 4])
x += [6]       # 修改了由x和z指向的列表（使用extend函数）。
x = sorted(x) # 不修改列表（仅为x创建了一个副本）。
x
# 输出: [1, 3, 4, 5, 6, 9]
y
# 输出: [1, 3, 5, 9]
z
```

```
to.append(elem)
return to
```

## Section 33.6: Argument passing and mutability

First, some terminology:

- **argument (actual parameter)**: the actual variable being passed to a function;
- **parameter (formal parameter)**: the receiving variable that is used in a function.

In Python, arguments are passed by **assignment** (as opposed to other languages, where arguments can be passed by value/reference(pointer)).

- Mutating a parameter will mutate the argument (if the argument's type is mutable).

```
def foo(x):      # here x is the parameter
    x[0] = 9     # This mutates the list labelled by both x and y
    print(x)

y = [4, 5, 6]
foo(y)          # call foo with y as argument
# Out: [9, 5, 6] # list labelled by x has been mutated
print(y)
# Out: [9, 5, 6] # list labelled by y has been mutated too
```

- Reassigning the parameter won't reassign the argument.

```
def foo(x):      # here x is the parameter, when we call foo(y) we assign y to x
    x[0] = 9     # This mutates the list labelled by both x and y
    x = [1, 2, 3] # x is now labeling a different list (y is unaffected)
    x[2] = 8     # This mutates x's list, not y's list

y = [4, 5, 6]    # y is the argument, x is the parameter
foo(y)          # Pretend that we wrote "x = y", then go to line 1
y
# Out: [9, 5, 6]
```

In Python, we don't really assign values to variables, instead we bind (i.e. assign, attach) variables (considered as names) to objects.

- **Immutable**: Integers, strings, tuples, and so on. All operations make copies.
- **Mutable**: Lists, dictionaries, sets, and so on. Operations may or may not mutate.

```
x = [3, 1, 9]
y = x
x.append(5)    # Mutates the list labelled by x and y, both x and y are bound to [3, 1, 9]
x.sort()       # Mutates the list labelled by x and y (in-place sorting)
x = x + [4]    # Does not mutate the list (makes a copy for x only, not y)
z = x          # z is x ([1, 3, 9, 4])
x += [6]       # Mutates the list labelled by both x and z (uses the extend function).
x = sorted(x) # Does not mutate the list (makes a copy for x only).
x
# Out: [1, 3, 4, 5, 6, 9]
y
# Out: [1, 3, 5, 9]
z
```

```
# 输出: [1, 3, 5, 9, 4, 6]
```

## 第33.7节：从函数返回值

函数可以返回一个你可以直接使用的值：

```
def give_me_five():
    return 5

print(give_me_five()) # 打印返回的值
# 输出: 5
```

或者将值保存以备后用：

```
num = give_me_five()
print(num)           # 打印保存的返回值
# 输出: 5
```

或者将该值用于任何运算：

```
print(give_me_five() + 10)
# 输出: 15
```

如果在函数中遇到return，函数将立即退出，后续操作将不会被执行：

```
def give_me_another_five():
    return 5
    print('这条语句永远不会被打印。')

print(give_me_another_five())
# 输出: 5
```

你也可以返回多个值（以元组的形式）：

```
def give_me_two_fives():
    return 5, 5 # 返回两个5

first, second = give_me_two_fives()
print(first)
# 输出: 5
print(second)
# 输出: 5
```

没有return语句的函数隐式返回None。同样，带有return语句但没有返回值或变量的函数也返回None。

```
# Out: [1, 3, 5, 9, 4, 6]
```

## Section 33.7: Returning values from functions

Functions can **return** a value that you can use directly:

```
def give_me_five():
    return 5

print(give_me_five()) # Print the returned value
# Out: 5
```

or save the value for later use:

```
num = give_me_five()
print(num)           # Print the saved returned value
# Out: 5
```

or use the value for any operations:

```
print(give_me_five() + 10)
# Out: 15
```

If **return** is encountered in the function the function will be exited immediately and subsequent operations will not be evaluated:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

You can also **return** multiple values (in the form of a tuple):

```
def give_me_two_fives():
    return 5, 5 # Returns two 5

first, second = give_me_two_fives()
print(first)
# Out: 5
print(second)
# Out: 5
```

A function with no **return** statement implicitly returns **None**. Similarly a function with a **return** statement, but no return value or variable returns **None**.

## 第33.8节：闭包

Python中的闭包是通过函数调用创建的。这里，对makeInc的调用为x创建了一个绑定，该绑定在函数inc内部被引用。每次调用makeInc都会创建该函数的新实例，但每个实例都链接到x的不同绑定。

```
def makeInc(x):
    def inc(y):
        # x 在 inc 的定义中是“附加”的
```

## Section 33.8: Closure

Closures in Python are created by function calls. Here, the call to `makeInc` creates a binding for `x` that is referenced inside the function `inc`. Each call to `makeInc` creates a new instance of this function, but each instance has a link to a different binding of `x`.

```
def makeInc(x):
    def inc(y):
        # x is "attached" in the definition of inc
```

```
return y + x
```

```
return inc
```

```
incOne = makeInc(1)  
incFive = makeInc(5)
```

```
incOne(5) # 返回 6  
incFive(5) # 返回 10
```

注意，虽然在普通闭包中，内部函数会完全继承其外部环境中的所有变量，但在此结构中，内部函数只能对继承的变量进行只读访问，不能对其进行赋值

```
def makeInc(x):  
    def inc(y):  
        # 不允许对 x 进行递增操作  
        x += y  
        return x  
  
    return inc
```

```
incOne = makeInc(1)  
incOne(5) # UnboundLocalError: 在赋值前引用了局部变量 'x'
```

Python 3 提供了 `nonlocal` 语句（非局部变量），用于实现带嵌套函数的完整闭包。

Python 3.x 版本  $\geq 3.0$

```
def makeInc(x):  
    def inc(y):  
        nonlocal x  
        # 现在允许给 x 赋值  
        x += y  
        return x  
  
    return inc
```

```
incOne = makeInc(1)  
incOne(5) # 返回 6
```

## 第 33.9 节：强制使用命名参数

函数签名中第一个星号之后指定的所有参数都是仅限关键字参数。

```
def f(*a, b):  
    pass  
  
f(1, 2, 3)  
# TypeError: f() 缺少 1 个必需的仅限关键字参数：'b'
```

在 Python 3 中，可以在函数签名中放置一个单独的星号，以确保剩余的参数只能通过关键字参数传递。

```
def f(a, b, *, c):  
    pass  
  
f(1, 2, 3)  
# TypeError: f() 接受 2 个位置参数，但给了 3 个
```

```
return y + x
```

```
return inc
```

```
incOne = makeInc(1)  
incFive = makeInc(5)
```

```
incOne(5) # returns 6  
incFive(5) # returns 10
```

Notice that while in a regular closure the enclosed function fully inherits all variables from its enclosing environment, in this construct the enclosed function has only read access to the inherited variables but cannot make assignments to them

```
def makeInc(x):  
    def inc(y):  
        # incrementing x is not allowed  
        x += y  
        return x  
  
    return inc
```

```
incOne = makeInc(1)  
incOne(5) # UnboundLocalError: local variable 'x' referenced before assignment
```

Python 3 offers the `nonlocal` statement (Nonlocal Variables) for realizing a full closure with nested functions.

Python 3.x Version  $\geq 3.0$

```
def makeInc(x):  
    def inc(y):  
        nonlocal x  
        # now assigning a value to x is allowed  
        x += y  
        return x  
  
    return inc  
  
incOne = makeInc(1)  
incOne(5) # returns 6
```

## Section 33.9: Forcing the use of named parameters

All parameters specified after the first asterisk in the function signature are keyword-only.

```
def f(*a, b):  
    pass  
  
f(1, 2, 3)  
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

In Python 3 it's possible to put a single asterisk in the function signature to ensure that the remaining arguments may only be passed using keyword arguments.

```
def f(a, b, *, c):  
    pass  
  
f(1, 2, 3)  
# TypeError: f() takes 2 positional arguments but 3 were given
```

```
f(1, 2, c=3)
# 无错误
```

## 第 33.10 节：嵌套函数

Python 中的函数是第一类对象。它们可以在任何作用域中定义

```
def fibonacci(n):
    def step(a,b):
        return b, a+b
    a, b = 0, 1
    for i in range(n):
        a, b = step(a, b)
    return a
```

函数捕获其封闭作用域，可以像其他任何类型的对象一样被传递

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#输出: 15
add6(10)
#输出: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Out: 26
```

## 第33.11节：递归限制

递归的最大深度是有限制的，这取决于Python的实现。当达到限制时，会引发RuntimeError异常：

```
def cursing(depth):
    try:
cursing(depth + 1) # 实际上，是再次递归
    except RuntimeError as RE:
        print('我递归了 {} 次！'.format(depth))

cursing(0)
# Out: 我递归了1083次！
```

可以使用 `sys.setrecursionlimit(limit)` 来更改递归深度限制，并通过 `sys.getrecursionlimit` 来检查该限制。

```
sys.setrecursionlimit(2000)
cursing(0)
# 输出：我递归了1997次！
```

```
f(1, 2, c=3)
# No error
```

## Section 33.10: Nested functions

Functions in python are first-class objects. They can be defined in any scope

```
def fibonacci(n):
    def step(a,b):
        return b, a+b
    a, b = 0, 1
    for i in range(n):
        a, b = step(a, b)
    return a
```

Functions capture their enclosing scope can be passed around like any other sort of object

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#Out: 15
add6(10)
#Out: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Out: 26
```

## Section 33.11: Recursion limit

There is a limit to the depth of possible recursion, which depends on the Python implementation. When the limit is reached, a RuntimeError exception is raised:

```
def cursing(depth):
    try:
cursing(depth + 1) # actually, re-cursing
    except RuntimeError as RE:
        print('I recursed {} times！'.format(depth))

cursing(0)
# Out: I recursed 1083 times！
```

It is possible to change the recursion depth limit by using `sys.setrecursionlimit(limit)` and check this limit by `sys.getrecursionlimit()`.

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times！
```

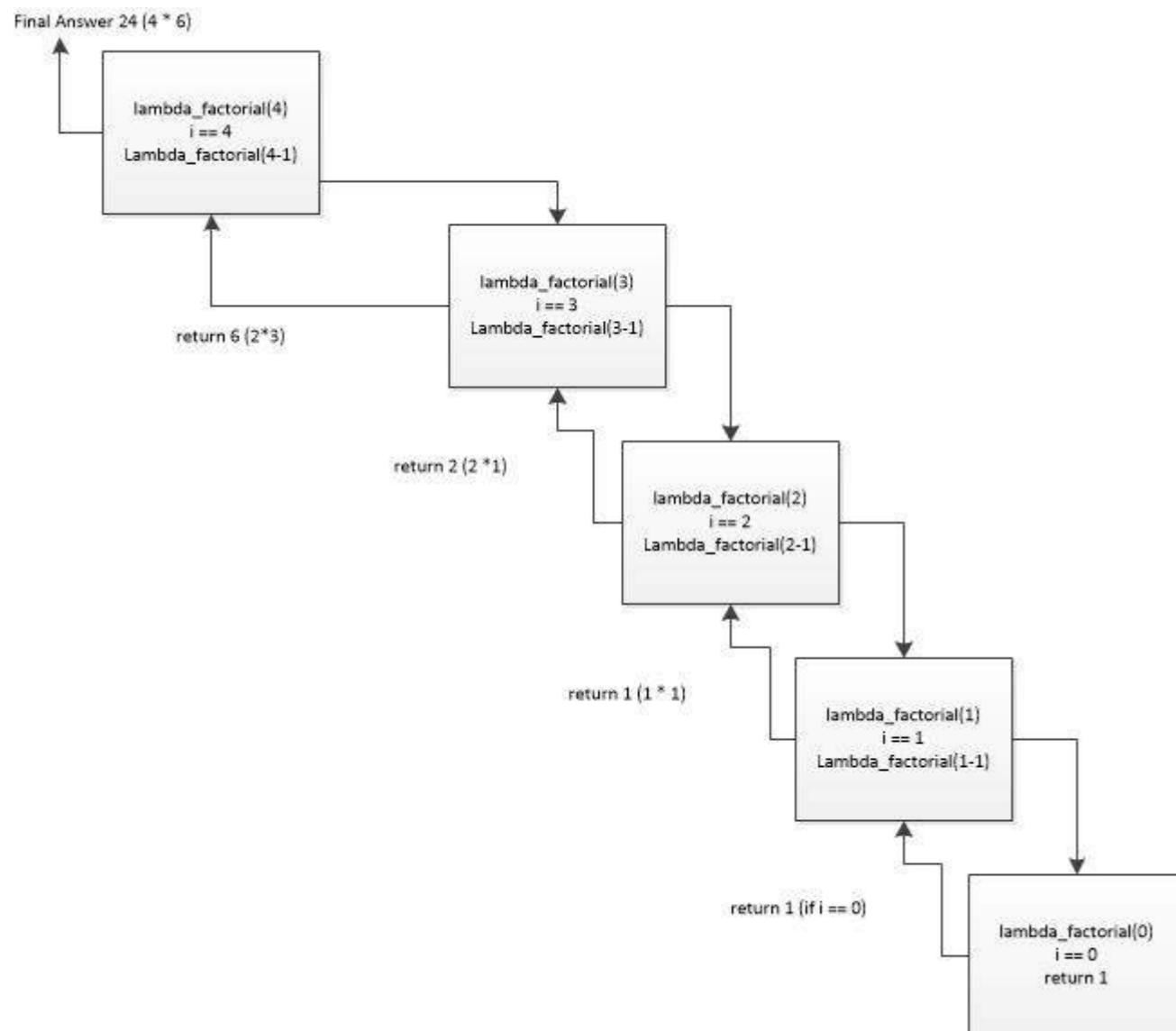
## 第33.12节：使用赋值变量的递归Lambda

创建递归Lambda函数的一种方法是将函数赋值给一个变量，然后在函数内部引用该变量。一个常见的例子是递归计算数字的阶乘——如以下代码所示：

```
lambda_factorial = lambda i:1 if i==0 else i*lambda_factorial(i-1)
print(lambda_factorial(4)) # 4 * 3 * 2 * 1 = 12 * 2 = 24
```

### 代码说明

通过变量赋值，lambda函数接收一个值（4），如果该值为0则返回1，否则返回当前值  $(i) * \text{lambda\_function}(i-1)$  的递归计算。该过程持续直到传入的值递减到0（返回 1）。该过程可以形象化为：



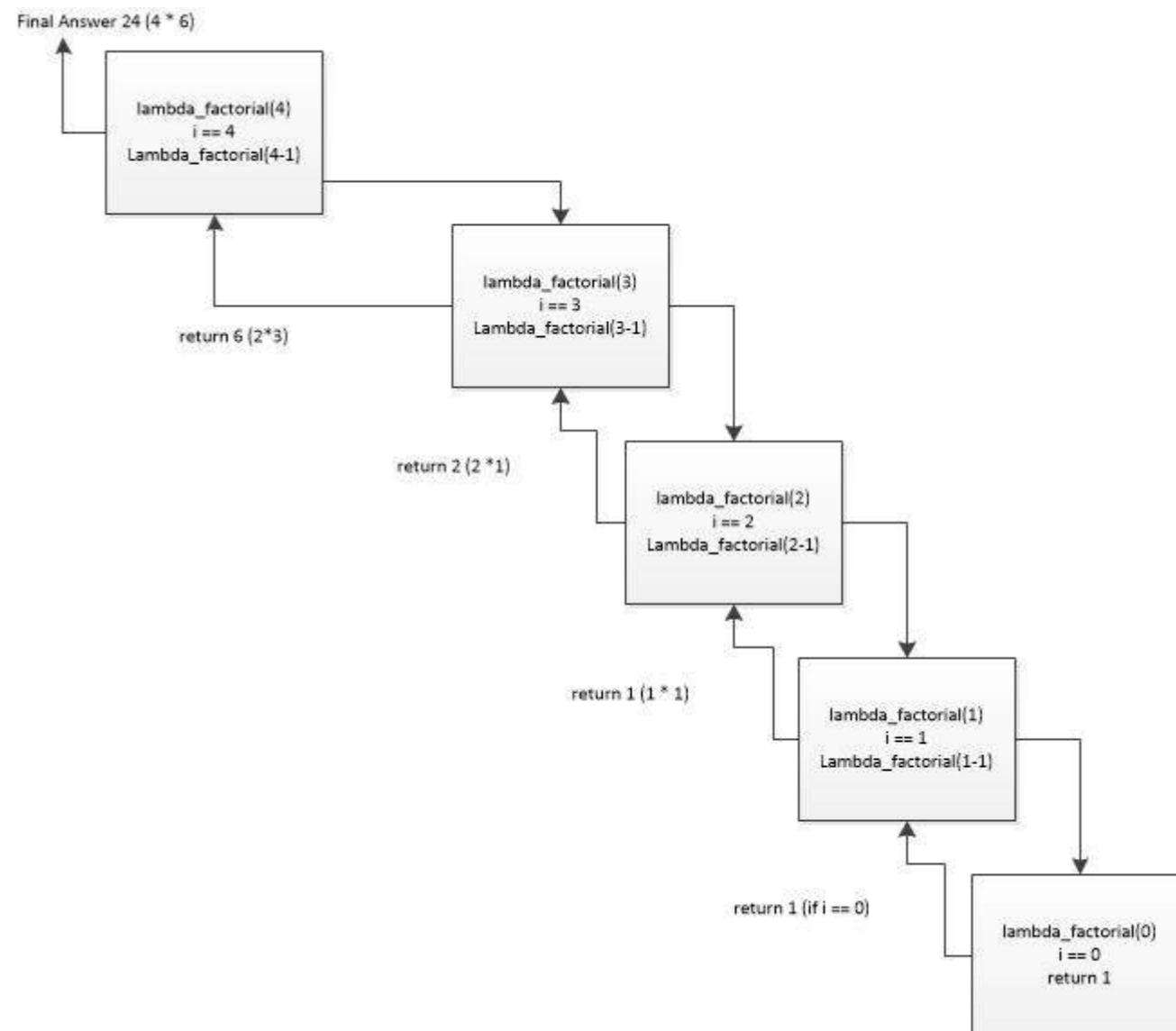
## Section 33.12: Recursive Lambda using assigned variable

One method for creating recursive lambda functions involves assigning the function to a variable and then referencing that variable within the function itself. A common example of this is the recursive calculation of the factorial of a number - such as shown in the following code:

```
lambda_factorial = lambda i:1 if i==0 else i*lambda_factorial(i-1)
print(lambda_factorial(4)) # 4 * 3 * 2 * 1 = 12 * 2 = 24
```

### Description of code

The lambda function, through its variable assignment, is passed a value (4) which it evaluates and returns 1 if it is 0 or else it returns the current value  $(i) * \text{another calculation by the lambda function of the value - 1}$  ( $i-1$ ). This continues until the passed value is decremented to 0 ([return 1](#)). A process which can be visualized as:



## 第33.13节：递归函数

递归函数是在其定义中调用自身的函数。例如数学函数阶乘，定义为 $\text{factorial}(n) = n*(n-1)*(n-2)*...*3*2*1$ 。可以编写为

## Section 33.13: Recursive functions

A recursive function is a function that calls itself in its definition. For example the mathematical function, factorial, defined by  $\text{factorial}(n) = n*(n-1)*(n-2)*...*3*2*1$ . can be programmed as

```
def factorial(n):
    #n 这里应该是一个整数
    如果 n == 0:
        返回 1
    否则:
        返回 n*factorial(n-1)
```

这里的输出是：

```
factorial(0)
#输出 1
factorial(1)
#输出 1
factorial(2)
#输出 2
factorial(3)
#out 6
```

如预期。注意这个函数是递归的，因为第二个return factorial(n-1)中，函数在定义中调用了自身。

一些递归函数可以用lambda实现，使用lambda的阶乘函数大致如下：

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

该函数的输出与上述相同。

## 第33.14节：定义带参数的函数

参数在函数名后面的括号中定义：

```
def divide(dividend, divisor): # 函数名及其参数名
    # 参数在函数体内可以通过名称访问
    print(dividend / divisor)
```

函数名及其参数列表称为函数的签名。每个命名参数实际上是函数的局部变量。

调用函数时，通过按顺序列出参数来传递值

```
divide(10, 2)
# 输出: 5
```

或者使用函数定义中的参数名以任意顺序指定它们：

```
divide(divisor=2, dividend=10)
# 输出: 5
```

## 第33.15节：可迭代对象和字典拆包

函数允许你指定以下类型的参数：位置参数、命名参数、可变位置参数、关键字参数（kwargs）。下面是每种类型的清晰简明示例。

```
def unpacking(a, b, c=45, d=60, *args, **kwargs):
```

```
def factorial(n):
    #n here should be an integer
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

the outputs here are:

```
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6
```

as expected. Notice that this function is recursive because the second `return factorial(n-1)`, where the function calls itself in its definition.

Some recursive functions can be implemented using lambda, the factorial function using lambda would be something like this:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

The function outputs the same as above.

## Section 33.14: Defining a function with arguments

Arguments are defined in parentheses after the function name:

```
def divide(dividend, divisor): # The names of the function and its arguments
    # The arguments are available by name in the body of the function
    print(dividend / divisor)
```

The function name and its list of arguments are called the *signature* of the function. Each named argument is effectively a local variable of the function.

When calling the function, give values for the arguments by listing them in order

```
divide(10, 2)
# output: 5
```

or specify them in any order using the names from the function definition:

```
divide(divisor=2, dividend=10)
# output: 5
```

## Section 33.15: Iterable and dictionary unpacking

Functions allow you to specify these types of parameters: positional, named, variable positional, Keyword args (kwargs). Here is a clear and concise use of each type.

```
def unpacking(a, b, c=45, d=60, *args, **kwargs):
```

```

print(a, b, c, d, args, kwargs)

>>> 解包(1, 2)
1 2 45 60 () {}

>>> unpacking(1, 2, 3, 4)
1 2 3 4 () {}

>>> unpacking(1, 2, c=3, d=4)
1 2 3 4 () {}

>>> unpacking(1, 2, d=4, c=3)
1 2 3 4 () {}

>>> pair = (3,)
>>> unpacking(1, 2, *pair, d=4)
1 2 3 4 () {}

>>> unpacking(1, 2, d=4, *pair)
1 2 3 4 () {}

>>> unpacking(1, 2, *pair, c=3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, c=3, *pair)
回溯 (最近一次调用最后) :
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

>>> args_list = [3]
>>> unpacking(1, 2, *args_list, d=4)
1 2 3 4 () {}

>>> unpacking(1, 2, d=4, *args_list)
1 2 3 4 () {}

>>> unpacking(1, 2, c=3, *args_list)
追踪 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <module>
类型错误: unpacking() 收到多个值 对应 参数 'c'
>>> unpacking(1, 2, *args_list, c=3)
回溯 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <module>
类型错误: unpacking() 收到多个值 对应 参数 'c'

>>> pair = (3, 4)
>>> unpacking(1, 2, *pair)
1 2 3 4 () {}

>>> unpacking(1, 2, 3, 4, *pair)
1 2 3 4 (3, 4) {}

>>> unpacking(1, 2, d=4, *pair)
追踪 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <module>
类型错误: unpacking() 收到多个值 对应 参数 'd'
>>> unpacking(1, 2, *pair, d=4)
回溯 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <module>
类型错误: unpacking() 收到多个值 对应 参数 'd'

>>> 参数列表 = [3, 4]
>>> unpacking(1, 2, *参数列表)
1 2 3 4 () {}

>>> unpacking(1, 2, 3, 4, *args_list)
1 2 3 4 (3, 4) {}

```

```

print(a, b, c, d, args, kwargs)

>>> unpacking(1, 2)
1 2 45 60 () {}

>>> unpacking(1, 2, 3, 4)
1 2 3 4 () {}

>>> unpacking(1, 2, c=3, d=4)
1 2 3 4 () {}

>>> unpacking(1, 2, d=4, c=3)
1 2 3 4 () {}

>>> pair = (3,)
>>> unpacking(1, 2, *pair, d=4)
1 2 3 4 () {}

>>> unpacking(1, 2, d=4, *pair)
1 2 3 4 () {}

>>> unpacking(1, 2, *pair, c=3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, c=3, *pair)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

>>> args_list = [3]
>>> unpacking(1, 2, *args_list, d=4)
1 2 3 4 () {}

>>> unpacking(1, 2, d=4, *args_list)
1 2 3 4 () {}

>>> unpacking(1, 2, c=3, *args_list)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, *args_list, c=3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

>>> pair = (3, 4)
>>> unpacking(1, 2, *pair)
1 2 3 4 () {}

>>> unpacking(1, 2, 3, 4, *pair)
1 2 3 4 (3, 4) {}

>>> unpacking(1, 2, d=4, *pair)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *pair, d=4)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

>>> args_list = [3, 4]
>>> unpacking(1, 2, *args_list)
1 2 3 4 () {}

>>> unpacking(1, 2, 3, 4, *args_list)
1 2 3 4 (3, 4) {}

```

```

>>> unpacking(1, 2, d=4, *args_list)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *args_list, d=4)
回溯 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <module>
类型错误: unpacking() 收到多个值 对应 参数 'd'

```

```

>>> arg_dict = {'c':3, 'd':4}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'d':4, 'c':3}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'c':3, 'd':4, 'not_a_parameter': 75}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {'not_a_parameter': 75}

```

```

>>> unpacking(1, 2, *pair, **arg_dict)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, 3, 4, **arg_dict)
回溯 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <module>
类型错误: unpacking() 收到多个值 对应 参数 'd'

```

```

# 位置参数优先于任何其他形式的参数传递
>>> 解包(1, 2, **arg_dict, c=3)
1 2 3 4 () {'not_a_parameter': 75}
>>> 解包(1, 2, 3, **arg_dict, c=3)
追踪 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <module>
类型错误: unpacking() 收到多个值 对应 参数 'c'

```

## 第33.16节：定义带有多个参数的函数

可以给函数传入任意数量的参数，唯一固定的规则是每个参数名必须唯一，且可选参数必须放在非可选参数之后：

```

def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)

```

调用函数时，可以不带参数名依次传入每个关键字参数，但顺序很重要：

```

print(func(1, 'a', 100))
# 输出: 1 a 100

print(func('abc', 14))
# abc 14 10

```

或者结合使用带名称和不带名称的参数。然后带名称的参数必须跟在不带名称的参数之后，但带名称参数的顺序无关紧要：

```

print(func('这是', optionalvalue='StackOverflow 文档', value2='是'))
# 输出: 这是 StackOverflow 文档

```

```

>>> unpacking(1, 2, d=4, *args_list)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *args_list, d=4)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

```

```

>>> arg_dict = {'c':3, 'd':4}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'d':4, 'c':3}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'c':3, 'd':4, 'not_a_parameter': 75}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {'not_a_parameter': 75}

```

```

>>> unpacking(1, 2, *pair, **arg_dict)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, 3, 4, **arg_dict)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

```

```

# Positional arguments take priority over any other form of argument passing
>>> unpacking(1, 2, **arg_dict, c=3)
1 2 3 4 () {'not_a_parameter': 75}
>>> unpacking(1, 2, 3, **arg_dict, c=3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

```

## Section 33.16: Defining a function with multiple arguments

One can give a function as many arguments as one wants, the only fixed rules are that each argument name must be unique and that optional arguments must be after the not-optional ones:

```

def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)

```

When calling the function you can either give each keyword without the name but then the order matters:

```

print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10

```

Or combine giving the arguments with name and without. Then the ones with name must follow those without but the order of the ones with name doesn't matter:

```

print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation

```

# 第34章：使用列表参数定义函数

## 第34.1节：函数与调用

列表作为参数只是另一种变量：

```
def func(myList):
    for item in myList:
        print(item)
```

并且可以直接在函数调用中传入：

```
func([1,2,3,5,7])
```

```
1
2
3
5
7
或者作为变量：
a
L
i
s
t
=
[
    'a',
    'b',
    'c',
    'd'
]
func(aList)
```

# Chapter 34: Defining functions with list arguments

## Section 34.1: Function and Call

Lists as arguments are just another variable:

```
def func(myList):
    for item in myList:
        print(item)
```

and can be passed in the function call itself:

```
func([1, 2, 3, 5, 7])
1
2
3
5
7
```

Or as a variable:

```
aList = ['a', 'b', 'c', 'd']
func(aList)
a
b
c
d
```

# 视频：机器学习A-Z：动手Python数据科学

向两位数据科学专家学习如何用Python创建机器学习算法。包含代码模板。



- ✓ 精通Python上的机器学习
- ✓ 对多种机器学习模型有良好的直觉
- ✓ 做出准确的预测
- ✓ 进行强有力的分析
- ✓ 构建稳健的机器学习模型
- ✓ 为您的业务创造强大的附加价值
- ✓ 将机器学习用于个人目的
- ✓ 处理强化学习、自然语言处理和深度学习等特定主题
- ✓ 掌握降维等高级技术
- ✓ 了解针对每种问题应选择哪种机器学习模型
- ✓ 构建一支强大的机器学习模型军团，并知道如何组合它们来解决任何问题

今天观看 →

# VIDEO: Machine Learning A-Z: Hands-On Python In Data Science

Learn to create Machine Learning Algorithms in Python from two Data Science experts. Code templates included.



- ✓ Master Machine Learning on Python
- ✓ Have a great intuition of many Machine Learning models
- ✓ Make accurate predictions
- ✓ Make powerful analysis
- ✓ Make robust Machine Learning models
- ✓ Create strong added value to your business
- ✓ Use Machine Learning for personal purpose
- ✓ Handle specific topics like Reinforcement Learning, NLP and Deep Learning
- ✓ Handle advanced techniques like Dimensionality Reduction
- ✓ Know which Machine Learning model to choose for each type of problem
- ✓ Build an army of powerful Machine Learning models and know how to combine them to solve any problem

Watch Today →

# 第35章：Python中的函数式编程

函数式编程将问题分解为一组函数。理想情况下，函数只接受输入并产生输出，且不具有任何影响给定输入输出的内部状态。以下是许多语言中常见的函数式技术：如lambda、map、reduce。

## 第35.1节：Lambda函数

使用lambda定义的匿名内联函数。lambda的参数定义在冒号左侧。函数体定义在冒号右侧。运行函数体的结果（隐式地）被返回。

```
s=lambda x:x**x  
s(2)    =>4
```

## 第35.2节：Map函数

Map接受一个函数和一个项目集合。它创建一个新的空集合，对原集合中的每个项目运行该函数，并将每个返回值插入新集合。它返回新集合。

这是一个简单的map，接受一个名字列表并返回这些名字长度的列表：

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])  
print(name_lengths)  =>[4, 4, 3]
```

## 第35.3节：Reduce函数

Reduce接受一个函数和一组元素。它返回一个通过组合这些元素生成的值。

这是一个简单的reduce。它返回集合中所有元素的和。

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])  
print(total)    =>10
```

## 第35.4节：Filter函数

Filter接受一个函数和一组元素。它返回一个集合，包含所有使函数返回True的元素。

```
arr=[1,2,3,4,5,6]  
[i for i in filter(lambda x:x>4,arr)]  # 输出[5,6]
```

# Chapter 35: Functional Programming in Python

Functional programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input. Below are functional techniques common to many languages: such as lambda, map, reduce.

## Section 35.1: Lambda Function

An anonymous, inlined function defined with lambda. The parameters of the lambda are defined to the left of the colon. The function body is defined to the right of the colon. The result of running the function body is (implicitly) returned.

```
s=lambda x:x**x  
s(2)    =>4
```

## Section 35.2: Map Function

Map takes a function and a collection of items. It makes a new, empty collection, runs the function on each item in the original collection and inserts each return value into the new collection. It returns the new collection.

This is a simple map that takes a list of names and returns a list of the lengths of those names:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])  
print(name_lengths)  =>[4, 4, 3]
```

## Section 35.3: Reduce Function

Reduce takes a function and a collection of items. It returns a value that is created by combining the items.

This is a simple reduce. It returns the sum of all the items in the collection.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])  
print(total)    =>10
```

## Section 35.4: Filter Function

Filter takes a function and a collection. It returns a collection of every item for which the function returned True.

```
arr=[1,2,3,4,5,6]  
[i for i in filter(lambda x:x>4,arr)]  # outputs[5,6]
```

# 第36章：偏函数

参数	细节
x	要幂运算的数
y	指数
raise	要被特化的函数

如果你来自面向对象编程（OOP）背景，你可能知道，特化一个抽象类并使用它是你在编写代码时应牢记的做法。

如果你可以定义一个抽象函数并对其进行特化，以创建不同版本的函数呢？把它看作是一种函数继承，你绑定特定参数，使其在特定场景下更可靠。

## 第36.1节：幂运算

假设我们想将 `x` 提升到一个数字 `y` 的幂。

你会这样写：

```
def raise_power(x, y):
    返回 x**y
```

如果你的 `y` 值可以取有限个值怎么办？

假设 `y` 可以是 `[3,4,5]` 中的一个，并且你不想让最终用户使用这样的函数因为它计算量非常大。实际上你会检查提供的 `y` 是否是有效值，并重写你的函数如下：

```
定义 raise(x, y):
    如果 y 在 (3,4,5) 中:
        返回 x**y
    raise NumberNotInRangeException("你应该提供一个有效的指数")
```

很乱？让我们使用抽象形式并针对这三种情况进行特化：我们部分地实现它们。

```
从 functors 导入 partial
raise_to_three = partial(raise, y=3)
raise_to_four = partial(raise, y=4)
raise_to_five = partial(raise, y=5)
```

这里发生了什么？我们固定了 `y` 参数，并定义了三个不同的函数。

不需要使用上面定义的抽象函数（你可以将其设为私有），但你可以使用**partial** 应用的函数来处理将数字提升到固定值的操作。

# Chapter 36: Partial functions

Param	details
x	the number to be raised
y	the exponent
raise	the function to be specialized

As you probably know if you came from OOP school, specializing an abstract class and use it is a practice you should keep in mind when writing your code.

What if you could define an abstract function and specialize it in order to create different versions of it? Thinks it as a sort of *function Inheritance* where you bind specific params to make them reliable for a specific scenario.

## Section 36.1: Raise the power

Let's suppose we want raise `x` to a number `y`.

You'd write this as:

```
def raise_power(x, y):
    return x**y
```

What if your `y` value can assume a finite set of values?

Let's suppose `y` can be one of `[3,4,5]` and let's say you don't want offer end user the possibility to use such function since it is very computationally intensive. In fact you would check if provided `y` assumes a valid value and rewrite your function as:

```
def raise(x, y):
    if y in (3, 4, 5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

Messy? Let's use the abstract form and specialize it to all three cases: let's implement them **partially**.

```
from functors import partial
raise_to_three = partial(raise, y=3)
raise_to_four = partial(raise, y=4)
raise_to_five = partial(raise, y=5)
```

What happens here? We fixed the `y` params and we defined three different functions.

No need to use the abstract function defined above (you could make it *private*) but you could use **partial applied** functions to deal with raising a number to a fixed value.

# 第37章：装饰器

参数	详情
f	被装饰（包装）的函数

装饰器函数是一种软件设计模式。它们动态地改变函数、方法或类的功能，而无需直接使用子类或更改被装饰函数的源代码。正确使用时，装饰器可以成为开发过程中的强大工具。本章节涵盖了Python中装饰器函数的实现和应用。

## 第37.1节：装饰器函数

装饰器增强其他函数或方法的行为。任何接受函数作为参数并返回增强函数的函数都可以用作装饰器。

```
# 这个最简单的装饰器对被装饰的函数没有任何改变。  
# 这种最小的装饰器有时可以用作一种代码标记。  
def super_secret_function(f):  
    return f  
  
@super_secret_function  
def my_function():  
    print("这是我的秘密函数。")
```

“@”符号是语法糖，其等价于以下写法：

```
my_function = super_secret_function(my_function)
```

理解装饰器的工作原理时，牢记这一点非常重要。这种“去糖”后的语法清楚地说明了为什么装饰器函数以一个函数作为参数，以及为什么它应该返回另一个函数。它还展示了如果你不返回函数会发生什么：

```
def disabled(f):  
    ...  
    pass  
  
@disabled  
def my_function():  
    print("这个函数无法再被调用了...")  
  
my_function()  
# TypeError: 'NoneType' 对象不可调用
```

因此，我们通常在装饰器内部定义一个新函数并返回它。这个新函数会先执行它需要做的事情，然后调用原始函数，最后处理返回值。考虑这个简单的装饰器函数，它打印原始函数接收到的参数，然后调用该函数。

```
#这是装饰器  
def print_args(func):  
    def inner_func(*args, **kwargs):  
        print(args)  
        print(kwargs)
```

# Chapter 37: Decorators

Parameter	Details
f	The function to be decorated (wrapped)

Decorator functions are software design patterns. They dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the decorated function. When used correctly, decorators can become powerful tools in the development process. This topic covers implementation and applications of decorator functions in Python.

## Section 37.1: Decorator function

Decorators augment the behavior of other functions or methods. Any function that takes a function as a parameter and returns an augmented function can be used as a **decorator**.

```
# This simplest decorator does nothing to the function being decorated. Such  
# minimal decorators can occasionally be used as a kind of code markers.  
def super_secret_function(f):  
    return f  
  
@super_secret_function  
def my_function():  
    print("This is my secret function.")
```

The @-notation is syntactic sugar that is equivalent to the following:

```
my_function = super_secret_function(my_function)
```

It is important to bear this in mind in order to understand how the decorators work. This "unsugared" syntax makes it clear why the decorator function takes a function as an argument, and why it should return another function. It also demonstrates what would happen if you *don't* return a function:

```
def disabled(f):  
    ...  
    This function returns nothing, and hence removes the decorated function  
    from the local scope.  
    ...  
    pass  
  
@disabled  
def my_function():  
    print("This function can no longer be called...")  
  
my_function()  
# TypeError: 'NoneType' object is not callable
```

Thus, we usually define a *new function* inside the decorator and return it. This new function would first do something that it needs to do, then call the original function, and finally process the return value. Consider this simple decorator function that prints the arguments that the original function receives, then calls it.

```
#This is the decorator  
def print_args(func):  
    def inner_func(*args, **kwargs):  
        print(args)  
        print(kwargs)
```

```

    return func(*args, **kwargs) #使用参数调用原始函数。
return inner_func

@print_args
def multiply(num_a, num_b):
    return num_a * num_b

print(multiply(3, 5))
#输出：
# (3,5) - 这实际上是函数接收的“args”。
# {} - 这是 'kwargs'，为空因为我们没有指定关键字参数。
# 15 - 函数的结果。

```

## 第37.2节：装饰器类

如引言中所述，装饰器是一个可以应用于另一个函数以增强其行为的函数。语法糖等同于以下写法：`my_func=decorator(my_func)`。但如果装饰器是一个类呢？语法仍然有效，只是现在`my_func`被替换成了`decorator`类的一个实例。

如果该类实现了`__call__()`魔法方法，那么仍然可以像使用函数一样使用`my_func`：

```

class Decorator(object):
    """简单的装饰器类。"""

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('函数调用之前。')
        res = self.func(*args, **kwargs)
        print('函数调用后。')
        return res

```

```

@装饰器
def 测试函数():
    print('函数内部。')

```

```

测试函数()
# 函数调用之前。
# 函数内部。
# 函数调用之后。

```

请注意，使用类装饰器装饰的函数从类型检查的角度来看将不再被视为“函数”：

```

import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>

```

### 装饰方法

要装饰方法，您需要定义一个额外的`__get__`方法：

```

from types import MethodType

class 装饰器(object):
    def __init__(self, func):

```

```

        return func(*args, **kwargs) #Call the original function with its arguments.
        return inner_func

@print_args
def multiply(num_a, num_b):
    return num_a * num_b

print(multiply(3, 5))
#Output:
# (3,5) - This is actually the 'args' that the function receives.
# {} - This is the 'kwargs', empty because we didn't specify keyword arguments.
# 15 - The result of the function.

```

## Section 37.2: Decorator class

As mentioned in the introduction, a decorator is a function that can be applied to another function to augment its behavior. The syntactic sugar is equivalent to the following: `my_func = decorator(my_func)`. But what if the decorator was instead a class? The syntax would still work, except that now `my_func` gets replaced with an instance of the decorator class. If this class implements the `__call__()` magic method, then it would still be possible to use `my_func` as if it was a function:

```

class Decorator(object):
    """Simple decorator class."""

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Before the function call.')
        res = self.func(*args, **kwargs)
        print('After the function call.')
        return res

@Decorator
def testfunc():
    print('Inside the function.')

testfunc()
# Before the function call.
# Inside the function.
# After the function call.

```

Note that a function decorated with a class decorator will no longer be considered a "function" from type-checking perspective:

```

import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>

```

### Decorating Methods

For decorating methods you need to define an additional `__get__`-method:

```

from types import MethodType

class Decorator(object):
    def __init__(self, func):

```

```

self.func = func

def __call__(self, *args, **kwargs):
    print('在装饰器内部。')
    return self.func(*args, **kwargs)

def __get__(self, instance, cls):
    # 如果在实例上调用，则返回一个方法
    return self if instance is None else MethodType(self, instance)

class 测试(object):
    @装饰器
    def __init__(self):
        pass

a = Test()

```

装饰器内部。

### 警告！

类装饰器对于特定函数只会产生一个实例，因此用类装饰器装饰方法会在该类的所有实例之间共享同一个装饰器：

```

from types import MethodType

class CountCallsDecorator(object):
    def __init__(self, func):
        self.func = func
        self.ncalls = 0    # 该方法的调用次数

    def __call__(self, *args, **kwargs):
        self.ncalls += 1  # 调用计数器加一
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        return self if instance is None else MethodType(self, instance)

class 测试(object):
    def __init__(self):
        pass

    @CountCallsDecorator
    def do_something(self):
        return '执行了一些操作'

a = Test()
a.do_something()
a.do_something.ncalls  # 1
b = Test()
b.do_something()
b.do_something.ncalls  # 2

```

## 第37.3节：带参数的装饰器（装饰器工厂）

装饰器只接受一个参数：被装饰的函数。无法传递其他参数。

但通常需要额外的参数。诀窍是编写一个接受任意参数的函数

```

self.func = func

def __call__(self, *args, **kwargs):
    print('Inside the decorator.')
    return self.func(*args, **kwargs)

def __get__(self, instance, cls):
    # Return a Method if it is called on an instance
    return self if instance is None else MethodType(self, instance)

class Test(object):
    @Decorator
    def __init__(self):
        pass

a = Test()

```

Inside the decorator.

### Warning!

Class Decorators only produce one instance for a specific function so decorating a method with a class decorator will share the same decorator between all instances of that class:

```

from types import MethodType

class CountCallsDecorator(object):
    def __init__(self, func):
        self.func = func
        self.ncalls = 0    # Number of calls of this method

    def __call__(self, *args, **kwargs):
        self.ncalls += 1  # Increment the calls counter
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        return self if instance is None else MethodType(self, instance)

class Test(object):
    def __init__(self):
        pass

    @CountCallsDecorator
    def do_something(self):
        return 'something was done'

a = Test()
a.do_something()
a.do_something.ncalls  # 1
b = Test()
b.do_something()
b.do_something.ncalls  # 2

```

## Section 37.3: Decorator with arguments (decorator factory)

A decorator takes just one argument: the function to be decorated. There is no way to pass other arguments.

But additional arguments are often desired. The trick is then to make a function which takes arbitrary arguments

并返回一个装饰器。

### 装饰器函数

```
def decoratorfactory(message):
    def decorator(func):
        def wrapped_func(*args, **kwargs):
            print('装饰器想告诉你：{}'.format(message))
            return func(*args, **kwargs)
        return wrapped_func
    return decorator

@test()
print(test())
test()
```

装饰器想告诉你：Hello World

### 重要提示：

使用这样的装饰器工厂时，必须用一对括号调用装饰器：

```
@decoratorfactory # 无括号情况下
def test():
    pass

test()
```

TypeError: decorator() 缺少 1 个必需的位置参数: 'func'

### 装饰器类

```
def decoratorfactory(*decorator_args, **decorator_kwargs):

    class Decorator(object):
        def __init__(self, func):
            self.func = func

        def __call__(self, *args, **kwargs):
            print('带参数的装饰器内部 {}'.format(decorator_args))
            return self.func(*args, **kwargs)

    return Decorator

@decoratorfactory(10)
def test():
    pass

test()
```

带参数的装饰器内部 (10,)

and returns a decorator.

### Decorator functions

```
def decoratorfactory(message):
    def decorator(func):
        def wrapped_func(*args, **kwargs):
            print('The decorator wants to tell you: {}'.format(message))
            return func(*args, **kwargs)
        return wrapped_func
    return decorator

@test()
print(test())
test()
```

The decorator wants to tell you: Hello World

### Important Note:

With such decorator factories you **must** call the decorator with a pair of parentheses:

```
@decoratorfactory # Without parentheses
def test():
    pass

test()
```

TypeError: decorator() missing 1 required positional argument: 'func'

### Decorator classes

```
def decoratorfactory(*decorator_args, **decorator_kwargs):

    class Decorator(object):
        def __init__(self, func):
            self.func = func

        def __call__(self, *args, **kwargs):
            print('Inside the decorator with arguments {}'.format(decorator_args))
            return self.func(*args, **kwargs)

    return Decorator

@decoratorfactory(10)
def test():
    pass

test()
```

Inside the decorator with arguments (10,)

## 第37.4节：让装饰器看起来像被装饰的函数

装饰器通常会剥离函数的元数据，因为它们不是同一个函数。这在使用元编程动态访问函数元数据时可能会导致问题。元数据还包括函数的文档字符串和名称。

`functools.wraps` 通过将多个属性复制到包装函数，使被装饰的函数看起来像原始函数。

```
from functools import wraps
```

这两种包装装饰器的方法都实现了隐藏原始函数已被装饰的效果。除非你已经在使用其中一种，否则没有理由偏好函数版本而非类版本。

### 作为一个函数

```
def decorator(func):
    # 将文档字符串、名称、注解和模块复制到装饰器
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped_func

@decorator
def test():
    pass

test.__name__
```

```
'test'
```

### 作为一个班级

```
class Decorator(object):
    def __init__(self, func):
        # 将名称、模块、注释和文档字符串复制到实例中。
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@装饰器
def test():
    """测试的文档字符串。””
    pass

test.__doc__
```

```
'测试的文档字符串。'
```

## 第37.5节：使用装饰器计时函数

```
import time
def timer(func):
    def inner(*args, **kwargs):
```

## Section 37.4: Making a decorator look like the decorated function

Decorators normally strip function metadata as they aren't the same. This can cause problems when using metaprogramming to dynamically access function metadata. Metadata also includes function's docstrings and its name. `functools.wraps` makes the decorated function look like the original function by copying several attributes to the wrapper function.

```
from functools import wraps
```

The two methods of wrapping a decorator are achieving the same thing in hiding that the original function has been decorated. There is no reason to prefer the function version to the class version unless you're already using one over the other.

### As a function

```
def decorator(func):
    # Copies the docstring, name, annotations and module to the decorator
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped_func

@decorator
def test():
    pass

test.__name__
```

```
'test'
```

### As a class

```
class Decorator(object):
    def __init__(self, func):
        # Copies name, module, annotations and docstring to the instance.
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring of test."""
    pass

test.__doc__
```

```
'Docstring of test.'
```

## Section 37.5: Using a decorator to time a function

```
import time
def timer(func):
    def inner(*args, **kwargs):
```

```

t1 = time.time()
f = func(*args, **kwargs)
t2 = time.time()
print 'Runtime took {0} seconds'.format(t2-t1)
return f
return inner

@timer
def example_function():
    #do stuff

example_function()

```

## 第37.6节：使用装饰器创建单例类

单例是一种限制类实例化为唯一实例/对象的模式。通过使用装饰器，我们可以 define a class as a singleton by forcing the class to either return an existing instance of the class or create a new instance (if it doesn't exist).

```

def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper

```

该装饰器可以添加到任何类声明中，并确保最多只创建该类的一个实例。任何后续调用都会返回已存在的类实例。

```

@singleton
class SomeSingletonClass:
    x = 2
    def __init__(self):
        print("Created!")

instance = SomeSingletonClass() # 输出: Created!
instance = SomeSingletonClass() # 不会打印任何内容
print(instance.x)             # 2

instance.x = 3
print(SomeSingletonClass().x)  # 3

```

所以无论你是通过本地变量引用类实例，还是创建另一个“实例”，你总是得到同一个对象。

```

t1 = time.time()
f = func(*args, **kwargs)
t2 = time.time()
print 'Runtime took {0} seconds'.format(t2-t1)
return f
return inner

@timer
def example_function():
    #do stuff

example_function()

```

## Section 37.6: Create singleton class with a decorator

A singleton is a pattern that restricts the instantiation of a class to one instance/object. Using a decorator, we can define a class as a singleton by forcing the class to either return an existing instance of the class or create a new instance (if it doesn't exist).

```

def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper

```

This decorator can be added to any class declaration and will make sure that at most one instance of the class is created. Any subsequent calls will return the already existing class instance.

```

@singleton
class SomeSingletonClass:
    x = 2
    def __init__(self):
        print("Created!")

instance = SomeSingletonClass() # prints: Created!
instance = SomeSingletonClass() # doesn't print anything
print(instance.x)             # 2

instance.x = 3
print(SomeSingletonClass().x)  # 3

```

So it doesn't matter whether you refer to the class instance via your local variable or whether you create another "instance", you always get the same object.

# 第38章：类

Python不仅作为一种流行的脚本语言，还支持面向对象编程范式。类描述数据并提供操作这些数据的方法，所有这些都包含在一个单一的对象中。此外，类通过将具体的实现细节与数据的抽象表示分离，实现了抽象。

使用类的代码通常更易于阅读、理解和维护。

## 第38.1节：类简介

类作为一个模板，定义了特定对象的基本特征。以下是一个示例：

```
class Person(object):
    """一个简单的类。"""
    # 文档字符串
    species = "智人" # 类属性

    def __init__(self, name):
        """这是初始化方法。它是一个特殊的方法（见下文）。
        """
        self.name = name # 实例属性

    def __str__(self):
        """当Python尝试
        将对象转换为字符串时运行此方法。使用print()等时返回
        该字符串。
        """
        return self.name

    def rename(self, renamed):
        """重新赋值并打印name属性。"""
        self.name = renamed
        print("现在我的名字是 {}".format(self.name))
```

在查看上述示例时，有几点需要注意。

1. 该类由属性（数据）和方法（函数）组成。
2. 属性和方法仅仅被定义为普通变量和函数。
3. 如相应的文档字符串中所述，`__init__()` 方法被称为初始化器。它等同于构造函数在其他面向对象语言中，是当你创建一个新对象或类的新实例时首先运行的方法。
4. 适用于整个类的属性首先被定义，称为类属性。
5. 适用于类的特定实例（对象）的属性称为实例属性。它们是通常在`__init__()`内定义；这不是必须的，但建议这样做（因为在`__init__()`之外定义的属性有可能在被定义之前就被访问）。
6. 类定义中包含的每个方法都将相关对象作为其第一个参数传递。该词 `self` 用于此参数（实际上使用`self`是约定俗成的，因为单词`self`在Python中没有固有含义，但这是Python最受尊重的约定之一，您应始终遵循它）。
7. 习惯于其他语言面向对象编程的人可能会对一些事情感到惊讶。其中之一是Python没有真正的“私有”元素的概念，因此默认情况下，所有内容都模仿C++/Java中“public”关键字的行为。更多信息请参见本页的“私有类成员”示例。
8. 该类的一些方法具有以下形式：`__functionname__(self, other_stuff)`。所有这些方法被称为“魔术方法”，是Python类的重要组成部分。例如，Python中的运算符重载就是通过魔术方法实现的。更多信息请参见相关

# Chapter 38: Classes

Python offers itself not only as a popular scripting language, but also supports the object-oriented programming paradigm. Classes describe data and provide methods to manipulate that data, all encompassed under a single object. Furthermore, classes allow for abstraction by separating concrete implementation details from abstract representations of data.

Code utilizing classes is generally easier to read, understand, and maintain.

## Section 38.1: Introduction to classes

A class, functions as a template that defines the basic characteristics of a particular object. Here's an example:

```
class Person(object):
    """A simple class."""
    species = "Homo Sapiens" # docstring
                           # class attribute

    def __init__(self, name):
        """This is the initializer. It's a special
        method (see below).
        """
        self.name = name # special method
                         # instance attribute

    def __str__(self):
        """This method is run when Python tries
        to cast the object to a string. Return
        this string when using print(), etc.
        """
        return self.name # special method

    def rename(self, renamed):
        """Reassign and print the name attribute."""
        self.name = renamed
        print("Now my name is {}".format(self.name)) # regular method
```

There are a few things to note when looking at the above example.

1. The class is made up of *attributes* (data) and *methods* (functions).
2. Attributes and methods are simply defined as normal variables and functions.
3. As noted in the corresponding docstring, the `__init__()` method is called the *initializer*. It's equivalent to the constructor in other object oriented languages, and is the method that is first run when you create a new object, or new instance of the class.
4. Attributes that apply to the whole class are defined first, and are called *class attributes*.
5. Attributes that apply to a specific instance of a class (an object) are called *instance attributes*. They are generally defined inside `__init__()`; this is not necessary, but it is recommended (since attributes defined outside of `__init__()` run the risk of being accessed before they are defined).
6. Every method, included in the class definition passes the object in question as its first parameter. The word `self` is used for this parameter (usage of `self` is actually by convention, as the word `self` has no inherent meaning in Python, but this is one of Python's most respected conventions, and you should always follow it).
7. Those used to object-oriented programming in other languages may be surprised by a few things. One is that Python has no real concept of private elements, so everything, by default, imitates the behavior of the C++/Java public keyword. For more information, see the "Private Class Members" example on this page.
8. Some of the class's methods have the following form: `__functionname__(self, other_stuff)`. All such methods are called "magic methods" and are an important part of classes in Python. For instance, operator overloading in Python is implemented with magic methods. For more information, see the relevant

文档。

现在让我们创建几个Person类的实例！

```
>>> # 实例
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

我们目前有三个Person对象，分别是kelly、joseph和john\_doe。

我们可以通过点操作符.访问每个实例的类属性。再次注意类属性和实例属性之间的区别：

```
>>> # 属性
>>> kelly.species
'Human'
>>> john_doe.species
'Human'
>>> joseph.species
'Human'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

我们可以使用相同的点操作符.来执行类的方法：

```
>>> # 方法
>>> john_doe.__str__()
'John Doe'
>>> print(john_doe)
'John Doe'
>>> john_doe.rename("John")
'现在我的名字是 John'
```

## 第38.2节：绑定方法、未绑定方法和静态方法

绑定方法和未绑定方法的概念在Python 3中被移除了。在Python 3中，当你在类中声明一个方法时，使用的是def关键字，从而创建了一个函数对象。这是一个普通函数，外围的类作为它的命名空间。在下面的例子中，我们在类A中声明了方法f，它变成了函数A.f：

```
Python 3.x 版本 ≥ 3.0
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...> (in Python 3.x)
```

在 Python 2 中，行为有所不同：类中的函数对象会被隐式替换为类型为instancemethod的对象，这些对象被称为unbound methods（未绑定方法），因为它们没有绑定到任何特定的类实例。可以通过.\_\_func\_\_属性访问底层函数。

```
Python 2.x 版本 ≥ 2.3
A.f
```

documentation.

Now let's make a few instances of our Person class!

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

We currently have three Person objects, kelly, joseph, and john\_doe.

We can access the attributes of the class from each instance using the dot operator . Note again the difference between class and instance attributes:

```
>>> # Attributes
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

We can execute the methods of the class using the same dot operator .:

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>> print(john_doe)
'John Doe'
>>> john_doe.rename("John")
'Now my name is John'
```

## Section 38.2: Bound, unbound, and static methods

The idea of bound and unbound methods was [removed in Python 3](#). In Python 3 when you declare a method within a class, you are using a def keyword, thus creating a function object. This is a regular function, and the surrounding class works as its namespace. In the following example we declare method f within class A, and it becomes a function A.f:

```
Python 3.x Version ≥ 3.0
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...> (in Python 3.x)
```

In Python 2 the behavior was different: function objects within the class were implicitly replaced with objects of type instancemethod, which were called *unbound methods* because they were not bound to any particular class instance. It was possible to access the underlying function using \_\_func\_\_ property.

```
Python 2.x Version ≥ 2.3
A.f
```

```
# <unbound method A.f> (在 Python 2.x 中)
A.f.__class__
# <type 'instancemethod'>
A.f.__func__
# <function f at ...>
```

后者的行为通过检查得以确认——在 Python 3 中，方法被识别为函数，而在 Python 2 中则保持区分。

Python 3.x 版本 ≥ 3.0

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# 假
```

Python 2.x 版本 ≥ 2.3

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# 真
```

在两个版本的Python中，函数/方法A.f都可以直接调用，前提是传入类A的一个实例作为第一个参数。

```
A.f(1, 7)
# Python 2: TypeError: unbound method f() must be called with
#           A instance as first argument (got int instance instead)
# Python 3: 14
a = A()
A.f(a, 20)
# Python 2 & 3: 40
```

现在假设 a 是类A的一个实例，那么 a.f 是什么呢？直观上，这应该是类A的同一个方法f，只不过它应该以某种方式“知道”它是应用于对象 a 的——在Python中，这被称为绑定到 a 的方法。A，只是它应该以某种方式“知道”它被应用到了对象 a——在Python中，这称为绑定到 a 的方法。

具体细节如下：写入 a.f 会调用 a 的魔法方法 `__getattribute__`，首先检查 a 是否有名为 f 的属性（没有），然后检查类 A 是否包含名为 f 的方法（有），并创建一个类型为 `method` 的新对象 m，m.`__func__` 中引用了原始的 A.f，m.`__self__` 中引用了对象 a。当该对象作为函数被调用时，它执行以下操作：`m(...)` => `m.__func__(m.__self__, ...)`。因此，该对象被称为 绑定方法，因为调用时它会自动将绑定的对象作为第一个参数传入。（这些机制在 Python 2 和 3 中是相同的）。

```
a = A()
a.f
# <bound method A.f of <__main__.A object at ...>>
a.f(2)
# 4

# 注意：绑定方法对象 a.f 在每次调用时都会重新创建：
a.f is a.f # False
# 作为性能优化，你可以将绑定方法存储在对象的
# __dict__ 中，这样方法对象将保持不变：
a.f = a.f
```

```
# <unbound method A.f> (in Python 2.x)
A.f.__class__
# <type 'instancemethod'>
A.f.__func__
# <function f at ...>
```

The latter behaviors are confirmed by inspection - methods are recognized as functions in Python 3, while the distinction is upheld in Python 2.

Python 3.x 版本 ≥ 3.0

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# False
```

Python 2.x 版本 ≥ 2.3

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# True
```

In both versions of Python function/method A.f can be called directly, provided that you pass an instance of class A as the first argument.

```
A.f(1, 7)
# Python 2: TypeError: unbound method f() must be called with
#           A instance as first argument (got int instance instead)
# Python 3: 14
a = A()
A.f(a, 20)
# Python 2 & 3: 40
```

Now suppose a is an instance of class A, what is a.f then? Well, intuitively this should be the same method f of class A, only it should somehow "know" that it was applied to the object a – in Python this is called method *bound to a*.

The nitty-gritty details are as follows: writing a.f invokes the magic `__getattribute__` method of a, which first checks whether a has an attribute named f (it doesn't), then checks the class A whether it contains a method with such a name (it does), and creates a new object m of type `method` which has the reference to the original A.f in `m.__func__`, and a reference to the object a in `m.__self__`. When this object is called as a function, it simply does the following: `m(...)` => `m.__func__(m.__self__, ...)`. Thus this object is called a **bound method** because when invoked it knows to supply the object it was bound to as the first argument. (These things work same way in Python 2 and 3).

```
a = A()
a.f
# <bound method A.f of <__main__.A object at ...>>
a.f(2)
# 4
```

```
# Note: the bound method object a.f is recreated *every time* you call it:
a.f is a.f # False
# As a performance optimization you can store the bound method in the object's
# __dict__, in which case the method object will remain fixed:
a.f = a.f
```

```
a.f is a.f # True
```

最后，Python 有类方法和静态方法——特殊类型的方法。类方法的工作方式与普通方法相同，除了当它们在对象上调用时，它们绑定的是对象的类而不是对象本身。因此 `m.__self__ = type(a)`。当你调用这样的绑定方法时，它会将 `a` 的类作为第一个参数传入。静态方法更简单：它们根本不绑定任何东西，只是直接返回底层函数，没有任何转换。

```
class D(object):
multiplier = 2

@classmethod
def f(cls, x):
    return cls.multiplier * x

@staticmethod
def g(name):
    print("Hello, %s" % name)

D.f
# <bound method type.f of <class '__main__.D'>>
D.f(12)
# 24
D.g
# <function D.g at ...>
D.g("world")
# Hello, world
```

请注意，即使在实例上访问，类方法仍绑定到类：

```
d = D()
d.multiplier = 1337
(D.multiplier, d.multiplier)
# (2, 1337)
d.f
# <bound method D.f of <class '__main__.D'>>
d.f(10)
# 20
```

值得注意的是，在最低层级，函数、方法、静态方法等实际上是描述符，它们会调用`__get__`、`__set__`以及可选的`__del__`特殊方法。关于类方法和静态方法的更多细节：

- [Python中@staticmethod和@classmethod有什么区别？](#)
- [初学者如何理解@classmethod和@staticmethod的含义？](#)

### 第38.3节：基本继承

Python中的继承基于与Java、C++等其他面向对象语言类似的思想。一个新类可以从现有类派生，方式如下。

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

```
a.f is a.f # True
```

Finally, Python has **class methods** and **static methods** – special kinds of methods. Class methods work the same way as regular methods, except that when invoked on an object they bind to the *class* of the object instead of to the object. Thus `m.__self__ = type(a)`. When you call such bound method, it passes the class of `a` as the first argument. Static methods are even simpler: they don't bind anything at all, and simply return the underlying function without any transformations.

```
class D(object):
    multiplier = 2

@classmethod
def f(cls, x):
    return cls.multiplier * x

@staticmethod
def g(name):
    print("Hello, %s" % name)

D.f
# <bound method type.f of <class '__main__.D'>>
D.f(12)
# 24
D.g
# <function D.g at ...>
D.g("world")
# Hello, world
```

Note that class methods are bound to the class even when accessed on the instance:

```
d = D()
d.multiplier = 1337
(D.multiplier, d.multiplier)
# (2, 1337)
d.f
# <bound method D.f of <class '__main__.D'>>
d.f(10)
# 20
```

It is worth noting that at the lowest level, functions, methods, staticmethods, etc. are actually descriptors that invoke `__get__`, `__set__` and optionally `__del__` special methods. For more details on classmethods and staticmethods:

- [What is the difference between @staticmethod and @classmethod in Python?](#)
- [Meaning of @classmethod and @staticmethod for beginner?](#)

### Section 38.3: Basic inheritance

Inheritance in Python is based on similar ideas used in other object oriented languages like Java, C++ etc. A new class can be derived from an existing class as follows.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

BaseClass 是已经存在的（父类）类，而 DerivedClass 是继承（或子类化）自 BaseClass 属性的新（子类）类。注意：从 Python 2.2 开始，所有类都隐式继承自 object 类，object 是所有内置类型的基类。

我们在下面的示例中定义了一个父类 Rectangle，该类隐式继承自 object：

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)
```

由于正方形是矩形的一种特殊情况，Rectangle类可以用作定义Square类的基类。

```
class Square(Rectangle):
    def __init__(self, s):
        # 调用父类构造函数，宽和高均为 s
        super(Square, self).__init__(s, s)
        self.s = s
```

Square 类将自动继承 Rectangle 类以及 object 类的所有属性。super() 用于调用 Rectangle 类的 \_\_init\_\_() 方法，实质上是调用基类中被重写的方法。

注意：在 Python 3 中，super() 不需要参数。

派生类对象可以访问并修改其基类的属性：

```
r.area()
# 输出: 12
r.perimeter()
# 输出: 14

s.area()
# 输出: 4
s.perimeter()
# 输出: 8
```

## 内置函数与继承相关

`issubclass(DerivedClass, BaseClass)`: 如果 DerivedClass 是 BaseClass 的子类，则返回 True

`isinstance(s, Class)`: 如果 s 是 Class 或其派生类的实例，则返回 True

```
# 子类检查
issubclass(Square, Rectangle)
# 输出: True

# 实例化
r = Rectangle(3, 4)
s = Square(2)

isinstance(r, Rectangle)
# 输出: True
isinstance(r, Square)
```

The BaseClass is the already existing (*parent*) class, and the DerivedClass is the new (*child*) class that inherits (or *subclasses*) attributes from BaseClass. **Note:** As of Python 2.2, all [classes implicitly inherit from the object class](#), which is the base class for all built-in types.

We define a parent Rectangle class in the example below, which implicitly inherits from object:

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)
```

The Rectangle class can be used as a base class for defining a Square class, as a square is a special case of rectangle.

```
class Square(Rectangle):
    def __init__(self, s):
        # call parent constructor, w and h are both s
        super(Square, self).__init__(s, s)
        self.s = s
```

The Square class will automatically inherit all attributes of the Rectangle class as well as the object class. `super()` is used to call the `__init__()` method of Rectangle class, essentially calling any overridden method of the base class.

**Note:** in Python 3, `super()` does not require arguments.

Derived class objects can access and modify the attributes of its base classes:

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

## Built-in functions that work with inheritance

`issubclass(DerivedClass, BaseClass)`: returns True if DerivedClass is a subclass of the BaseClass

`isinstance(s, Class)`: returns True if s is an instance of Class or any of the derived classes of Class

```
# subclass check
issubclass(Square, Rectangle)
# Output: True

# instantiate
r = Rectangle(3, 4)
s = Square(2)

isinstance(r, Rectangle)
# Output: True
isinstance(r, Square)
```

```
# 输出: False  
# 一个矩形不是正方形  
  
isinstance(s, Rectangle)  
# 输出: True  
# 一个正方形是矩形  
isinstance(s, Square)  
# 输出: True
```

```
# Output: False  
# A rectangle is not a square  
  
isinstance(s, Rectangle)  
# Output: True  
# A square is a rectangle  
isinstance(s, Square)  
# Output: True
```

## 第38.4节：猴子补丁

在这种情况下，“猴子补丁”指的是在类定义之后，向类添加新的变量或方法。例如，假设我们定义了类A为

```
class A(object):  
    def __init__(self, num):  
        self.num = num  
  
    def __add__(self, other):  
        return A(self.num + other.num)
```

但现在我们想在代码后面添加另一个函数。假设这个函数如下。

```
def get_num(self):  
    return self.num
```

但是我们如何将其作为方法添加到A中呢？这很简单，我们只需通过赋值语句将该函数放入A中即可。

```
A.get_num = get_num
```

为什么这样做有效？因为函数和其他对象一样本身就是对象，而方法是属于类的函数。

函数get\_num将对所有现有（已创建的）以及新创建的A实例可用。

这些添加会自动对该类（或其子类）的所有实例生效。例如：

```
foo = A(42)  
  
A.get_num = get_num  
  
bar = A(6);  
  
foo.get_num() # 42  
  
bar.get_num() # 6
```

请注意，与其他一些语言不同，这种技术对某些内置类型无效，并且不被认为是良好的编程风格。

## 第38.5节：新式类与旧式类

Python 2.x 版本 ≥ 2.2.0

新式类是在Python 2.2中引入的，用于统一类和类型。它们继承自顶层的object

## Section 38.4: Monkey Patching

In this case, "monkey patching" means adding a new variable or method to a class after it's been defined. For instance, say we defined class A as

```
class A(object):  
    def __init__(self, num):  
        self.num = num  
  
    def __add__(self, other):  
        return A(self.num + other.num)
```

But now we want to add another function later in the code. Suppose this function is as follows.

```
def get_num(self):  
    return self.num
```

But how do we add this as a method in A? That's simple we just essentially place that function into A with an assignment statement.

```
A.get_num = get_num
```

Why does this work? Because functions are objects just like any other object, and methods are functions that belong to the class.

The function get\_num shall be available to all existing (already created) as well to the new instances of A

These additions are available on all instances of that class (or its subclasses) automatically. For example:

```
foo = A(42)  
  
A.get_num = get_num  
  
bar = A(6);  
  
foo.get_num() # 42  
  
bar.get_num() # 6
```

Note that, unlike some other languages, this technique does not work for certain built-in types, and it is not considered good style.

## Section 38.5: New-style vs. old-style classes

Python 2.x Version ≥ 2.2.0

New-style classes were introduced in Python 2.2 to unify *classes* and *types*. They inherit from the top-level object

类型。新式类是用户定义的类型，与内置类型非常相似。

```
# 新式类
class New(object):
    pass

# 新式实例
new = New()

new.__class__
# <class '__main__.New'>
类型(新)
# <class '__main__.New'>
issubclass(New, object)
# 真
```

旧式类不继承自`object`。旧式实例总是用内置的`instance`

type实现。

```
# 旧式类
class Old:
    pass

# 旧式实例
old = Old()

old.__class__
# <class '__main__.Old' at ...>
type(old)
# <type 'instance'>
issubclass(Old, object)
# 假
```

Python 3.x 版本 ≥ 3.0.0

在 Python 3 中，旧式类被移除了。

Python 3 中的新式类隐式继承自`object`，因此不再需要指定`MyClass(object)`。

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# 真
```

## 第38.6节：类方法：替代初始化器

类方法提供了构建类实例的替代方式。为说明这一点，我们来看一个例子。

假设我们有一个相对简单的Person类：

```
class Person(object):
```

type. A new-style class is a user-defined type, and is very similar to built-in types.

```
# new-style class
class New(object):
    pass

# new-style instance
new = New()

new.__class__
# <class '__main__.New'>
type(new)
# <class '__main__.New'>
issubclass(New, object)
# True
```

Old-style classes do **not** inherit from `object`. Old-style instances are always implemented with a built-in `instance` type.

```
# old-style class
class Old:
    pass

# old-style instance
old = Old()

old.__class__
# <class '__main__.Old' at ...>
type(old)
# <type 'instance'>
issubclass(Old, object)
# False
```

Python 3.x Version ≥ 3.0.0

In Python 3, old-style classes were removed.

New-style classes in Python 3 implicitly inherit from `object`, so there is no need to specify `MyClass(object)` anymore.

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

## Section 38.6: Class methods: alternate initializers

Class methods present alternate ways to build instances of classes. To illustrate, let's look at an example.

Let's suppose we have a relatively simple Person class:

```
class Person(object):
```

```

def __init__(self, first_name, last_name, age):
    self.first_name = first_name
    self.last_name = last_name
    self.age = age
    self.full_name = first_name + " " + last_name

def greet(self):
    print("Hello, my name is " + self.full_name + ".")

```

有时可能需要通过指定全名而不是分别指定名字和姓氏来创建该类的实例。一种方法是将last\_name设为可选参数，并假设如果未提供该参数，则传入的是全名：

```

class Person(object):

    def __init__(self, first_name, age, last_name=None):
        if last_name is None:
            self.first_name, self.last_name = first_name.split(" ", 2)
        else:
            self.first_name = first_name
            self.last_name = last_name

        self.full_name = self.first_name + " " + self.last_name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")

```

但是，这段代码存在两个主要问题：

1. 参数first\_name和last\_name现在具有误导性，因为你可以在first\_name中输入全名  
此外，如果有更多情况和/或更多具有这种灵活性的参数，  
if/elif/else 分支会变得非常烦人。
2. 虽然不那么重要，但仍值得指出：如果last\_name是None，但first\_name没有通过空格拆分成两个或更多部分怎么办？  
我们又面临一层输入验证和/或异常处理的问题.....

引入类方法。我们不会只有一个初始化方法，而是创建一个单独的初始化方法，称为from\_full\_name，并用内置的classmethod装饰器装饰它。

```

class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    @classmethod
    def from_full_name(cls, name, age):
        if " " not in name:
            raise ValueError
        first_name, last_name = name.split(" ", 2)
        return cls(first_name, last_name, age)

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")

```

```

def __init__(self, first_name, last_name, age):
    self.first_name = first_name
    self.last_name = last_name
    self.age = age
    self.full_name = first_name + " " + last_name

def greet(self):
    print("Hello, my name is " + self.full_name + ".")

```

It might be handy to have a way to build instances of this class specifying a full name instead of first and last name separately. One way to do this would be to have last\_name be an optional parameter, and assuming that if it isn't given, we passed the full name in:

```

class Person(object):

    def __init__(self, first_name, age, last_name=None):
        if last_name is None:
            self.first_name, self.last_name = first_name.split(" ", 2)
        else:
            self.first_name = first_name
            self.last_name = last_name

        self.full_name = self.first_name + " " + self.last_name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")

```

However, there are two main problems with this bit of code:

1. The parameters first\_name and last\_name are now misleading, since you can enter a full name for first\_name. Also, if there are more cases and/or more parameters that have this kind of flexibility, the if/elif/else branching can get annoying fast.
2. Not quite as important, but still worth pointing out: what if last\_name is `None`, but first\_name doesn't split into two or more things via spaces? We have yet another layer of input validation and/or exception handling...

Enter class methods. Rather than having a single initializer, we will create a separate initializer, called `from_full_name`, and decorate it with the (built-in) `classmethod` decorator.

```

class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    @classmethod
    def from_full_name(cls, name, age):
        if " " not in name:
            raise ValueError
        first_name, last_name = name.split(" ", 2)
        return cls(first_name, last_name, age)

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")

```

注意cls而不是self作为from\_full\_name的第一个参数。类方法应用于整个类，而不是某个类的实例（通常由self表示）。因此，如果cls是我们的Person类，那么from\_full\_name类方法返回的值是Person(first\_name, last\_name, age)，使用了Person的\_\_init\_\_来创建Person类的实例。特别是，如果我们创建了Person的子类Employee，那么from\_full\_name在Employee类中也能正常工作。

为了证明这按预期工作，让我们在没有\_init\_中分支的情况下，用多种方式创建Person的实例：

```
In [2]: bob = Person("Bob", "Bobberson", 42)
```

```
In [3]: alice = Person.from_full_name("Alice Henderson", 31)
```

```
In [4]: bob.greet()
```

你好，我的名字是Bob Bobberson。

```
In [5]: alice.greet()
```

你好，我的名字是Alice Henderson。

其他参考资料：

- [Python @classmethod 和 @staticmethod 入门？](#)
- <https://docs.python.org/2/library/functions.html#classmethod>
- <https://docs.python.org/3.5/library/functions.html#classmethod>

## 第38.7节：多重继承

Python使用C3线性化算法来确定解析类属性（包括方法）的顺序。这被称为方法解析顺序（MRO）。

这是一个简单的例子：

```
class Foo(object):  
    foo = 'Foo的属性foo'
```

```
class Bar(object):  
    foo = 'Bar的属性foo' # 我们不会看到这个。  
    bar = 'Bar的属性bar'
```

```
class FooBar(Foo, Bar):  
    foobar = 'FooBar的属性foobar'
```

现在如果我们实例化FooBar，查找foo属性时，会先找到Foo的属性

```
fb = FooBar()
```

和

```
>>> fb.foo  
'Foo的属性foo'
```

这是FooBar的MRO（方法解析顺序）：

Notice cls instead of `self` as the first argument to `from_full_name`. Class methods are applied to the overall class, not an instance of a given class (which is what `self` usually denotes). So, if `cls` is our `Person` class, then the returned value from the `from_full_name` class method is `Person(first_name, last_name, age)`, which uses `Person's __init__` to create an instance of the `Person` class. In particular, if we were to make a subclass `Employee` of `Person`, then `from_full_name` would work in the `Employee` class as well.

To show that this works as expected, let's create instances of `Person` in more than one way without the branching in `__init__`:

```
In [2]: bob = Person("Bob", "Bobberson", 42)
```

```
In [3]: alice = Person.from_full_name("Alice Henderson", 31)
```

```
In [4]: bob.greet()
```

Hello, my name `is` Bob Bobberson.

```
In [5]: alice.greet()
```

Hello, my name `is` Alice Henderson.

Other references:

- [Python @classmethod and @staticmethod for beginner?](#)
- <https://docs.python.org/2/library/functions.html#classmethod>
- <https://docs.python.org/3.5/library/functions.html#classmethod>

## Section 38.7: Multiple Inheritance

Python uses the [C3 linearization](#) algorithm to determine the order in which to resolve class attributes, including methods. This is known as the Method Resolution Order (MRO).

Here's a simple example:

```
class Foo(object):  
    foo = 'attr foo of Foo'
```

```
class Bar(object):  
    foo = 'attr foo of Bar' # we won't see this.  
    bar = 'attr bar of Bar'
```

```
class FooBar(Foo, Bar):  
    foobar = 'attr foobar of FooBar'
```

Now if we instantiate `FooBar`, if we look up the `foo` attribute, we see that `Foo`'s attribute is found first

```
fb = FooBar()
```

and

```
>>> fb.foo  
'attr foo of Foo'
```

Here's the MRO of `FooBar`:

```
>>> FooBar.mro()
[<类 '__main__.FooBar'>, <类 '__main__.Foo'>, <类 '__main__.Bar'>, <类型 'object'>]
```

可以简单地说，Python的MRO算法是

1. 深度优先（例如FooBar然后Foo），除非
2. 一个共享的父类（object）被子类（Bar）阻塞，且
3. 不允许循环继承关系。

也就是说，例如，Bar不能继承自FooBar，而FooBar又继承自Bar。

有关 Python 的综合示例，请参见 [wikipedia 条目](#)。

继承中的另一个强大特性是 super。super 可以调用父类的功能。

```
class Foo(object):
    def foo_method(self):
        print "foo 方法"

class Bar(object):
    def bar_method(self):
        print "bar 方法"

class FooBar(Foo, Bar):
    def foo_method(self):
        super(FooBar, self).foo_method()
```

多重继承中如果每个类都有自己的 init 方法，当尝试多重继承时，只有第一个继承的类的 init 方法会被调用。

以下示例中，只有 Foo 类的 init 方法被调用，Bar 类的 init 方法未被调用

```
class Foo(object):
    def __init__(self):
        print "foo init"

class Bar(object):
    def __init__(self):
        打印"bar init"

class FooBar(Foo, Bar):
    def __init__(self):
        print "foobar 初始化"
        super(FooBar, self).__init__()

a = FooBar()
```

输出：

```
foobar 初始化
foo 初始化
```

但这并不意味着Bar类没有被继承。最终的**FooBar**类的实例也是Bar类和**Foo**

类的实例。

```
print isinstance(a, FooBar)
print isinstance(a, Foo)
```

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

It can be simply stated that Python's MRO algorithm is

1. Depth first (e.g. FooBar then Foo) unless
2. a shared parent (**object**) is blocked by a child (Bar) and
3. no circular relationships allowed.

That is, for example, Bar cannot inherit from FooBar while FooBar inherits from Bar.

For a comprehensive example in Python, see the [wikipedia entry](#).

Another powerful feature in inheritance is **super**. super can fetch parent classes features.

```
class Foo(object):
    def foo_method(self):
        print "foo Method"

class Bar(object):
    def bar_method(self):
        print "bar Method"

class FooBar(Foo, Bar):
    def foo_method(self):
        super(FooBar, self).foo_method()
```

Multiple inheritance with init method of class, when every class has own init method then we try for multiple inheritance then only init method get called of class which is inherit first.

for below example only Foo class **init** method getting called **Bar** class init not getting called

```
class Foo(object):
    def __init__(self):
        print "foo init"

class Bar(object):
    def __init__(self):
        打印"bar init"

class FooBar(Foo, Bar):
    def __init__(self):
        print "foobar init"
        super(FooBar, self).__init__()

a = FooBar()
```

Output:

```
foobar init
foo init
```

But it doesn't mean that **Bar** class is not inherit. Instance of final **FooBar** class is also instance of **Bar** class and **Foo** class.

```
print isinstance(a, FooBar)
print isinstance(a, Foo)
```

```
print isinstance(a,Bar)
```

输出：

```
True  
True  
True
```

## 第38.8节：属性

Python类支持属性（properties），看起来像普通的对象变量，但可以附加自定义行为和文档说明。

```
class MyClass(object):  
  
    def __init__(self):  
        self._my_string = ""  
  
    @property  
    def string(self):  
        """一个极其重要的字符串。"""  
        return self._my_string  
  
    @string.setter  
    def string(self, new_value):  
        assert isinstance(new_value, str), \  
            "给我一个字符串，而不是 %r!" % type(new_value)  
        self._my_string = new_value  
  
    @string.deleter  
    def x(self):  
        self._my_string = None
```

类MyClass的对象看起来会有一个属性.string，然而它的行为现在被严格控制：

```
mc = MyClass()  
mc.string = "String!"  
print(mc.string)  
del mc.string
```

除了上述有用的语法外，属性语法还允许对这些属性进行验证或添加其他增强功能。这在公共API中尤其有用——应该为用户提供一定程度的帮助。

属性的另一个常见用途是使类能够呈现“虚拟属性”——这些属性实际上并未存储，而仅在请求时计算。

```
class Character(object):  
    def __init__(name, max_hp):  
        self._name = name  
        self._hp = max_hp  
        self._max_hp = max_hp  
  
    # 通过不提供设置方法使hp只读  
    @property  
    def hp(self):  
        return self._hp
```

```
print isinstance(a,Bar)
```

Output:

```
True  
True  
True
```

## Section 38.8: Properties

Python classes support **properties**, which look like regular object variables, but with the possibility of attaching custom behavior and documentation.

```
class MyClass(object):  
  
    def __init__(self):  
        self._my_string = ""  
  
    @property  
    def string(self):  
        """A profoundly important string."""  
        return self._my_string  
  
    @string.setter  
    def string(self, new_value):  
        assert isinstance(new_value, str), \  
            "Give me a string, not a %r!" % type(new_value)  
        self._my_string = new_value  
  
    @string.deleter  
    def x(self):  
        self._my_string = None
```

The object's of class MyClass will appear to have a property .string, however it's behavior is now tightly controlled:

```
mc = MyClass()  
mc.string = "String!"  
print(mc.string)  
del mc.string
```

As well as the useful syntax as above, the property syntax allows for validation, or other augmentations to be added to those attributes. This could be especially useful with public APIs - where a level of help should be given to the user.

Another common use of properties is to enable the class to present 'virtual attributes' - attributes which aren't actually stored but are computed only when requested.

```
class Character(object):  
    def __init__(name, max_hp):  
        self._name = name  
        self._hp = max_hp  
        self._max_hp = max_hp  
  
    # Make hp read only by not providing a set method  
    @property  
    def hp(self):  
        return self._hp
```

```

# 通过不提供设置方法使名称只读
@property
def name(self):
    return self.name

def take_damage(self, damage):
    self.hp -= damage
    self.hp = 0 if self.hp < 0 else self.hp

@property
def is_alive(self):
    return self.hp != 0

@property
def is_wounded(self):
    return self.hp < self.max_hp if self.hp > 0 else False

@property
def is_dead(self):
    return not self.is_alive

bilbo = Character('比尔博·巴金斯', 100)
bilbo.hp
# 输出 : 100
bilbo.hp = 200
# 输出 : AttributeError: 不能设置属性
# hp 属性是只读的。

bilbo.is_alive
# 输出 : True
bilbo.is_wounded
# 输出 : False
bilbo.is_dead
# 输出 : False

bilbo.take_damage( 50 )

bilbo.hp
# 输出 : 50

bilbo.is_alive
# 输出 : True
bilbo.is_wounded
# 输出 : True
bilbo.is_dead
# 输出 : False

bilbo.take_damage( 50 )
bilbo.hp
# 输出 : 0

bilbo.is_alive
# 输出 : False
bilbo.is_wounded
# 输出 : False
bilbo.is_dead
# 输出 : True

```

## 第38.9节：实例变量的默认值

如果变量包含不可变类型的值（例如字符串），那么像这样赋予默认值是可以的

```

# Make name read only by not providing a set method
@property
def name(self):
    return self.name

def take_damage(self, damage):
    self.hp -= damage
    self.hp = 0 if self.hp < 0 else self.hp

@property
def is_alive(self):
    return self.hp != 0

@property
def is_wounded(self):
    return self.hp < self.max_hp if self.hp > 0 else False

@property
def is_dead(self):
    return not self.is_alive

bilbo = Character('Bilbo Baggins', 100)
bilbo.hp
# out : 100
bilbo.hp = 200
# out : AttributeError: can't set attribute
# hp attribute is read only.

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )

bilbo.hp
# out : 50

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : True
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )
bilbo.hp
# out : 0

bilbo.is_alive
# out : False
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : True

```

## Section 38.9: Default values for instance variables

If the variable contains a value of an immutable type (e.g. a string) then it is okay to assign a default value like this

```

class Rectangle(object):
    def __init__(self, width, height, color='blue'):
        self.width = width
        self.height = height
        self.color = color

    def area(self):
        return self.width * self.height

# 创建该类的一些实例
default_rectangle = Rectangle(2, 3)
print(default_rectangle.color) # 蓝色

red_rectangle = Rectangle(2, 3, '红色')
print(red_rectangle.color) # 红色

```

在构造函数中初始化可变对象（如列表）时需要小心。考虑以下示例：

```

类 Rectangle2D(对象):
    定义 __init__(self, 宽度, 高度, 位置=[0,0], 颜色='蓝色'):
        self.宽度 = 宽度
        self.高度 = 高度
        self.位置 = 位置
        self.颜色 = 颜色

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.位置[0] = 4
r1.位置 # [4, 0]
r2.位置 # [4, 0] r2 的位置也发生了变化

```

这种行为是由于 Python 中默认参数是在函数执行时绑定的，而不是在函数声明时绑定的。为了获得不在实例间共享的默认实例变量，应使用如下构造：

```

类 Rectangle2D(对象):
    def __init__(self, width, height, pos=None, color='blue'):
        self.width = width
        self.height = height
        self.pos = pos or [0, 0] # 默认值是 [0, 0]
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.位置[0] = 4
r1.位置 # [4, 0]
r2.位置 # [0, 0] r2 的位置没有改变

```

另见 [Mutable Default Arguments](#) 和 “Least Astonishment” 以及 [Mutable Default Argument](#)。

## 第38.10节：类变量和实例变量

实例变量对每个实例都是唯一的，而类变量被所有实例共享。

```

class C:
    x = 2 # 类变量

```

```

class Rectangle(object):
    def __init__(self, width, height, color='blue'):
        self.width = width
        self.height = height
        self.color = color

    def area(self):
        return self.width * self.height

# Create some instances of the class
default_rectangle = Rectangle(2, 3)
print(default_rectangle.color) # blue

red_rectangle = Rectangle(2, 3, 'red')
print(red_rectangle.color) # red

```

One needs to be careful when initializing mutable objects such as lists in the constructor. Consider the following example:

```

class Rectangle2D(object):
    def __init__(self, width, height, pos=[0,0], color='blue'):
        self.width = width
        self.height = height
        self.pos = pos
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [4, 0] r2's pos has changed as well

```

This behavior is caused by the fact that in Python default parameters are bound at function execution and not at function declaration. To get a default instance variable that's not shared among instances, one should use a construct like this:

```

class Rectangle2D(object):
    def __init__(self, width, height, pos=None, color='blue'):
        self.width = width
        self.height = height
        self.pos = pos or [0, 0] # default value is [0, 0]
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [0, 0] r2's pos hasn't changed

```

See also [Mutable Default Arguments](#) and “Least Astonishment” and the [Mutable Default Argument](#).

## Section 38.10: Class and instance variables

Instance variables are unique for each instance, while class variables are shared by all instances.

```

class C:
    x = 2 # class variable

```

```
def __init__(self, y):
    self.y = y # 实例变量
```

```
C.x
# 2
C.y
# AttributeError: 类型对象 'C' 没有属性 'y'
```

```
c1 = C(3)
c1.x
# 2
c1.y
# 3
```

```
c2 = C(4)
c2.x
# 2
c2.y
# 4
```

类变量可以通过该类的实例访问，但赋值给类属性会创建一个实例变量，从而遮蔽类变量

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

请注意，从实例中修改类变量可能会导致一些意想不到的后果。

```
类 D:
x = []
def __init__(self, item):
    self.x.append(item) # 注意这不是赋值操作！
```

```
d1 = D(1)
d2 = D(2)
```

```
d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]
```

## 第38.11节：类组合

类组合允许对象之间的显式关系。在此示例中，人们居住在属于国家的城市中。组合允许人们访问其国家中所有居住人口的数量：

```
class Country(object):
    def __init__(self):
        self.cities=[]

    def addCity(self,city):
        self.cities.append(city)
```

```
def __init__(self, y):
    self.y = y # instance variable
```

```
C.x
# 2
C.y
# AttributeError: type object 'C' has no attribute 'y'
```

```
c1 = C(3)
c1.x
# 2
c1.y
# 3
```

```
c2 = C(4)
c2.x
# 2
c2.y
# 4
```

Class variables can be accessed on instances of this class, but assigning to the class attribute will create an instance variable which shadows the class variable

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Note that mutating class variables from instances can lead to some unexpected consequences.

```
class D:
    x = []
    def __init__(self, item):
        self.x.append(item) # note that this is not an assignment!
```

```
d1 = D(1)
d2 = D(2)
```

```
d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]
```

## Section 38.11: Class composition

Class composition allows explicit relations between objects. In this example, people live in cities that belong to countries. Composition allows people to access the number of all people living in their country:

```
class Country(object):
    def __init__(self):
        self.cities=[]

    def addCity(self,city):
        self.cities.append(city)
```

```

class City(object):
    def __init__(self, numPeople):
        self.people = []
        self.numPeople = numPeople

    def addPerson(self, person):
        self.people.append(person)

    def join_country(self, country):
        self.country = country
        country.addCity(self)

        for i in range(self.numPeople):
            person(i).join_city(self)

class Person(object):
    def __init__(self, ID):
        self.ID=ID

    def join_city(self, city):
        self.city = city
        city.addPerson(self)

    def people_in_my_country(self):
        x= sum([len(c.people) for c in self.city.country.cities])
        return x

US=Country()
NYC=City(10).join_country(US)
SF=City(5).join_country(US)

print(US.cities[0].people[0].people_in_my_country())

```

# 15

## 第38.12节：列出所有类成员

可以使用 `dir()` 函数来获取类成员的列表：

`dir(Class)`

例如：

```

>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

```

通常只查找“非魔法”成员。可以使用一个简单的列表推导来列出名称不以 `_` 开头的成员：

```

>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']

```

```

class City(object):
    def __init__(self, numPeople):
        self.people = []
        self.numPeople = numPeople

    def addPerson(self, person):
        self.people.append(person)

    def join_country(self, country):
        self.country = country
        country.addCity(self)

        for i in range(self.numPeople):
            person(i).join_city(self)

class Person(object):
    def __init__(self, ID):
        self.ID=ID

    def join_city(self, city):
        self.city = city
        city.addPerson(self)

    def people_in_my_country(self):
        x= sum([len(c.people) for c in self.city.country.cities])
        return x

US=Country()
NYC=City(10).join_country(US)
SF=City(5).join_country(US)

print(US.cities[0].people[0].people_in_my_country())

```

# 15

## Section 38.12: Listing All Class Members

The `dir()` function can be used to get a list of the members of a class:

`dir(Class)`

For example:

```

>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

```

It is common to look only for “non-magic” members. This can be done using a simple comprehension that lists members with names not starting with `_`:

```

>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']

```

## 注意事项：

类可以定义一个 `__dir__()` 方法。如果该方法存在，调用 `dir()` 会调用 `__dir__()`，否则 Python 会尝试创建类成员的列表。这意味着 `dir` 函数可能会有意想不到的结果。以下是来自 官方 Python 文档 的两段重要引用：

如果对象没有提供 `dir()`，该函数会尽力从对象的 `dict` 属性（如果定义了）和其类型对象中收集信息。生成的列表不一定完整，当对象有自定义的 `getattr()` 时，可能不准确。

注意：因为 `dir()` 主要作为交互式提示符下的便利工具，它尝试提供一组有趣的名称，而不是严格或一致定义的名称集合，其具体行为可能会随着版本变化而改变。例如，当参数是类时，元类属性不会出现在结果列表中。

## 第38.13节：单例类

单例是一种限制类实例化为唯一实例/对象的模式。有关 Python 单例设计模式的更多信息，请参见 [here](#)。

```
class Singleton:
    def __new__(cls):
        try:
            it = cls.__it__
        except AttributeError:
            it = cls.__it__ = object.__new__(cls)
            return it

    def __repr__(self):
        return '<{}>.format(self.__class__.__name__.upper())'

    def __eq__(self, other):
        return other is self
```

另一种方法是装饰你的类。按照这个[answer](#)中的示例，创建一个单例类：

```
class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
    class that should be a singleton.

    The decorated class can define one `__init__` function that
    takes only the `self` argument. Other than that, there are
    no restrictions that apply to the decorated class.
```

被装饰的类可以定义一个只接受 `'self'` 参数的 `'__init__'` 函数。除此之外，对被装饰类没有其他限制。

要获取单例实例，请使用 `'Instance'` 方法。尝试使用 `'__call__'` 会导致抛出 `'TypeError'` 异常。

限制：被装饰的类不能被继承。

...

```
def __init__(self, decorated):
```

## Caveats:

Classes can define a `__dir__()` method. If that method exists calling `dir()` will call `__dir__()`, otherwise Python will try to create a list of members of the class. This means that the `dir` function can have unexpected results. Two quotes of importance from [the official python documentation](#):

If the object does not provide `dir()`, the function tries its best to gather information from the object's `dict` attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom `getattr()`.

**Note:** Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

## Section 38.13: Singleton class

A singleton is a pattern that restricts the instantiation of a class to one instance/object. For more info on python singleton design patterns, see [here](#).

```
class Singleton:
    def __new__(cls):
        try:
            it = cls.__it__
        except AttributeError:
            it = cls.__it__ = object.__new__(cls)
            return it

    def __repr__(self):
        return '<{}>.format(self.__class__.__name__.upper())'

    def __eq__(self, other):
        return other is self
```

另一种方法是装饰你的类。按照这个[answer](#)中的示例，创建一个单例类：

```
class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
    class that should be a singleton.
```

The decorated class can define one `'__init__'` function that takes only the `'self'` argument. Other than that, there are no restrictions that apply to the decorated class.

To get the singleton instance, use the `'Instance'` method. Trying to use `'__call__'` will result in a `'TypeError'` being raised.

Limitations: The decorated class cannot be inherited from.

...

```
def __init__(self, decorated):
```

```

self._decorated = decorated

def Instance(self):
    """
    返回单例实例。首次调用时，会创建被装饰类的新实例并调用其 `__init__` 方法。
    在所有后续调用中，返回已创建的实例。
    """

    try:
        return self._instance
    except AttributeError:
        self._instance = self._decorated()
        return self._instance

def __call__(self):
    raise TypeError('Singletons must be accessed through `Instance()`.')

def __instancecheck__(self, inst):
    return isinstance(inst, self._decorated)

```

要使用，可以使用Instance方法

```

@Singleton
class Single:
    def __init__(self):
        self.name=None
        self.val=0
    def getName(self):
        print(self.name)

x=Single.Instance()
y=Single.Instance()
x.name='I\'m single'
x.getName() # 输出 I'm single
y.getName() # 输出 I'm single

```

## 第38.14节：描述符和点查找

**描述符**是通常作为类属性的对象，并且具有任何 `__get__`、`__set__` 或 `__delete__` 特殊方法。

数据描述符具有任何 `__set__` 或 `__delete__`

这些可以控制实例上的点查找，用于实现函数、`staticmethod`、`classmethod` 和 `property`。点查找（例如类 `Foo` 的实例 `foo` 查找属性 `bar`，即 `foo.bar`）使用以下算法：

- 在类 `Foo` 中查找 `bar`。如果存在且是数据描述符，则使用该数据描述符。  
这就是属性如何能够控制对实例中数据的访问，并且实例无法覆盖这一点。如果一个如果没有数据描述符，则
- `bar` 会在实例的 `__dict__` 中查找。这就是为什么我们可以重写或阻止从一个带点查找的实例。如果实例中存在 `bar`，则使用它。如果不存在，则在类 `Foo` 中查找 `bar`
- 如果它是一个 **Descriptor**，则使用描述符协议。这就是函数的工作方式  
(在此上下文中，未绑定方法)、`classmethod` 和 `staticmethod` 被实现。否则它只是返回该处的对象，或者会出现 `AttributeError`

```

self._decorated = decorated

def Instance(self):
    """
    Returns the singleton instance. Upon its first call, it creates a new instance of the decorated class and calls its `__init__` method. On all subsequent calls, the already created instance is returned.
    """

    try:
        return self._instance
    except AttributeError:
        self._instance = self._decorated()
        return self._instance

def __call__(self):
    raise TypeError('Singletons must be accessed through `Instance()`.')

def __instancecheck__(self, inst):
    return isinstance(inst, self._decorated)

```

To use you can use the `Instance` method

```

@Singleton
class Single:
    def __init__(self):
        self.name=None
        self.val=0
    def getName(self):
        print(self.name)

x=Single.Instance()
y=Single.Instance()
x.name='I\'m single'
x.getName() # outputs I'm single
y.getName() # outputs I'm single

```

## Section 38.14: Descriptors and Dotted Lookups

**Descriptors** are objects that are (usually) attributes of classes and that have any of `__get__`, `__set__`, or `__delete__` special methods.

**Data Descriptors** have any of `__set__`, or `__delete__`

These can control the dotted lookup on an instance, and are used to implement functions, `staticmethod`, `classmethod`, and `property`. A dotted lookup (e.g. instance `foo` of class `Foo` looking up attribute `bar` - i.e. `foo.bar`) uses the following algorithm:

- `bar` is looked up in the class, `Foo`. If it is there and it is a **Data Descriptor**, then the data descriptor is used.  
That's how `property` is able to control access to data in an instance, and instances cannot override this. If a **Data Descriptor** is not there, then
- `bar` is looked up in the instance `__dict__`. This is why we can override or block methods being called from an instance with a dotted lookup. If `bar` exists in the instance, it is used. If not, we then
- look in the class `Foo` for `bar`. If it is a **Descriptor**, then the descriptor protocol is used. This is how functions (in this context, unbound methods), `classmethod`, and `staticmethod` are implemented. Else it simply returns the object there, or there is an `AttributeError`

# 第39章：元类

元类允许你通过替换新类默认使用的type元类，深入修改Python类的行为（包括它们的定义、实例化、访问等方面）。

## 第39.1节：基本元类

当`type`被调用且带有三个参数时，它的行为就像它本身的（元）类，并创建一个新的实例，即生成一个新的类/类型。

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__          # <type 'type'>
Dummy().__class__.__class__ # <type 'type'>
```

可以通过子类化`type`来创建自定义元类。

```
class mytype(type):
    def __init__(cls, name, bases, dict):
        # 调用基类初始化器
        type.__init__(cls, name, bases, dict)

        # 执行自定义初始化...
    cls.__custom_attribute__ = 2
```

现在，我们有了一个新的自定义`mytype`元类，可以像使用`type`一样用它来创建类。

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__          # <class '__main__.mytype'>
MyDummy().__class__.__class__ # <class '__main__.mytype'>
MyDummy.__custom_attribute__ # 2
```

当我们使用`class`关键字创建新类时，元类默认是根据基类选择的。

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

在上述示例中，唯一的基类是`object`，因此我们的元类将是`object`的类型，即`type`。可以覆盖默认设置，但这取决于我们使用的是Python 2还是Python 3：

Python 2.x 版本 ≤ 2.7

可以使用一个特殊的类级属性`__metaclass__`来指定元类。

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy) # <class '__main__.mytype'>

Python 3.x 版本 ≥ 3.0
```

可以使用一个特殊的`metaclass`关键字参数来指定元类。

```
class MyDummy(metaclass=mytype):
```

# Chapter 39: Metaclasses

Metaclasses allow you to deeply modify the behaviour of Python classes (in terms of how they're defined, instantiated, accessed, and more) by replacing the `type` metaclass that new classes use by default.

## Section 39.1: Basic Metaclasses

When `type` is called with three arguments it behaves as the (meta)class it is, and creates a new instance, ie. it produces a new class/type.

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__          # <type 'type'>
Dummy().__class__.__class__ # <type 'type'>
```

It is possible to subclass `type` to create a custom metaclass.

```
class mytype(type):
    def __init__(cls, name, bases, dict):
        # call the base initializer
        type.__init__(cls, name, bases, dict)

        # perform custom initialization...
    cls.__custom_attribute__ = 2
```

Now, we have a new custom `mytype` metaclass which can be used to create classes in the same manner as `type`.

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__          # <class '__main__.mytype'>
MyDummy().__class__.__class__ # <class '__main__.mytype'>
MyDummy.__custom_attribute__ # 2
```

When we create a new class using the `class` keyword the metaclass is by default chosen based on upon the baseclasses.

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

In the above example the only baseclass is `object` so our metaclass will be the type of `object`, which is `type`. It is possible override the default, however it depends on whether we use Python 2 or Python 3:

Python 2.x 版本 ≤ 2.7

A special class-level attribute `__metaclass__` can be used to specify the metaclass.

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy) # <class '__main__.mytype'>

Python 3.x 版本 ≥ 3.0
```

A special `metaclass` keyword argument specify the metaclass.

```
class MyDummy(metaclass=mytype):
```

```
pass  
type(MyDummy) # <class '__main__.mytype'>
```

类声明中的任何关键字参数（除了metaclass）都会传递给元类。因此，`class MyDummy(metaclass=mytype, x=2)`会将 `x=2` 作为关键字参数传递给 `mytype` 的构造函数。

阅读此关于 [Python 元类的深入描述](#)以获取更多细节。

## 第39.2节：使用元类的单例模式

单例是一种限制类实例化为唯一实例/对象的模式。有关 Python 单例设计模式的更多信息，请参见 [here](#)。

```
class SingletonType(type):  
    def __call__(cls, *args, **kwargs):  
        try:  
            return cls.__instance  
        except AttributeError:  
            cls.__instance = super(SingletonType, cls).__call__(*args, **kwargs)  
            return cls.__instance
```

Python 2.x 版本 ≤ 2.7

```
class MySingleton(object):  
    __metaclass__ = SingletonType
```

Python 3.x 版本 ≥ 3.0

```
class MySingleton(metaclass=SingletonType):  
    pass
```

`MySingleton() is MySingleton()` # True, only one instantiation occurs

## 第39.3节：使用元类

### 元类语法

Python 2.x 版本 ≤ 2.7

```
class MyClass(object):  
    __metaclass__ = SomeMetaclass
```

Python 3.x 版本 ≥ 3.0

```
class MyClass(metaclass=SomeMetaclass):  
    pass
```

**Python 2 和 3 的兼容性，使用 six**

```
import six
```

```
class MyClass(six.with_metaclass(SomeMetaclass)):  
    pass
```

## 第39.4节：元类简介

### 什么是元类？

在 Python 中，一切皆为对象：整数、字符串、列表，甚至函数和类本身都是对象。而且每个对象都是某个类的实例。

要检查对象 `x` 的类，可以调用 `type(x)`，因此：

```
pass  
type(MyDummy) # <class '__main__.mytype'>
```

Any keyword arguments (except metaclass) in the class declaration will be passed to the metaclass. Thus `class MyDummy(metaclass=mytype, x=2)` will pass `x=2` as a keyword argument to the `mytype` constructor.

Read this [in-depth description of python meta-classes](#) for more details.

## Section 39.2: Singletons using metaclasses

A singleton is a pattern that restricts the instantiation of a class to one instance/object. For more info on python singleton design patterns, see [here](#).

```
class SingletonType(type):  
    def __call__(cls, *args, **kwargs):  
        try:  
            return cls.__instance  
        except AttributeError:  
            cls.__instance = super(SingletonType, cls).__call__(*args, **kwargs)  
            return cls.__instance
```

Python 2.x Version ≤ 2.7

```
class MySingleton(object):  
    __metaclass__ = SingletonType
```

Python 3.x Version ≥ 3.0

```
class MySingleton(metaclass=SingletonType):  
    pass
```

`MySingleton() is MySingleton()` # True, only one instantiation occurs

## Section 39.3: Using a metaclass

### Metaclass syntax

Python 2.x Version ≤ 2.7

```
class MyClass(object):  
    __metaclass__ = SomeMetaclass
```

Python 3.x Version ≥ 3.0

```
class MyClass(metaclass=SomeMetaclass):  
    pass
```

### Python 2 and 3 compatibility with six

```
import six
```

```
class MyClass(six.with_metaclass(SomeMetaclass)):  
    pass
```

## Section 39.4: Introduction to Metaclasses

### What is a metaclass?

In Python, everything is an object: integers, strings, lists, even functions and classes themselves are objects. And every object is an instance of a class.

To check the class of an object `x`, one can call `type(x)`, so:

```
>>> type(5)
<type '整型'
>>> type(字符串)
<type '类型'
>>> type([1, 2, 3])
<type '列表'>
```

```
>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

Python 中大多数类都是 type 的实例。 type 本身也是一个类。这样的类，其实例也是类，称为元类。

## 最简单的元类

好的，Python 中已经有一个元类：type。我们能创建另一个元类吗？

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

这并没有增加任何功能，但它是一个新的元类，看看 MyClass 现在是 SimplestMetaclass 的实例：

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

## 一个做某事的元类

一个做某事的元类通常会重写 type 的 \_\_new\_\_ 方法，以便在调用创建类的原始 \_\_new\_\_ 之前，修改将要创建的类的一些属性：

```
class AnotherMetaclass(type):
    def __new__(cls, name, parents, dct):
        # cls 是这个类
        # name 是将要创建的类的名称
        # parents 是该类的父类列表
        # dct 是类的属性列表（方法、静态变量）

        # 在这里，所有属性都可以在创建类之前被修改，例如

        dct['x'] = 8 # 现在该类将有一个静态变量 x = 8

        # 返回值是新类。super 会处理这个
        return super(AnotherMetaclass, cls).__new__(cls, name, parents, dct)
```

## 第39.5节：使用元类实现自定义功能

元类中的功能可以被更改，以便每当一个类被构建时，都会向标准输出打印一个字符串，或者抛出一个异常。该元类将打印正在构建的类的名称。

```
class VerboseMetaclass(type):
```

```
>>> type(5)
<type 'int'
>>> type(str)
<type 'type'
>>> type([1, 2, 3])
<type 'list'>
```

```
>>> class C(object):
...     pass
...
>>> type(C)
<type 'type'>
```

Most classes in python are instances of type. type itself is also a class. Such classes whose instances are also classes are called metaclasses.

## The Simplest Metaclass

OK, so there is already one metaclass in Python: type. Can we create another one?

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

That does not add any functionality, but it is a new metaclass, see that MyClass is now an instance of SimplestMetaclass:

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

## A Metaclass which does Something

A metaclass which does something usually overrides type's \_\_new\_\_, to modify some properties of the class to be created, before calling the original \_\_new\_\_ which creates the class:

```
class AnotherMetaclass(type):
    def __new__(cls, name, parents, dct):
        # cls is this class
        # name is the name of the class to be created
        # parents is the list of the class's parent classes
        # dct is the list of class's attributes (methods, static variables)

        # here all of the attributes can be modified before creating the class, e.g.

        dct['x'] = 8 # now the class will have a static variable x = 8

        # return value is the new class. super will take care of that
        return super(AnotherMetaclass, cls).__new__(cls, name, parents, dct)
```

## Section 39.5: Custom functionality with metaclasses

Functionality in metaclasses can be changed so that whenever a class is built, a string is printed to standard output, or an exception is thrown. This metaclass will print the name of the class being built.

```
class VerboseMetaclass(type):
```

```
def __new__(cls, class_name, class_parents, class_dict):
    print("Creating class ", class_name)
new_class = super().__new__(cls, class_name, class_parents, class_dict)
    return new_class
```

你可以这样使用元类：

```
class Spam(metaclass=VerboseMetaclass):
    def eggs(self):
        print("[insert example string here]")
s = Spam()
s.eggs()
```

标准输出将是：

```
创建 类 Spam
[插入示例 字符串 这里]
```

## 第39.6节：默认元类

你可能听说过Python中的一切都是对象。这是真的，所有对象都有一个类：

```
>>> type(1)
int
```

字面量1是int的一个实例。让我们声明一个类：

```
>>> class Foo(object):
...     pass
...
```

现在让我们实例化它：

```
>>> bar = Foo()
```

bar的类是什么？

```
>>> type(bar)
Foo
```

很好，bar 是 Foo 的一个实例。但 Foo 本身的类是什么？

```
>>> type(Foo)
type
```

好的，Foo 本身是 type 的一个实例。那 type 本身呢？

```
>>> type(type)
type
```

那么什么是元类？目前我们先假装它只是“类的类”的一个花哨名称。要点如下：

- 在 Python 中一切都是对象，所以一切都有类
- 类的类被称为元类
- 默认的元类是 type，而且它是最常见的元类

```
def __new__(cls, class_name, class_parents, class_dict):
    print("Creating class ", class_name)
new_class = super().__new__(cls, class_name, class_parents, class_dict)
    return new_class
```

You can use the metaclass like so:

```
class Spam(metaclass=VerboseMetaclass):
    def eggs(self):
        print("[insert example string here]")
s = Spam()
s.eggs()
```

The standard output will be:

```
Creating class Spam
[insert example string here]
```

## Section 39.6: The default metaclass

You may have heard that everything in Python is an object. It is true, and all objects have a class:

```
>>> type(1)
int
```

The literal 1 is an instance of int. Let's declare a class:

```
>>> class Foo(object):
...     pass
...
```

Now let's instantiate it:

```
>>> bar = Foo()
```

What is the class of bar?

```
>>> type(bar)
Foo
```

Nice, bar is an instance of Foo. But what is the class of Foo itself?

```
>>> type(Foo)
type
```

Ok, Foo itself is an instance of type. How about type itself?

```
>>> type(type)
type
```

So what is a metaclass? For now let's pretend it is just a fancy name for the class of a class. Takeaways:

- Everything is an object in Python, so everything has a class
- The class of a class is called a metaclass
- The default metaclass is type, and by far it is the most common metaclass

但为什么你需要了解元类？嗯，Python 本身相当“可破解”，元类的概念在你做高级操作比如元编程，或者想控制类的初始化方式时非常重要。

But why should you know about metaclasses? Well, Python itself is quite "hackable", and the concept of metaclass is important if you are doing advanced stuff like meta-programming or if you want to control how your classes are initialized.

# 视频：机器学习 、数据科学和使用 Pyth on的深度学习

完整的动手机器学习教程，涵盖数据科学、Tensorflow、人工智能和神经网络



- ✓ 使用Tensorflow和Keras构建人工神经网络
- ✓ 使用深度学习对图像、数据和情感进行分类
- ✓ 使用线性回归、多项式回归和多元回归进行预测
- ✓ 使用Matplotlib和Seaborn进行数据可视化
- ✓ 利用Apache Spark的MLlib实现大规模机器学习
- ✓ 理解强化学习——以及如何构建吃豆人机器人
- ✓ 使用K-Means聚类、支持向量机（SVM）、K近邻（KNN）、决策树、朴素贝叶斯和主成分分析（PCA）对数据进行分类
- ✓ 使用训练/测试和K折交叉验证选择和调优模型
- ✓ 使用基于物品和基于用户的协同过滤构建电影推荐系统

今天观看 →

# VIDEO: Machine Learning, Data Science and Deep Learning with Python

Complete hands-on machine learning tutorial with data science, Tensorflow, artificial intelligence, and neural networks



- ✓ Build artificial neural networks with Tensorflow and Keras
- ✓ Classify images, data, and sentiments using deep learning
- ✓ Make predictions using linear regression, polynomial regression, and multivariate regression
- ✓ Data Visualization with Matplotlib and Seaborn
- ✓ Implement machine learning at massive scale with Apache Spark's MLlib
- ✓ Understand reinforcement learning - and how to build a Pac-Man bot
- ✓ Classify data using K-Means clustering, Support Vector Machines (SVM), KNN, Decision Trees, Naive Bayes, and PCA
- ✓ Use train/test and K-Fold cross validation to choose and tune your models
- ✓ Build a movie recommender system using item-based and user-based collaborative filtering

Watch Today →

# 第40章：字符串格式化

当存储和转换供人查看的数据时，字符串格式化可能变得非常重要。Python 提供了多种字符串格式化方法，本主题将对其进行概述。

## 第40.1节：字符串格式化基础

```
foo = 1  
bar = 'bar'  
baz = 3.14
```

你可以使用`str.format`来格式化输出。括号对会被传入的参数按顺序替换：

```
print('{}', {}, 和 {}'.format(foo, bar, baz))  
# 输出: "1, bar 和 3.14"
```

也可以在括号内指定索引。数字对应传入`str.format`函数的参数索引（从0开始）。

```
print('{0}, {1}, {2}, 和 {3}'.format(foo, bar, baz))  
# 输出: "1, bar, 3.14, 和 bar"  
print('{0}, {1}, {2}, 和 {3}'.format(foo, bar, baz))  
# 输出: 索引起超出范围错误
```

也可以使用命名参数：

```
print("X 值是: {x_val}。Y 值是: {y_val}.".format(x_val=2, y_val=3))  
# 输出: "X 值是: 2。Y 值是: 3。"
```

传入`str.format`时可以引用对象属性：

```
class AssignValue(object):  
    def __init__(self, value):  
        self.value = value  
my_value = AssignValue(6)  
print('我的值是: {0.value}'.format(my_value)) # "0" 是可选的  
# 输出: "我的值是: 6"
```

字典键也可以使用：

```
my_dict = {'key': 6, 'other_key': 7}  
print("我的另一个键是: {0[other_key]}".format(my_dict)) # "0" 是可选的  
# 输出: "我的另一个键是: 7"
```

同样适用于列表和元组的索引：

```
my_list = ['zero', 'one', 'two']  
print("第2个元素是: {0[2]}".format(my_list)) # "0" 是可选的  
# 输出: "第2个元素是: two"
```

注意：除了`str.format`，Python 还提供了取模运算符%——也称为字符串格式化或插值运算符（参见 PEP 3101）——用于格式化字符串。`str.format`是%的继任者

# Chapter 40: String Formatting

When storing and transforming data for humans to see, string formatting can become very important. Python offers a wide variety of string formatting methods which are outlined in this topic.

## Section 40.1: Basics of String Formatting

```
foo = 1  
bar = 'bar'  
baz = 3.14
```

You can use `str.format` to format output. Bracket pairs are replaced with arguments in the order in which the arguments are passed:

```
print('{}', {} 和 {}'.format(foo, bar, baz))  
# Out: "1, bar and 3.14"
```

Indexes can also be specified inside the brackets. The numbers correspond to indexes of the arguments passed to the `str.format` function (0-based).

```
print('{0}, {1}, {2}, 和 {3}'.format(foo, bar, baz))  
# Out: "1, bar, 3.14, and bar"  
print('{0}, {1}, {2}, 和 {3}'.format(foo, bar, baz))  
# Out: index out of range error
```

Named arguments can be also used:

```
print("X value is: {x_val}。Y value is: {y_val}.".format(x_val=2, y_val=3))  
# Out: "X value is: 2. Y value is: 3。"
```

Object attributes can be referenced when passed into `str.format`:

```
class AssignValue(object):  
    def __init__(self, value):  
        self.value = value  
my_value = AssignValue(6)  
print('My value is: {0.value}'.format(my_value)) # "0" is optional  
# Out: "My value is: 6"
```

Dictionary keys can be used as well:

```
my_dict = {'key': 6, 'other_key': 7}  
print("My other key is: {0[other_key]}".format(my_dict)) # "0" is optional  
# Out: "My other key is: 7"
```

Same applies to list and tuple indices:

```
my_list = ['zero', 'one', 'two']  
print("2nd element is: {0[2]}".format(my_list)) # "0" is optional  
# Out: "2nd element is: two"
```

Note: In addition to `str.format`, Python also provides the modulo operator %--also known as the *string formatting* or *interpolation operator* (see [PEP 3101](#))--for formatting strings. `str.format` is a successor of %

它提供了更大的灵活性，例如更容易进行多重替换。

除了参数索引外，你还可以在大括号内包含一个格式说明。这是一个遵循特殊规则的表达式，必须以冒号（:）开头。完整的格式说明请参见文档。格式说明的一个例子是对齐指令 :~^20 (^表示居中对齐，总宽度20，填充字符为~)：

```
'{:~^20}'.format('centered')  
# 输出: '~~~~~centered~~~~~'
```

format允许实现 % 无法实现的行为，例如参数的重复使用：

```
t = (12, 45, 2222, 103, 6)  
print '{0} {2} {1} {2} {3} {2} {4} {2}'.format(*t)  
# Out: 12 2222 45 2222 103 2222 6 2222
```

由于format是一个函数，它可以作为其他函数的参数使用：

```
number_list = [12,45,78]  
print map('数字是 {}'.format, number_list)  
# 输出: ['数字是 12', '数字是 45', '数字是 78']  
  
from datetime import datetime, timedelta  
  
once_upon_a_time = datetime(2010, 7, 1, 12, 0, 0)  
delta = timedelta(days=13, hours=8, minutes=20)  
  
gen = (once_upon_a_time + x * delta for x in xrange(5))  
  
print ''.join(map('{:%Y-%m-%d %H:%M:%S}'.format, gen))#输出: 2010-  
07-01 12:00:00  
# 2010-07-14 20:20:00  
# 2010-07-28 04:40:00  
# 2010-08-10 13:00:00  
# 2010-08-23 21:20:00
```

## 第40.2节：对齐和填充

Python 2.x 版本 ≥ 2.6

format()方法可以用来改变字符串的对齐方式。你需要使用格式表达式形式为:[填充字符][对齐操作符][宽度]，其中对齐操作符是以下之一：

- < 强制字段在宽度内左对齐。
- > 强制字段在width内右对齐。
- ^ 强制字段在width内居中。
- = 强制填充放置在符号后面（仅限数值类型）。

fill\_char (如果省略，默认是空白字符) 是用于填充的字符。

```
'{:~<9s}, World'.format('Hello')  
# 'Hello~~~~~, World'  
  
'{:~>9s}, World'.format('Hello')  
# '~~~~Hello, World'
```

and it offers greater flexibility, for instance by making it easier to carry out multiple substitutions.

In addition to argument indexes, you can also include a *format specification* inside the curly brackets. This is an expression that follows special rules and must be preceded by a colon (:). See the [docs](#) for a full description of format specification. An example of format specification is the alignment directive :~^20 (^ stands for center alignment, total width 20, fill with ~ character):

```
'{:~^20}'.format('centered')  
# Out: '~~~~~centered~~~~~'
```

format allows behaviour not possible with %, for example repetition of arguments:

```
t = (12, 45, 2222, 103, 6)  
print '{0} {2} {1} {2} {3} {2} {4} {2}'.format(*t)  
# Out: 12 2222 45 2222 103 2222 6 2222
```

As format is a function, it can be used as an argument in other functions:

```
number_list = [12,45,78]  
print map('the number is {}'.format, number_list)  
# Out: ['the number is 12', 'the number is 45', 'the number is 78']
```

```
from datetime import datetime, timedelta  
  
once_upon_a_time = datetime(2010, 7, 1, 12, 0, 0)  
delta = timedelta(days=13, hours=8, minutes=20)  
  
gen = (once_upon_a_time + x * delta for x in xrange(5))  
  
print '\n'.join(map('{:%Y-%m-%d %H:%M:%S}'.format, gen))  
#Out: 2010-07-01 12:00:00  
# 2010-07-14 20:20:00  
# 2010-07-28 04:40:00  
# 2010-08-10 13:00:00  
# 2010-08-23 21:20:00
```

## Section 40.2: Alignment and padding

Python 2.x Version ≥ 2.6

The format() method can be used to change the alignment of the string. You have to do it with a format expression of the form :[fill\_char][align\_operator][width] where align\_operator is one of:

- < forces the field to be left-aligned within width.
- > forces the field to be right-aligned within width.
- ^ forces the field to be centered within width.
- = forces the padding to be placed after the sign (numeric types only).

fill\_char (if omitted default is whitespace) is the character used for the padding.

```
'{:~<9s}, World'.format('Hello')  
# 'Hello~~~~~, World'  
  
'{:~>9s}, World'.format('Hello')  
# '~~~~Hello, World'
```

```
'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# '-00123'
```

注意：你可以使用字符串函数ljust()、rjust()、center()、zfill()来实现相同的效果，然而这些函数自2.5版本起已被弃用。

## 第40.3节：格式化字面量 (f-string)

格式化字面量字符串是在PEP 498 (Python3.6及以上版本) 中引入的，允许你在字符串字面量开头加上f，从而有效地对其应用.format，使用当前作用域中的所有变量。

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

这也适用于更高级的格式字符串，包括对齐和点符号表示法。

```
>>> f'{foo:^7s}'
' bar '
```

注意：f"并不像python2中的b"表示bytes或 u"表示unicode那样表示特定类型。格式化会立即应用，结果是一个普通字符串。

格式字符串也可以嵌套：

```
>>> price = 478.23
>>> f"${{f'${{price:0.2f}}':>20s}}"
'*****$478.23'
```

f字符串中的表达式按从左到右的顺序求值。只有当表达式具有副作用时，这一点才可被检测到：

```
>>> def fn(l, incr):
...     result = l[0]
...     l[0] += incr
...     return result
...
>>> lst = [0]
>>> f'{fn(lst,2)} {fn(lst,3)}'
'0 2'
>>> f'{fn(lst,2)} {fn(lst,3)}'
'5 7'
>>> lst
[10]
```

## 第40.4节：浮点数格式化

```
>>> '{0:.0f}'.format(42.12345)
'42'

>>> '{0:.1f}'.format(42.12345)
'42.1'

>>> '{0:.3f}'.format(42.12345)
```

```
'{:~^9s}'.format('Hello')
# '~~Hello~~'

'{:0=6d}'.format(-123)
# '-00123'
```

Note: you could achieve the same results using the string functions ljust(), rjust(), center(), zfill(), however these functions are deprecated since version 2.5.

## Section 40.3: Format literals (f-string)

Literal format strings were introduced in [PEP 498](#) (Python3.6 and upwards), allowing you to prepend f to the beginning of a string literal to effectively apply .format to it with all variables in the current scope.

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

This works with more advanced format strings too, including alignment and dot notation.

```
>>> f'{foo:^7s}'
' bar '
```

**Note:** The f' ' does not denote a particular type like b' ' for [bytes](#) or u' ' for [unicode](#) in python2. The formatting is immediately applied, resulting in a normal string.

The format strings can also be *nested*:

```
>>> price = 478.23
>>> f"${{f'${{price:0.2f}}':>20s}}"
'*****$478.23'
```

The expressions in an f-string are evaluated in left-to-right order. This is detectable only if the expressions have side effects:

```
>>> def fn(l, incr):
...     result = l[0]
...     l[0] += incr
...     return result
...
>>> lst = [0]
>>> f'{fn(lst,2)} {fn(lst,3)}'
'0 2'
>>> f'{fn(lst,2)} {fn(lst,3)}'
'5 7'
>>> lst
[10]
```

## Section 40.4: Float formatting

```
>>> '{0:.0f}'.format(42.12345)
'42'

>>> '{0:.1f}'.format(42.12345)
'42.1'

>>> '{0:.3f}'.format(42.12345)
```

```
'42.123'
```

```
>>> '{0:.5f}'.format(42.12345)
```

```
'42.12345'
```

```
>>> '{0:.7f}'.format(42.12345)
```

```
'42.1234500'
```

其他引用方式同理：

```
>>> '{:.3f}'.format(42.12345)
```

```
'42.123'
```

```
>>> '{answer:.3f}'.format(answer=42.12345)
```

```
'42.123'
```

浮点数也可以格式化为科学计数法或百分比形式：

```
>>> '{0:.3e}'.format(42.12345)
```

```
'4.212e+01'
```

```
>>> '{0:.0%}'.format(42.12345)
```

```
'4212%'
```

你也可以结合{0}和{name}的写法。当你想用一次声明将所有变量四舍五入到预设的小数位数时，这尤其有用：

```
>>> s = 'Hello'
```

```
>>> a, b, c = 1.12345, 2.34567, 34.5678
```

```
>>> digits = 2
```

```
>>> '{0}! {1:.{n}f}, {2:.{n}f}, {3:.{n}f}'.format(s, a, b, c, n=digits)
```

```
'Hello! 1.12, 2.35, 34.57'
```

## 第40.5节：命名占位符

格式字符串可以包含命名占位符，这些占位符通过关键字参数传递给format进行插值。

### 使用字典（Python 2.x）

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
```

```
>>> '{first} {last}'.format(**data)
```

```
'Hodor Hodor!'
```

### 使用字典（Python 3.2及以上）

```
>>> '{first} {last}'.format_map(data)
```

```
'Hodor Hodor!'
```

`str.format_map` 允许使用字典而无需先解包它们。同时，使用的是 `data` 的类（可能是自定义类型），而不是新填充的 `dict`。

### 无字典情况下：

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
```

```
'Hodor Hodor!'
```

```
'42.123'
```

```
>>> '{0:.5f}'.format(42.12345)
```

```
'42.12345'
```

```
>>> '{0:.7f}'.format(42.12345)
```

```
'42.1234500'
```

Same hold for other way of referencing:

```
>>> '{:.3f}'.format(42.12345)
```

```
'42.123'
```

```
>>> '{answer:.3f}'.format(answer=42.12345)
```

```
'42.123'
```

Floating point numbers can also be formatted in [scientific notation](#) or as percentages:

```
>>> '{0:.3e}'.format(42.12345)
```

```
'4.212e+01'
```

```
>>> '{0:.0%}'.format(42.12345)
```

```
'4212%'
```

You can also combine the `{0}` and `{name}` notations. This is especially useful when you want to round all variables to a pre-specified number of decimals *with 1 declaration*:

```
>>> s = 'Hello'
```

```
>>> a, b, c = 1.12345, 2.34567, 34.5678
```

```
>>> digits = 2
```

```
>>> '{0}! {1:.{n}f}, {2:.{n}f}, {3:.{n}f}'.format(s, a, b, c, n=digits)
```

```
'Hello! 1.12, 2.35, 34.57'
```

## Section 40.5: Named placeholders

Format strings may contain named placeholders that are interpolated using keyword arguments to `format`.

### Using a dictionary (Python 2.x)

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
```

```
>>> '{first} {last}'.format(**data)
```

```
'Hodor Hodor!'
```

### Using a dictionary (Python 3.2+)

```
>>> '{first} {last}'.format_map(data)
```

```
'Hodor Hodor!'
```

`str.format_map` allows to use dictionaries without having to unpack them first. Also the class of `data` (which might be a custom type) is used instead of a newly filled `dict`.

### Without a dictionary:

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
```

```
'Hodor Hodor!'
```

## 第40.6节：使用datetime进行字符串格式化

任何类都可以通过`__format__`方法配置自己的字符串格式化语法。Python标准库中一个方便使用此功能的类型是`datetime`类型，可以直接在`str.format`中使用类似`strftime`的格式化代码：

```
>>> from datetime import datetime  
>>> '北美: {dt:%m/%d/%Y}。 ISO: {dt:%Y-%m-%d}'.format(dt=datetime.now())  
'北美: 07/21/2016。 ISO: 2016-07-21.'
```

完整的`datetime`格式化代码列表可以在官方文档中找到。

## 第40.7节：数值格式化

`.format()`方法可以将数字解释为不同格式，例如：

```
>>> '{:c}'.format(65)    # Unicode字符  
'A'  
  
>>> '{:d}'.format(0x0a)  # 十进制  
'10'  
  
>>> '{:n}'.format(0x0a) # 使用当前区域设置的分隔符的十进制  
'10'
```

### 将整数格式化为不同进制（十六进制、八进制、二进制）

```
>>> '{0:x}'.format(10) # 十六进制, 小写  
'a'  
  
>>> '{0:X}'.format(10) # 十六进制, 大写  
'A'  
  
>>> '{:o}'.format(10) # 八进制  
'12'  
  
>>> '{:b}'.format(10) # 二进制  
'1010'  
  
>>> '{0:#b}, {0:#o}, {0:#x}'.format(42) # 带前缀  
'0b101010, 0o52, 0x2a'  
  
>>> '8 位: {0:08b}; 三字节: {0:06x}'.format(42) # 添加零填充  
'8 bit: 00101010; Three bytes: 00002a'
```

使用格式化将 RGB 浮点元组转换为颜色十六进制字符串：

```
>>> r, g, b = (1.0, 0.4, 0.0)  
>>> '#{:02X}{:02X}{:02X}'.format(int(255 * r), int(255 * g), int(255 * b))  
'#FF6600'
```

只能转换整数：

```
>>> '{:x}'.format(42.0)  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
ValueError: Unknown format code 'x' for object of type 'float'
```

## Section 40.6: String formatting with datetime

Any class can configure its own string formatting syntax through the `__format__` method. A type in the standard Python library that makes handy use of this is the `datetime` type, where one can use `strftime`-like formatting codes directly within `str.format`:

```
>>> from datetime import datetime  
>>> 'North America: {dt:%m/%d/%Y}. ISO: {dt:%Y-%m-%d}'.format(dt=datetime.now())  
'North America: 07/21/2016. ISO: 2016-07-21.'
```

A full list of list of `datetime` formatters can be found in the [official documentation](#).

## Section 40.7: Formatting Numerical Values

The `.format()` method can interpret a number in different formats, such as:

```
>>> '{:c}'.format(65)      # Unicode character  
'A'  
  
>>> '{:d}'.format(0x0a)   # base 10  
'10'  
  
>>> '{:n}'.format(0x0a)   # base 10 using current locale for separators  
'10'
```

### Format integers to different bases (hex, oct, binary)

```
>>> '{0:x}'.format(10) # base 16, lowercase - Hexadecimal  
'a'  
  
>>> '{0:X}'.format(10) # base 16, uppercase - Hexadecimal  
'A'  
  
>>> '{:o}'.format(10) # base 8 - Octal  
'12'  
  
>>> '{:b}'.format(10) # base 2 - Binary  
'1010'  
  
>>> '{0:#b}, {0:#o}, {0:#x}'.format(42) # With prefix  
'0b101010, 0o52, 0x2a'  
  
>>> '8 bit: {0:08b}; Three bytes: {0:06x}'.format(42) # Add zero padding  
'8 bit: 00101010; Three bytes: 00002a'
```

Use formatting to convert an RGB float tuple to a color hex string:

```
>>> r, g, b = (1.0, 0.4, 0.0)  
>>> '#{0:02X}{0:02X}{0:02X}'.format(int(255 * r), int(255 * g), int(255 * b))  
'#FF6600'
```

Only integers can be converted:

```
>>> '{:x}'.format(42.0)  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
ValueError: Unknown format code 'x' for object of type 'float'
```

## 第40.8节：嵌套格式化

某些格式可以接受额外参数，例如格式化字符串的宽度或对齐方式：

```
>>> '{.:>10}'.format('foo')
'.....foo'
```

这些参数也可以通过在{}内嵌套更多{}作为format的参数提供：

```
>>> '{.:>{}>'.format('foo', 10)
'.....foo'
'{:{}{}{}{}>'.format('foo', '*', '^', 15)
'*****foo*****'
```

在后一个例子中，格式字符串'{:{}{}{}{}{'被修改为'{:^15}'（即“居中并用\*填充，总长度为15”），然后应用于实际要格式化的字符串'foo'。

这在参数事先未知的情况下非常有用，例如对齐表格数据时：

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
a
bbbbbbb
ccc
```

## 第40.9节：使用Getitem和getattr进行格式化

任何支持\_\_getitem\_\_的数据结构都可以格式化其嵌套结构：

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

对象属性可以使用getattr()访问：

```
class Person(object):
first = 'Zaphod'
last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

## 第40.10节：字符串的填充和截断，组合使用

假设你想在一个3字符宽的列中打印变量。

注意：双写{}和{}用于转义它们。

```
s = """
填充
{{:3}}
:a:3:
截断
```

## Section 40.8: Nested formatting

Some formats can take additional parameters, such as the width of the formatted string, or the alignment:

```
>>> '{.:>10}'.format('foo')
'.....foo'
```

Those can also be provided as parameters to format by nesting more {} inside the {}:

```
>>> '{.:>{}>'.format('foo', 10)
'.....foo'
'{:{}{}{}{}>'.format('foo', '*', '^', 15)
'*****foo*****'
```

In the latter example, the format string '{:{}{}{}{}{' is modified to '{:^15}' (i.e. "center and pad with \* to total length of 15") before applying it to the actual string 'foo' to be formatted that way.

This can be useful in cases when parameters are not known beforehand, for instances when aligning tabular data:

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
a
bbbbbbb
ccc
```

## Section 40.9: Format using Getitem and getattr

Any data structure that supports \_\_getitem\_\_ can have their nested structure formatted:

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

Object attributes can be accessed using getattr():

```
class Person(object):
first = 'Zaphod'
last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

## Section 40.10: Padding and truncating strings, combined

Say you want to print variables in a 3 character column.

Note: doubling {} and {} escapes them.

```
s = """
pad
{{:3}}
:a:3:
truncate
```

```

{{:.3}}      :{e:.3}:
组合
{{:>3.3}}   :{a:>3.3}:
{{:3.3}}     :{a:3.3}:
{{:3.3}}     :{c:3.3}:
{{:3.3}}     :{e:3.3}:
"""

print(s.format(a="1"*1, c="3"*3, e="5"*5))

```

输出：

```

填充
{:3}        :1 :
截断
{:.3}       :555:
组合
{:>3.3}    : 1:
{:3.3}      :1 :
{:3.3}      :333:
{:3.3}      :555:

```

## 第40.11节：类的自定义格式化

注意：

以下内容适用于str.format方法以及format函数。文中两者可互换使用。

对于传递给format函数的每个值，Python都会查找该参数的\_\_format\_\_方法。

因此，您自定义的类可以拥有自己的\_\_format\_\_方法，以决定format函数如何显示和格式化您的类及其属性。

这与\_\_str\_\_方法不同，因为在\_\_format\_\_方法中，您可以考虑格式化语言，包括对齐、字段宽度等，甚至（如果愿意）实现自己的格式说明符和格式语言扩展。[1](#)

```
object.__format__(self, format_spec)
```

例如：

```

# Python 2 中的示例 - 但也可轻松应用于Python 3

class Example(object):
    def __init__(self,a,b,c):
        self.a, self.b, self.c = a,b,c

    def __format__(self, format_spec):
        """ 实现 's' 格式说明符的特殊语义 """
        # 拒绝任何不是 s 的格式说明符
        if format_spec[-1] != 's':
            raise ValueError('{} 格式说明符不适用于此对象',
format_spec[:-1])

```

```

{{:.3}}      :{e:.3}:
combined
{{:>3.3}}   :{a:>3.3}:
{{:3.3}}     :{a:3.3}:
{{:3.3}}     :{c:3.3}:
{{:3.3}}     :{e:3.3}:
"""


```

```
print(s.format(a="1"*1, c="3"*3, e="5"*5))
```

Output:

```

pad
{:3}        :1 :
truncate
{:.3}       :555:
combined
{:>3.3}    : 1:
{:3.3}      :1 :
{:3.3}      :333:
{:3.3}      :555:

```

## Section 40.11: Custom formatting for a class

Note:

Everything below applies to the `str.format` method, as well as the `format` function. In the text below, the two are interchangeable.

For every value which is passed to the `format` function, Python looks for a `__format__` method for that argument. Your own custom class can therefore have their own `__format__` method to determine how the `format` function will display and format your class and its attributes.

This is different than the `__str__` method, as in the `__format__` method you can take into account the formatting language, including alignment, field width etc, and even (if you wish) implement your own format specifiers, and your own formatting language extensions.[1](#)

```
object.__format__(self, format_spec)
```

For example:

```

# Example in Python 2 - but can be easily applied to Python 3

class Example(object):
    def __init__(self,a,b,c):
        self.a, self.b, self.c = a,b,c

    def __format__(self, format_spec):
        """ Implement special semantics for the 's' format specifier """
        # Reject anything that isn't an s
        if format_spec[-1] != 's':
            raise ValueError('{} format specifier not understood for this object',
format_spec[:-1])

```

```

# 本例中的输出格式为 (<a>,<b>,<c>)
raw = "(" + ",".join([str(self.a), str(self.b), str(self.c)]) + ")"
# 通过使用内置的字符串格式来遵守格式语言
# 因为我们知道原始的 format_spec 以 's' 结尾
# 我们可以利用上面构造的字符串参数的 str.format 方法

return "{r:{f}}".format( r=raw, f=format_spec )

inst = Example(1,2,3)
print "{0:>20s}".format( inst )
# 输出 :
#       (1,2,3)
# 注意右对齐和字段宽度20已被应用。

```

注意：

如果你的自定义类没有自定义的 `__format__` 方法，并且类的实例被传递给 `format` 函数，**Python2** 总是使用 `__str__` 方法或 `__repr__` 方法的返回值来决定打印内容（如果两者都不存在，则使用默认的 `repr`），你需要使用 `s` 格式说明符来格式化它。对于 Python3，要将自定义类传递给 `format` 函数，你需要在自定义类中定义 `__format__` 方法。

```

# Output in this example will be (<a>,<b>,<c>)
raw = "(" + ",".join([str(self.a), str(self.b), str(self.c)]) + ")"
# Honor the format language by using the inbuilt string format
# Since we know the original format_spec ends in an 's'
# we can take advantage of the str.format method with a
# string argument we constructed above
return "{r:{f}}".format( r=raw, f=format_spec )

inst = Example(1,2,3)
print "{0:>20s}".format( inst )
# out :
#       (1,2,3)
# Note how the right align and field width of 20 has been honored.

```

Note:

If your custom class does not have a custom `__format__` method and an instance of the class is passed to the `format` function, **Python2** will always use the return value of the `__str__` method or `__repr__` method to determine what to print (and if neither exist then the default `repr` will be used), and you will need to use the `s` format specifier to format this. With **Python3**, to pass your custom class to the `format` function, you will need define `__format__` method on your custom class.

# 第41章：字符串方法

## 第41.1节：改变字符串的大小写

Python 的字符串类型提供了许多作用于字符串大小写的函数。这些包括：

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

对于 Unicode 字符串（Python 3 中的默认类型），这些操作并非一一对应或可逆。大多数这些操作旨在显示用途，而非规范化。

Python 3.x 版本 ≥ 3.3

### str.casefold()

`str.casefold` 创建一个适合大小写不敏感比较的小写字符串。这比 `str.lower` 更激进，可能会修改已经是小写的字符串或导致字符串长度增加，且不用于显示目的。

```
"XΒΣ".casefold()  
# ' XSSΩ'
```

```
"XΒΣ".lower()  
# ' XΒΣ'
```

大小写折叠（casefolding）所进行的转换由 Unicode 联盟在其网站上的 CaseFolding.txt 文件中定义。

### str.upper()

`str.upper` 将字符串中的每个字符转换为其大写等价字符，例如：

```
"这是一个'字符串'.".upper()  
# "这是一个 '字符串'."
```

### str.lower()

`str.lower` 的作用相反；它将字符串中的每个字符转换为对应的小写字母：

```
"This IS a 'string'.".lower()  
# "this is a 'string'."
```

### str.capitalize()

`str.capitalize` 返回字符串的首字母大写版本，即首字符大写，其余字符小写：

```
"this Is A 'String'.".capitalize() # 首字母大写，其余字符小写
```

# Chapter 41: String Methods

## Section 41.1: Changing the capitalization of a string

Python's string type provides many functions that act on the capitalization of a string. These include:

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

With unicode strings (the default in Python 3), these operations are **not** 1:1 mappings or reversible. Most of these operations are intended for display purposes, rather than normalization.

Python 3.x Version ≥ 3.3

### str.casefold()

`str.casefold` creates a lowercase string that is suitable for case insensitive comparisons. This is more aggressive than `str.lower` and may modify strings that are already in lowercase or cause strings to grow in length, and is not intended for display purposes.

```
"XΒΣ".casefold()  
# ' XSSΩ'
```

```
"XΒΣ".lower()  
# ' XΒΣ'
```

The transformations that take place under casefolding are defined by the Unicode Consortium in the CaseFolding.txt file on their website.

### str.upper()

`str.upper` takes every character in a string and converts it to its uppercase equivalent, for example:

```
"This is a 'string'.".upper()  
# "THIS IS A 'STRING'."
```

### str.lower()

`str.lower` does the opposite; it takes every character in a string and converts it to its lowercase equivalent:

```
"This IS a 'string'.".lower()  
# "this is a 'string'."
```

### str.capitalize()

`str.capitalize` returns a capitalized version of the string, that is, it makes the first character have upper case and the rest lower:

```
"this Is A 'String'.".capitalize() # Capitalizes the first character and lowerscases all others
```

```
# "This is a 'string'."
```

### str.title()

`str.title` 返回字符串的标题格式版本，即每个单词开头的字母大写，其余字母小写：

```
"this Is a 'String'" .title()  
# "This Is A 'String'"
```

### str.swapcase()

`str.swapcase` 返回一个新的字符串对象，其中所有小写字母转换为大写，所有大写字母转换为小写：

```
"this iS A STRiNG" .swapcase() #交换每个字符的大小写  
# "THIS Is a strIng"
```

## 作为str类方法的用法

值得注意的是，这些方法既可以在字符串对象上调用（如上所示），也可以作为`str`类的方法调用（显式调用`str.upper`等）

```
str.upper("This is a 'string'")  
# "THIS IS A 'STRING'"
```

当在例如`map`函数中一次对许多字符串应用这些方法之一时，这非常有用。

```
map(str.upper, ["These", "are", "some", "strings"])  
# ['THESE', 'ARE', 'SOME', 'STRINGS']
```

## 第41.2节：str.translate：字符串中字符的翻译

Python 支持在`str`类型上使用`translate`方法，该方法允许你指定翻译表（用于替换）以及在过程中应删除的任何字符。

`str.translate(table[, deletechars])`

#### 参数 描述

`table` 这是一个查找表，定义了从一个字符到另一个字符的映射关系。

`deletechars` 要从字符串中删除的字符列表。

`maketrans`方法（Python 3 中为`str.maketrans`，Python 2 中为`string.maketrans`）允许你生成一个翻译表。

```
>>> translation_table = str.maketrans("aeiou", "12345")  
>>> my_string = "This is a string!"  
>>> translated = my_string.translate(translation_table)  
'Th3s 3s 1 str3ng!'
```

`translate`方法返回一个字符串，该字符串是原始字符串的翻译副本。

如果只需要删除字符，可以将 `table`参数设置为`None`。

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
```

```
# "This is a 'string'."
```

### str.title()

`str.title` returns the title cased version of the string, that is, every letter in the beginning of a word is made upper case and all others are made lower case:

```
"this Is a 'String'" .title()  
# "This Is A 'String'"
```

### str.swapcase()

`str.swapcase` returns a new string object in which all lower case characters are swapped to upper case and all upper case characters to lower:

```
"this iS A STRiNG" .swapcase() #Swaps case of each character  
# "THIS Is a strIng"
```

### Usage as str class methods

It is worth noting that these methods may be called either on string objects (as shown above) or as a class method of the `str` class (with an explicit call to `str.upper`, etc.)

```
str.upper("This is a 'string'")  
# "THIS IS A 'STRING'"
```

This is most useful when applying one of these methods to many strings at once in say, a `map` function.

```
map(str.upper, ["These", "are", "some", "strings"])  
# ['THESE', 'ARE', 'SOME', 'STRINGS']
```

## Section 41.2: str.translate: Translating characters in a string

Python supports a `translate` method on the `str` type which allows you to specify the translation table (used for replacements) as well as any characters which should be deleted in the process.

`str.translate(table[, deletechars])`

#### Parameter Description

`table` It is a lookup table that defines the mapping from one character to another.

`deletechars` A list of characters which are to be removed from the string.

The `maketrans` method (`str.maketrans` in Python 3 and `string.maketrans` in Python 2) allows you to generate a translation table.

```
>>> translation_table = str.maketrans("aeiou", "12345")  
>>> my_string = "This is a string!"  
>>> translated = my_string.translate(translation_table)  
'Th3s 3s 1 str3ng!'
```

The `translate` method returns a string which is a translated copy of the original string.

You can set the `table` argument to `None` if you only need to delete characters.

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
```

```
'ths syntx s vry sf1'
```

## 第41.3节：str.format和f字符串：将值格式化为字符串

Python通过str.format函数（在2.6版本引入）和f字符串（在3.6版本引入）提供字符串插值和格式化功能。

给定以下变量：

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

以下语句均等价

```
"10 1.5 foo ['a', 1, 2] {'a': 1, 2: 'foo'}"
>>> "{} {} {} {} {}".format(i, f, s, l, d)
>>> str.format("{} {} {} {} {}", i, f, s, l, d)
>>> "{0} {1} {2} {3} {4}".format(i, f, s, l, d)
>>> "{0:d} {1:0.1f} {2} {3!r} {4!r}".format(i, f, s, l, d)
>>> "{i:d} {f:0.1f} {s} {l!r} {d!r}".format(i=i, f=f, s=s, l=l, d=d)
>>> f"{i} {f} {s} {l} {d}"
>>> f"{i:d} {f:0.1f} {s} {l!r} {d!r}"
```

作为参考，Python 也支持 C 风格的字符串格式化限定符。下面的示例与上述等价，但由于灵活性、一致的符号和可扩展性的优势，推荐使用str.format版本：

```
%d %.1f %s %r %r" % (i, f, s, l, d)
"%(i)d %(f)0.1f %(s)s %(l)r %(d)r" % dict(i=i, f=f, s=s, l=l, d=d)
```

str.format中用于插值的大括号也可以编号，以减少格式化字符串时的重复。例如，以下语句等价：

```
"我来自澳大利亚。我喜欢澳大利亚的杯子蛋糕！"
>>> "我来自{}。我喜欢{}的杯子蛋糕！".format("澳大利亚", "澳大利亚")
>>> "我来自{0}。我喜欢{0}的杯子蛋糕！".format("澳大利亚")
```

虽然官方的 Python 文档一如既往地详尽，但pyformat.info提供了一套带有详细解释的优秀示例。

此外，{和}字符可以通过使用双括号进行转义：

```
"{'a': 5, 'b': 6}"
```

```
'ths syntx s vry sf1'
```

## Section 41.3: str.format and f-strings: Format values into a string

Python provides string interpolation and formatting functionality through the `str.format` function, introduced in version 2.6 and f-strings introduced in version 3.6.

Given the following variables:

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

The following statements are all equivalent

```
"10 1.5 foo ['a', 1, 2] {'a': 1, 2: 'foo'}"
>>> "{} {} {} {} {}".format(i, f, s, l, d)
>>> str.format("{} {} {} {} {}", i, f, s, l, d)
>>> "{0} {1} {2} {3} {4}".format(i, f, s, l, d)
>>> "{0:d} {1:0.1f} {2} {3!r} {4!r}".format(i, f, s, l, d)
>>> "{i:d} {f:0.1f} {s} {l!r} {d!r}".format(i=i, f=f, s=s, l=l, d=d)
>>> f"{i} {f} {s} {l} {d}"
>>> f"{i:d} {f:0.1f} {s} {l!r} {d!r}"
```

For reference, Python also supports C-style qualifiers for string formatting. The examples below are equivalent to those above, but the `str.format` versions are preferred due to benefits in flexibility, consistency of notation, and extensibility:

```
%d %.1f %s %r %r" % (i, f, s, l, d)
"%(i)d %(f)0.1f %(s)s %(l)r %(d)r" % dict(i=i, f=f, s=s, l=l, d=d)
```

The braces uses for interpolation in `str.format` can also be numbered to reduce duplication when formatting strings. For example, the following are equivalent:

```
"I am from Australia. I love cupcakes from Australia!"
>>> "I am from {}. I love cupcakes from {}!".format("Australia", "Australia")
>>> "I am from {0}. I love cupcakes from {0}!".format("Australia")
```

While the official python documentation is, as usual, thorough enough, [pyformat.info](#) has a great set of examples with detailed explanations.

Additionally, the { and } characters can be escaped by using double brackets:

```
"{'a': 5, 'b': 6}"
```

```
>>> "{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)
>>> f"{{'{}': {5}, '{}': {6}}}"
```

有关更多信息，请参见字符串格式化。`str.format()`是在PEP 3101中提出的，`f-strings`则是在PEP 498中提出的。

## 第41.4节：字符串模块的有用常量

Python的`string`模块提供了用于字符串相关操作的常量。要使用它们，请导入`string`模块：

```
>>> import string
```

`string.ascii_letters`:

`ascii_lowercase`和`ascii_uppercase`的连接：

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.ascii_lowercase`:

包含所有小写ASCII字符：

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

`string.ascii_uppercase`：

包含所有大写ASCII字符：

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.digits`:

包含所有十进制数字字符：

```
>>> string.digits
'0123456789'
```

`string.hexdigits`:

包含所有十六进制数字字符：

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

`string.octaldigits`:

包含所有八进制数字字符：

```
>>> string.octaldigits
'01234567'
```

```
>>> "{{'{}': {}, '{}': {}}}".format("a", 5, "b", 6)
>>> f"{{'{}': {5}, '{}': {6}}}"
```

See String Formatting for additional information. `str.format()` was proposed in [PEP 3101](#) and `f-strings` in [PEP 498](#).

## Section 41.4: String module's useful constants

Python's `string` module provides constants for string related operations. To use them, import the `string` module:

```
>>> import string
```

`string.ascii_letters`:

Concatenation of `ascii_lowercase` and `ascii_uppercase`:

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.ascii_lowercase`:

Contains all lower case ASCII characters:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

`string.ascii_uppercase`:

Contains all upper case ASCII characters:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.digits`:

Contains all decimal digit characters:

```
>>> string.digits
'0123456789'
```

`string.hexdigits`:

Contains all hex digit characters:

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

`string.octaldigits`:

Contains all octal digit characters:

```
>>> string.octaldigits
'01234567'
```

### string.punctuation:

包含所有在C语言环境中被视为标点符号的字符：

```
>>> string.punctuation  
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

### string.whitespace:

包含所有被视为空白的ASCII字符：

```
>>> string.whitespace '\r\x0b\x0d'
```

在脚本模式下，`print(string.whitespace)` 将打印实际字符，使用 `str` 来获取上面返回的字符串。

### string.printable :

包含所有被认为是可打印的字符；是`string.digits`、`string.ascii_letters`、`string.punctuation`和`string.whitespace`的组合。

```
>>> string.printable  
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~\r\x0d'
```

## 第41.5节：从字符串中去除不需要的前导/尾随字符

提供了三种方法，可以去除字符串的前导和尾随字符：`str.strip`、`str.rstrip`和`str.lstrip`。三种方法的签名相同，且都返回一个新的字符串对象，其中不需要的字符已被移除。

### str.strip([chars])

`str.strip`作用于给定字符串，移除（剥离）参数中包含的任何前导或尾随字符 `chars`；如果未提供`chars`或为`None`，默认移除所有空白字符。例如：

```
>>> "一行带有前后空格的文本".strip()  
'一行带有前后空格的文本'
```

如果提供了`chars`，字符串中包含的所有该字符都会被移除，并返回结果。例如：

```
>>> ">>> 一个 Python 提示符".strip('> ') # 去除 '>' 字符和空格字符  
'一个 Python 提示符'
```

### str.rstrip([chars]) 和 str.lstrip([chars])

这些方法的语义和参数与 `str.strip()` 类似，区别在于它们开始处理的方向不同。`str.rstrip()` 从字符串末尾开始，而 `str.lstrip()` 从字符串开头开始。

### string.punctuation:

Contains all characters which are considered punctuation in the C locale:

```
>>> string.punctuation  
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

### string.whitespace:

Contains all ASCII characters considered whitespace:

```
>>> string.whitespace  
'\t\n\r\x0b\x0c'
```

In script mode, `print(string.whitespace)` will print the actual characters, use `str` to get the string returned above.

### string.printable:

Contains all characters which are considered printable; a combination of `string.digits`, `string.ascii_letters`, `string.punctuation`, and `string.whitespace`.

```
>>> string.printable  
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~\t\n\r\x0b\x0c'
```

## Section 41.5: Stripping unwanted leading/trailing characters from a string

Three methods are provided that offer the ability to strip leading and trailing characters from a string: `str.strip`, `str.rstrip` and `str.lstrip`. All three methods have the same signature and all three return a new string object with unwanted characters removed.

### str.strip([chars])

`str.strip` acts on a given string and removes (strips) any leading or trailing characters contained in the argument `chars`; if `chars` is not supplied or is `None`, all white space characters are removed by default. For example:

```
>>> "    a line with leading and trailing space    ".strip()  
'a line with leading and trailing space'
```

If `chars` is supplied, all characters contained in it are removed from the string, which is returned. For example:

```
>>> ">>> a Python prompt".strip('> ') # strips '>' character and space character  
'a Python prompt'
```

### str.rstrip([chars]) and str.lstrip([chars])

These methods have similar semantics and arguments with `str.strip()`, their difference lies in the direction from which they start. `str.rstrip()` starts from the end of the string while `str.lstrip()` splits from the start of the string.

例如，使用 str.rstrip：

```
>>> "    宽敞的字符串    ".rstrip()  
'宽敞的字符串'
```

而使用 str.lstrip：

```
>>> "    宽敞的字符串    ".lstrip()  
'宽敞的字符串'
```

## 第41.6节：字符串反转

字符串可以使用内置的 reversed() 函数反转，该函数接受一个字符串并返回一个逆序的迭代器。

```
>>> reversed('hello')  
<reversed object at 0x0000000000000000>  
>>> [char for char in reversed('hello')]  
['o', 'l', 'l', 'e', 'h']
```

reversed() 可以用 ".join()" 包裹，将迭代器转换成字符串。

```
>>> ''.join(reversed('hello'))  
'olleh'
```

虽然使用 reversed() 对不熟悉Python的用户来说可能更易读，但使用步长为 -1 的扩展切片更快且更简洁。这里，尝试将其实现为函数：

```
>>> def reversed_string(main_string):  
...     return main_string[::-1]  
  
>>> reversed_string('hello')  
'olleh'
```

## 第41.7节：基于分隔符将字符串拆分成字符串列表

`str.split(sep=None, maxsplit=-1)`

`str.split` 接受一个字符串并返回原字符串的子字符串列表。其行为取决于是否提供了 `sep` 参数。

如果未提供 `sep`，或其值为 `None`，则在任何空白处进行分割。但会忽略开头和结尾的空白字符，且多个连续的空白字符被视为一个空白字符：

```
>>> "This is a sentence.".split()  
['This', 'is', 'a', 'sentence.'][  
  
>>> " This is a sentence. ".split()  
['This', 'is', 'a', 'sentence.'][  
  
>>> ".split()  
[]
```

For example, using `str.rstrip()`:

```
>>> "    spacious string    ".rstrip()  
'spacious string'
```

While, using `str.lstrip()`:

```
>>> "    spacious string    ".lstrip()  
'spacious string'
```

## Section 41.6: Reversing a string

A string can be reversed using the built-in `reversed()` function, which takes a string and returns an iterator in reverse order.

```
>>> reversed('hello')  
<reversed object at 0x0000000000000000>  
>>> [char for char in reversed('hello')]  
['o', 'l', 'l', 'e', 'h']
```

`reversed()` can be wrapped in a call to `'' .join()` to make a string from the iterator.

```
>>> ''.join(reversed('hello'))  
'olleh'
```

While using `reversed()` might be more readable to uninitiated Python users, using extended slicing with a step of -1 is faster and more concise. Here, try to implement it as a function:

```
>>> def reversed_string(main_string):  
...     return main_string[::-1]  
  
>>> reversed_string('hello')  
'olleh'
```

## Section 41.7: Split a string based on a delimiter into a list of strings

`str.split(sep=None, maxsplit=-1)`

`str.split` takes a string and returns a list of substrings of the original string. The behavior differs depending on whether the `sep` argument is provided or omitted.

If `sep` isn't provided, or is `None`, then the splitting takes place wherever there is whitespace. However, leading and trailing whitespace is ignored, and multiple consecutive whitespace characters are treated the same as a single whitespace character:

```
>>> "This is a sentence.".split()  
['This', 'is', 'a', 'sentence.'][  
  
>>> " This is a sentence. ".split()  
['This', 'is', 'a', 'sentence.'][  
  
>>> ".split()  
[]
```

sep参数可用于定义分隔符字符串。原字符串在分隔符出现处被分割，分隔符本身被丢弃。多个连续的分隔符不被视为单个分隔符，而是会产生空字符串。

```
>>> "This is a sentence.".split(' ')
['This', 'is', 'a', 'sentence.']

>>> "Earth,Stars,Sun,Moon".split(',')
['Earth', 'Stars', 'Sun', 'Moon']

>>> " This is    a sentence. ".split(' ')
['', 'This', 'is', '', '', ' ', 'a', 'sentence.', '', '']

>>> "This is a sentence.".split('e')
['This is a s', 'nt', 'nc', '']

>>> "This is a sentence.".split('en')
['This is a s', 't', 'ce.']

默认情况下会对分隔符的每个出现位置进行分割，但maxsplit参数限制了分割次数。默认值-1表示无限制：
```

```
>>> "This is a sentence.".split('e', maxsplit=0)
['This is a sentence.']

>>> "This is a sentence.".split('e', maxsplit=1)
['This is a s', 'ntence.']

>>> "这是一个句子。".split('e', maxsplit=2)
["这是一个 s", "nt", "nce."]

>>> "这是一个句子。".split('e', maxsplit=-1)
["这是一个 s", "nt", "nc", ""]
```

#### str.rsplit(sep=None, maxsplit=-1)

str.rsplit ("右分割") 与 str.split ("左分割") 在指定maxsplit时不同。分割从字符串的末尾开始，而不是从开头开始：

```
>>> "这是一个句子。".rsplit('e', maxsplit=1)
["这是一个句子", "."]

>>> "这是一个句子。".rsplit('e', maxsplit=2)
["这是一个句", "nc", "."]
```

注意：Python 指定执行的最大分割次数，而大多数其他编程语言指定创建的最大子字符串数。这在移植或比较代码时可能会引起混淆。

## 第41.8节：将一个子字符串的所有出现替换为另一个子字符串

Python 的str类型也有一个方法，用于在给定字符串中将一个子字符串的出现替换为另一个子字符串。对于更复杂的情况，可以使用re.sub

#### str.replace(old, new[, count]):

The sep parameter can be used to define a delimiter string. The original string is split where the delimiter string occurs, and the delimiter itself is discarded. Multiple consecutive delimiters are *not* treated the same as a single occurrence, but rather cause empty strings to be created.

```
>>> "This is a sentence.".split(' ')
['This', 'is', 'a', 'sentence.']

>>> "Earth,Stars,Sun,Moon".split(',')
['Earth', 'Stars', 'Sun', 'Moon']

>>> " This is    a sentence. ".split(' ')
['', 'This', 'is', '', '', ' ', 'a', 'sentence.', '', '']

>>> "This is a sentence.".split('e')
['This is a s', 'nt', 'nc', '']

>>> "This is a sentence.".split('en')
['This is a s', 't', 'ce.']

The default is to split on every occurrence of the delimiter, however the maxsplit parameter limits the number of splittings that occur. The default value of -1 means no limit:
```

```
>>> "This is a sentence.".split('e', maxsplit=0)
['This is a sentence.']

>>> "This is a sentence.".split('e', maxsplit=1)
['This is a s', 'ntence.']

>>> "This is a sentence.".split('e', maxsplit=2)
['This is a s', 'nt', 'nce.']

>>> "This is a sentence.".split('e', maxsplit=-1)
['This is a s', 'nt', 'nc', '']
```

#### str.rsplit(sep=None, maxsplit=-1)

str.rsplit ("right split") differs from str.split ("left split") when maxsplit is specified. The splitting starts at the end of the string rather than at the beginning:

```
>>> "This is a sentence.".rsplit('e', maxsplit=1)
['This is a sentenc', '.']

>>> "This is a sentence.".rsplit('e', maxsplit=2)
['This is a sent', 'nc', '.']
```

Note: Python specifies the maximum number of splits performed, while most other programming languages specify the maximum number of substrings created. This may create confusion when porting or comparing code.

## Section 41.8: Replace all occurrences of one sub-string with another sub-string

Python's str type also has a method for replacing occurrences of one sub-string with another sub-string in a given string. For more demanding cases, one can use re.sub.

#### str.replace(old, new[, count]):

`str.replace`接受两个参数`old`和`new`，包含要被`new`子字符串替换的`old`子字符串。可选参数`count`指定要进行的替换次数：

例如，为了将以下字符串中的'`foo`'替换为'`spam`'，我们可以调用`str.replace`，参数为`old = 'foo'`和`new = 'spam'`：

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
"Make sure to spam your sentence."
```

如果给定的字符串包含多个匹配`old`参数的例子，所有出现的地方都会被替换为`new`提供的值：

```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
"It can spam multiple examples of spam if you want."
```

当然，除非我们为`count`提供了一个值。在这种情况下，将替换`count`个出现的内容：

```
>>> """It can foo multiple examples of foo if you want,
... or you can limit the foo with the third argument."""
.replace('foo', 'spam', 1)
'It can spam multiple examples of foo if you want, or you can limit the foo with the third
argument.'
```

## 第41.9节：测试字符串的组成内容

Python的`str`类型还具有许多方法，可以用来评估字符串的内容。这些方法包括`str.isalpha`、`str.isdigit`、`str.isalnum`、`str.isspace`。大写字母的检测可以使用`str.isupper`，`str.islower`和`str.istitle`。

### `str.isalpha`

`str.isalpha` 不接受参数，如果给定字符串中的所有字符都是字母，则返回`True`，例如：

```
>>> "Hello World".isalpha() # 包含空格
False
>>> "Hello2World".isalpha() # 包含数字
False
>>> "HelloWorld!".isalpha() # 包含标点符号
False
>>> "HelloWorld".isalpha()
True
```

作为一个边界情况，空字符串在使用`".isalpha()`时会被评估为`False`。

### `str.isupper`, `str.islower`, `str.istitle`

这些方法用于测试给定字符串中的大小写情况。

`str.isupper` 是一个方法，如果给定字符串中的所有字符都是大写，则返回`True`，否则返回`False`。

```
>>> "HeLL0 WORLd".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> """.isupper()
```

`str.replace` takes two arguments `old` and `new` containing the `old` sub-string which is to be replaced by the `new` sub-string. The optional argument `count` specifies the number of replacements to be made:

For example, in order to replace '`foo`' with '`spam`' in the following string, we can call `str.replace` with `old = 'foo'` and `new = 'spam'`:

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
"Make sure to spam your sentence."
```

If the given string contains multiple examples that match the `old` argument, **all** occurrences are replaced with the value supplied in `new`:

```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
"It can spam multiple examples of spam if you want."
```

unless, of course, we supply a value for `count`. In this case `count` occurrences are going to get replaced:

```
>>> """It can foo multiple examples of foo if you want,
... or you can limit the foo with the third argument."""
.replace('foo', 'spam', 1)
'It can spam multiple examples of foo if you want, or you can limit the foo with the third
argument.'
```

## Section 41.9: Testing what a string is composed of

Python's `str` type also features a number of methods that can be used to evaluate the contents of a string. These are `str.isalpha`, `str.isdigit`, `str.isalnum`, `str.isspace`. Capitalization can be tested with `str.isupper`, `str.islower` and `str.istitle`.

### `str.isalpha`

`str.isalpha` takes no arguments and returns `True` if the all characters in a given string are alphabetic, for example:

```
>>> "Hello World".isalpha() # contains a space
False
>>> "Hello2World".isalpha() # contains a number
False
>>> "HelloWorld!".isalpha() # contains punctuation
False
>>> "HelloWorld".isalpha()
True
```

As an edge case, the empty string evaluates to `False` when used with `".isalpha()`.

### `str.isupper`, `str.islower`, `str.istitle`

These methods test the capitalization in a given string.

`str.isupper` is a method that returns `True` if all characters in a given string are uppercase and `False` otherwise.

```
>>> "HeLL0 WORLd".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> """.isupper()
```

False

相反, `str.islower` 是一个方法, 如果给定字符串中的所有字符都是小写, 则返回True, 否则返回False。

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` 如果给定字符串是标题格式 (即每个单词以大写字母开头, 后跟小写字母) 则返回True。

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

#### str.isdecimal, str.isdigit, str.isnumeric

`str.isdecimal` 返回字符串是否为十进制数字序列, 适合表示十进制数字。

`str.isdigit` 包含不适合表示十进制数字形式的数字, 例如上标数字。

`str.isnumeric` 包含任何数字值, 即使不是数字字符, 例如超出0-9范围的值。

	isdecimal	isdigit	isnumeric
12345	True	True	True
?2??5	True	True	True
????????	False	True	True
??	False	False	True
Five	False	False	False

字节串 (Python 3 中的`bytes`, Python 2 中的`str`) 仅支持`isdigit`, 该方法仅检查基本的 ASCII 数字。

与`str.isalpha`类似, 空字符串的结果为False。

#### str.isalnum

这是`str.isalpha`和`str.isnumeric`的组合, 具体来说, 如果给定字符串中的所有字符都是字母数字字符, 即由字母或数字字符组成, 则返回True:

```
>>> "Hello2World".isalnum()
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
```

False

Conversely, `str.islower` is a method that returns True if all characters in a given string are lowercase and False otherwise.

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` returns True if the given string is title cased; that is, every word begins with an uppercase character followed by lowercase characters.

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

#### str.isdecimal, str.isdigit, str.isnumeric

`str.isdecimal` returns whether the string is a sequence of decimal digits, suitable for representing a decimal number.

`str.isdigit` includes digits not in a form suitable for representing a decimal number, such as superscript digits.

`str.isnumeric` includes any number values, even if not digits, such as values outside the range 0-9.

	isdecimal	isdigit	isnumeric
12345	True	True	True
?2??5	True	True	True
????????	False	True	True
??	False	False	True
Five	False	False	False

Bytestrings (`bytes` in Python 3, `str` in Python 2), only support `isdigit`, which only checks for basic ASCII digits.

As with `str.isalpha`, the empty string evaluates to False.

#### str.isalnum

This is a combination of `str.isalpha` and `str.isnumeric`, specifically it evaluates to True if all characters in the given string are alphanumeric, that is, they consist of alphabetic or numeric characters:

```
>>> "Hello2World".isalnum()
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
```

```
>>> "Hello World".isalnum() # 包含空格  
False
```

### str.isspace

如果字符串只包含空白字符，则返回True。

```
>>> "\r".isspace()  
True  
>>> " ".isspace()  
True
```

有时一个字符串看起来“空”，但我们不知道是因为它只包含空白字符还是根本没有任何字符

```
>>> "".isspace()  
False
```

为涵盖这种情况，我们需要额外的测试

```
>>> my_str = ''  
>>> my_str.isspace()  
False  
>>> my_str.isspace() or not my_str  
True
```

但测试字符串是否为空或仅包含空白字符的最简方法是使用strip（不带参数时会移除所有开头和结尾的空白字符）

```
>>> not my_str.strip()  
True
```

## 第41.10节：字符串包含

Python使得检查字符串是否包含给定子串非常直观。只需使用in运算符：

```
>>> "foo" in "foo.baz.bar"  
True
```

注意：测试空字符串总是返回True：

```
>>> "" in "test"  
True
```

## 第41.11节：将字符串列表连接成一个字符串

字符串可以用作分隔符，通过join()方法将字符串列表连接成一个单一的字符串。例如，你可以创建一个字符串，其中列表中的每个元素用空格分隔。

```
>>> " ".join(["once", "upon", "a", "time"])  
"once upon a time"
```

下面的示例用三个连字符分隔字符串元素。

```
>>> "---".join(["once", "upon", "a", "time"])
```

```
>>> "Hello World".isalnum() # contains whitespace  
False
```

### str.isspace

Evaluates to True if the string contains only whitespace characters.

```
>>> "\t\r\n".isspace()  
True  
>>> " ".isspace()  
True
```

Sometimes a string looks “empty” but we don't know whether it's because it contains just whitespace or no character at all

```
>>> "".isspace()  
False
```

To cover this case we need an additional test

```
>>> my_str = ''  
>>> my_str.isspace()  
False  
>>> my_str.isspace() or not my_str  
True
```

But the shortest way to test if a string is empty or just contains whitespace characters is to use strip(with no arguments it removes all leading and trailing whitespace characters)

```
>>> not my_str.strip()  
True
```

## Section 41.10: String Contains

Python makes it extremely intuitive to check if a string contains a given substring. Just use the in operator:

```
>>> "foo" in "foo.baz.bar"  
True
```

Note: testing an empty string will always result in True:

```
>>> "" in "test"  
True
```

## Section 41.11: Join a list of strings into one string

A string can be used as a separator to join a list of strings together into a single string using the join() method. For example you can create a string where each element in a list is separated by a space.

```
>>> " ".join(["once", "upon", "a", "time"])  
"once upon a time"
```

The following example separates the string elements with three hyphens.

```
>>> "---".join(["once", "upon", "a", "time"])
```

## 第41.12节：计算子字符串在字符串中出现的次数

有一种方法可以计算子字符串在另一个字符串中出现的次数，即str.count方法。

```
str.count(sub[, start[, end]])
```

str.count返回一个int，表示子字符串sub在另一个字符串中不重叠出现的次数。可选参数start和end表示搜索的起始和结束位置。默认情况下，start = 0, end = len(str)，表示搜索整个字符串：

```
>>> s = "She sells seashells by the seashore."  
>>> s.count("sh")  
2  
>>> s.count("se")  
3  
>>> s.count("sea")  
2  
>>> s.count("seashells")  
1
```

通过为start和end指定不同的值，我们可以获得更局部化的搜索和计数，例如，如果start等于13，则调用：

```
>>> s.count("sea", start)  
1
```

等价于：

```
>>> t = s[start:]  
>>> t.count("sea")  
1
```

## 第41.13节：不区分大小写的字符串比较

以不区分大小写的方式比较字符串看似简单，但实际上并非如此。本节仅考虑unicode字符串（Python 3中的默认字符串）。注意，Python 2相较于Python 3可能存在细微的不足——后者的unicode处理更为完善。

首先需要注意的是，unicode中的大小写转换并非简单操作。有些文本满足text.lower() != text.upper().lower()，例如"ß"：

```
>>> "ß".lower()  
'ß'  
  
>>> "ß".upper().lower()  
'ss'
```

但假设你想无视大小写地比较 "BUSSE" 和 "Buße"。你可能还想把 "BUSSE" 和 "BUE" 视为相等——那是较新的大写形式。推荐的做法是使用 casefold：

Python 3.x 版本 ≥ 3.3

## Section 41.12: Counting number of times a substring appears in a string

One method is available for counting the number of occurrences of a sub-string in another string, str.count.

```
str.count(sub[, start[, end]])
```

str.count returns an int indicating the number of non-overlapping occurrences of the sub-string sub in another string. The optional arguments start and end indicate the beginning and the end in which the search will take place. By default start = 0 and end = len(str) meaning the whole string will be searched:

```
>>> s = "She sells seashells by the seashore."  
>>> s.count("sh")  
2  
>>> s.count("se")  
3  
>>> s.count("sea")  
2  
>>> s.count("seashells")  
1
```

By specifying a different value for start, end we can get a more localized search and count, for example, if start is equal to 13 the call to:

```
>>> s.count("sea", start)  
1
```

is equivalent to:

```
>>> t = s[start:]  
>>> t.count("sea")  
1
```

## Section 41.13: Case insensitive string comparisons

Comparing string in a case insensitive way seems like something that's trivial, but it's not. This section only considers unicode strings (the default in Python 3). Note that Python 2 may have subtle weaknesses relative to Python 3 - the later's unicode handling is much more complete.

The first thing to note is that case-removing conversions in unicode aren't trivial. There is text for which text.lower() != text.upper().lower(), such as "ß":

```
>>> "ß".lower()  
'ß'  
  
>>> "ß".upper().lower()  
'ss'
```

But let's say you wanted to caselessly compare "BUSSE" and "Buße". You probably also want to compare "BUSSE" and "BU E" equal - that's the newer capital form. The recommended way is to use casefold:

Python 3.x Version ≥ 3.3

```
>>> help(str.casefold)
```

'''

方法描述符帮助：

```
casefold(...)  
    S.casefold() -> str
```

返回适合无视大小写比较的 S 版本。

'''

不要仅仅使用 lower。如果没有 casefold，使用 .upper().lower() 会有所帮助（但效果有限）。

然后你还应该考虑重音符号。如果你的字体渲染器很好，你可能认为 "ê" == "e" ——但事实并非如此：

```
>>> "ê" == "ê"  
False
```

这是因为它们实际上是

```
>>> import unicodedata  
  
>>> [unicodedata.name(char) for char in "ê"]  
['带抑扬符的拉丁小写字母e']  
  
>>> [unicodedata.name(char) for char in "ê"]  
['拉丁小写字母e', '组合抑扬符']
```

处理这个问题最简单的方法是使用unicodedata.normalize。你可能想使用NFKD规范化，但可以自由查看文档。然后可以这样做

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê")  
True
```

最后，这里用函数来表示：

```
import unicodedata  
  
def normalize_caseless(text):  
    return unicodedata.normalize("NFKD", text.casefold())  
  
def caseless_equal(left, right):  
    return normalize_caseless(left) == normalize_caseless(right)
```

## 第41.14节：字符串对齐

Python 提供了用于字符串对齐的函数，使文本填充变得更容易，从而方便对齐各种字符串。

下面是 str.ljust 和 str.rjust 的示例：

```
interstates_lengths = {  
    5: (1381, 2222),  
    19: (63, 102),  
    40: (2555, 4112),  
    93: (189, 305),  
}  
for road, length in interstates_lengths.items():  
    miles, kms = length
```

```
>>> help(str.casefold)
```

'''

Help on method\_descriptor:

```
casefold(...)  
    S.casefold() -> str
```

Return a version of S suitable for caseless comparisons.

'''

Do not just use lower. If casefold is not available, doing .upper().lower() helps (but only somewhat).

Then you should consider accents. If your font renderer is good, you probably think "ê" == "ê" - but it doesn't:

```
>>> "ê" == "ê"  
False
```

This is because they are actually

```
>>> import unicodedata  
  
>>> [unicodedata.name(char) for char in "ê"]  
['LATIN SMALL LETTER E WITH CIRCUMFLEX']  
  
>>> [unicodedata.name(char) for char in "ê"]  
['LATIN SMALL LETTER E', 'COMBINING CIRCUMFLEX ACCENT']
```

The simplest way to deal with this is unicodedata.normalize. You probably want to use **NFKD** normalization, but feel free to check the documentation. Then one does

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê")  
True
```

To finish up, here this is expressed in functions:

```
import unicodedata  
  
def normalize_caseless(text):  
    return unicodedata.normalize("NFKD", text.casefold())  
  
def caseless_equal(left, right):  
    return normalize_caseless(left) == normalize_caseless(right)
```

## Section 41.14: Justify strings

Python provides functions for justifying strings, enabling text padding to make aligning various strings much easier.

Below is an example of str.ljust and str.rjust:

```
interstates_lengths = {  
    5: (1381, 2222),  
    19: (63, 102),  
    40: (2555, 4112),  
    93: (189, 305),  
}  
for road, length in interstates_lengths.items():  
    miles, kms = length
```

```

print('{} -> {} mi. ({} km.)'.format(str(road).rjust(4), str(miles).ljust(4),
str(kms).ljust(4)))
40-> 2555 英里 (4112 公里)
19-> 63 英里 (102 公里)
5-> 1381 英里 (2222 公里)
93-> 189 英里 (305 公里)

```

`ljust` 和 `rjust` 非常相似。两者都有一个 `width` 参数和一个可选的 `fillchar` 参数。由这些函数创建的任何字符串长度至少与传入函数的 `width` 参数一样长。如果字符串已经比 `width` 长，则不会被截断。`fillchar` 参数，默认是空格字符 ' '，必须是单个字符，不能是多字符字符串。

`ljust` 函数用 `fillchar` 填充调用它的字符串的末尾，直到字符串长度达到 `width` 个字符。`rjust` 函数以类似的方式填充字符串的开头。因此，这些函数名称中的 `l` 和 `r` 指的是原始字符串在输出字符串中所处的位置，而不是 `fillchar` 所在的位置。

## 第41.15节：测试字符串的起始和结束字符

为了测试Python中给定字符串的开头和结尾，可以使用方法 `str.startswith()` 和 `str.endswith()`。

### `str.startswith(prefix[, start[, end]])`

顾名思义，`str.startswith` 用于测试给定字符串是否以 `prefix` 中的字符开头。

```

>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
>>> s.startswith("thi")
False

```

可选参数 `start` 和 `end` 指定测试开始和结束的位置。

在下面的例子中，通过指定起始值为2，字符串将从位置2开始搜索：

```

>>> s.startswith("is", 2)
True

```

结果为 `True`，因为 `s[2] == 'i'` 且 `s[3] == 's'`。

你也可以使用 `tuple` 来检查字符串是否以一组字符串中的任意一个开头

```

>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False

```

### `str.endswith(prefix[, start[, end]])`

`str.endswith` 与 `str.startswith` 非常相似，唯一的区别是它搜索的是结尾字符而不是起始字符。例如，要测试一个字符串是否以句号结尾，可以写成：

```

print('{} -> {} mi. ({} km.)'.format(str(road).rjust(4), str(miles).ljust(4),
str(kms).ljust(4)))
40 -> 2555 mi. (4112 km.)
19 -> 63 mi. (102 km.)
5 -> 1381 mi. (2222 km.)
93 -> 189 mi. (305 km.)

```

`ljust` 和 `rjust` 非常相似。两者都有一个 `width` 参数和一个可选的 `fillchar` 参数。由这些函数创建的任何字符串长度至少与传入函数的 `width` 参数一样长。如果字符串已经比 `width` 长，则不会被截断。`fillchar` 参数，默认是空格字符 ' '，必须是单个字符，不能是多字符字符串。

The `ljust` function pads the end of the string it is called on with the `fillchar` until it is `width` characters long. The `rjust` function pads the beginning of the string in a similar fashion. Therefore, the `l` and `r` in the names of these functions refer to the side that the original string, *not the fillchar*, is positioned in the output string.

## Section 41.15: Test the starting and ending characters of a string

In order to test the beginning and ending of a given string in Python, one can use the methods `str.startswith()` and `str.endswith()`.

### `str.startswith(prefix[, start[, end]])`

As its name implies, `str.startswith` is used to test whether a given string starts with the given characters in `prefix`.

```

>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
>>> s.startswith("thi")
False

```

The optional arguments `start` and `end` specify the start and end points from which the testing will start and finish. In the following example, by specifying a `start` value of 2 our string will be searched from position 2 and afterwards:

```

>>> s.startswith("is", 2)
True

```

This yields `True` since `s[2] == 'i'` and `s[3] == 's'`.

You can also use a `tuple` to check if it starts with any of a set of strings

```

>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False

```

### `str.endswith(prefix[, start[, end]])`

`str.endswith` is exactly similar to `str.startswith` with the only difference being that it searches for ending characters and not starting characters. For example, to test if a string ends in a full stop, one could write:

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!')
False
```

与startswith类似，结尾序列也可以使用多个字符：

```
>>> s.endswith('stop.')
True
>>> s.endswith('Stop.')
False
```

你也可以使用tuple来检查字符串是否以一组字符串中的任意一个结尾

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

## 第41.16节：str或bytes数据与unicode字符之间的转换

文件内容和网络消息可能表示编码字符。它们通常需要转换为unicode以便正确显示。

在 Python 2 中，您可能需要将 str 数据转换为 Unicode 字符。默认的 ('', "") 等是 ASCII 字符串，任何超出 ASCII I 范围的值都会显示为转义值。Unicode 字符串是 u" (或 u"" 等)。

Python 2.x 版本 ≥ 2.3

```
# 您从文件、网络或其他数据源获得以 UTF-8 编码的 "© abc"

s = '\xc2\xa9 abc' # s 是字节数组，不是字符字符串
# 不知道原始数据是 UTF-8 编码
# Python 2 中字符串字面量的默认形式
s[0] # 'A' - 无意义的字节 (没有编码等上下文时)
type(s) # str - 即使没有已知编码也不是有用的字符串

u = s.decode('utf-8') # u'© abc'
# 现在我们有了一个 Unicode 字符串，可以作为 UTF-8 读取并正确打印

# 在 Python 2 中，Unicode 字符串字面量需要前缀 u# str.decode
# 将可能包含转义字节的字符串转换为 Unicode 字符串

u[0] # u'©' - Unicode 字符 '版权符号' (U+00A9) '©'
type(u) # unicode

u.encode('utf-8') # '© abc'
# unicode.encode 对非 ASCII 字符产生带转义字节的字符串
```

在 Python 3 中，您可能需要将字节数组（称为“字节字面量”）转换为 Unicode 字符串。默认现在是Unicode字符串，字节字面量现在必须写成b' '、b""等形式。字节字面量在假设some\_val是可能被编码为字节的字符串时，isinstance(some\_val,byte)会返回True。

Python 3.x 版本 ≥ 3.0

```
# 你从文件或网络获得的UTF-8编码的"© abc"
```

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!')
False
```

as with startswith more than one characters can be used as the ending sequence:

```
>>> s.endswith('stop.')
True
>>> s.endswith('Stop.')
False
```

You can also use a tuple to check if it ends with any of a set of strings

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

## Section 41.16: Conversion between str or bytes data and unicode characters

The contents of files and network messages may represent encoded characters. They often need to be converted to unicode for proper display.

In Python 2, you may need to convert str data to Unicode characters. The default ('', "", etc.) is an ASCII string, with any values outside of ASCII range displayed as escaped values. Unicode strings are u" (or u"" , etc.).

Python 2.x Version ≥ 2.3

```
# You get "© abc" encoded in UTF-8 from a file, network, or other data source

s = '\xc2\x9a abc' # s is a byte array, not a string of characters
# Doesn't know the original was UTF-8
# Default form of string literals in Python 2
s[0] # '\xc2' - meaningless byte (without context such as an encoding)
type(s) # str - even though it's not a useful one w/o having a known encoding

u = s.decode('utf-8') # u'\xa9 abc'
# Now we have a Unicode string, which can be read as UTF-8 and printed
properly
# In Python 2, Unicode string literals need a leading u
# str.decode converts a string which may contain escaped bytes to a Unicode
string
u[0] # u'\xa9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u) # unicode

u.encode('utf-8') # '\xc2\x9a abc'
# unicode.encode produces a string with escaped bytes for non-ASCII characters
```

In Python 3 you may need to convert arrays of bytes (referred to as a 'byte literal') to strings of Unicode characters. The default is now a Unicode string, and bytestring literals must now be entered as b' ' , b"" , etc. A byte literal will return True to isinstance(some\_val, byte), assuming some\_val to be a string that might be encoded as bytes.

Python 3.x Version ≥ 3.0

```
# You get from file or network "© abc" encoded in UTF-8
```

```

s = b'\xc2\xa9 abc' # s是字节数组, 不是字符
                     # 在Python 3中, 默认字符串字面量是Unicode; 字节数组字面量需要前缀
b
s[0]               # b'\u2022 - 无意义的字节 (没有编码等上下文时)
type(s)            # bytes - 现在字节数组是显式的, Python可以显示出来。

u = s.decode('utf-8') # 在Unicode终端显示为'© abc'# bytes.decode
                     # 将字节数组转换为字符串 (在Python 3中是Unicode)

u[0]               # '\u00a9' - Unicode字符“版权符号”(U+00A9) '©'
type(u)            # str
                     # Python 3中默认的字符串字面量是UTF-8 Unicode

u.encode('utf-8')  # b'\xc2\x9c abc'
                     # str.encode生成字节数组, 显示ASCII范围内的字节为未转义字符。

```

```

s = b'\xc2\x9c abc' # s is a byte array, not characters
                     # In Python 3, the default string literal is Unicode; byte array literals need a
                     leading b
s[0]               # b'\xc2' - meaningless byte (without context such as an encoding)
type(s)            # bytes - now that byte arrays are explicit, Python can show that.

u = s.decode('utf-8') # '© abc' on a Unicode terminal
                     # bytes.decode converts a byte array to a string (which will, in Python 3, be
                     Unicode)
u[0]               # '\u00a9' - Unicode Character 'COPYRIGHT SIGN' (U+00A9) '©'
type(u)            # str
                     # The default string literal in Python 3 is UTF-8 Unicode

u.encode('utf-8')  # b'\xc2\x9c abc'
                     # str.encode produces a byte array, showing ASCII-range bytes as unescaped
                     characters.

```

# 第42章：在函数中使用循环

在Python中，函数一旦执行到“return”语句就会立即返回。

## 第42.1节：函数中循环内的return语句

在此示例中，函数一旦变量var的值为1就会返回

```
def func(params):
    for value in params:
        print ('获取到的值 {}'.format(value))

        if value == 1:
            # 当值为1时立即从函数返回
            print (">>> 获取到1")
            return

    print ("仍在循环中")

return "找不到 1"

func([5, 3, 1, 2, 8, 9])
```

output

```
获取值 5
仍在循环
获取值 3
仍在循环
获取值 1
>>> 获取到 1
```

# Chapter 42: Using loops within functions

In Python function will be returned as soon as execution hits "return" statement.

## Section 42.1: Return statement inside loop in a function

In this example, function will return as soon as value var has 1

```
def func(params):
    for value in params:
        print ('Got value {}'.format(value))

        if value == 1:
            # Returns from function as soon as value is 1
            print (">>> Got 1")
            return

    print ("Still looping")

return "Couldn't find 1"

func([5, 3, 1, 2, 8, 9])
```

output

```
Got value 5
Still looping
Got value 3
Still looping
Got value 1
>>> Got 1
```

# 第43章：导入模块

## 第43.1节：导入模块

使用import语句：

```
>>> import random  
>>> print(random.randint(1, 10))  
4
```

import模块将导入一个模块，然后允许你使用module.name语法引用其对象——例如值、函数和类。在上面的例子中，导入了random模块，该模块包含randint函数。因此，通过导入random，你可以用randint调用为random.randint。

你可以导入一个模块并给它指定一个不同的名称：

```
>>> import random as rn  
>>> print(rn.randint(1, 10))  
4
```

如果你的 Python 文件main.py和custom.py在同一个文件夹中。你可以这样导入它：

```
import custom
```

也可以从模块中导入一个函数：

```
>>> from math import sin  
>>> sin(1)  
0.8414709848078965
```

要从模块的更深层导入特定函数，可以在import关键字的左侧仅使用点操作符：

import 关键字：

```
from urllib.request import urlopen
```

在 Python 中，我们有两种方式调用顶层函数。一种是import，另一种是从。当存在命名冲突的可能时，我们应该使用import。假设我们有hello.py文件和world.py文件，它们都有同名的函数function。那么import语句将会很好地工作。

```
from hello import function  
从 world 导入 function
```

```
function() # 将调用 world 的函数，而不是 hello 的函数
```

一般来说，import 会为你提供一个命名空间。

```
import hello  
import world  
  
hello.function() # 仅调用 hello 的函数  
world.function() # 仅调用 world 的函数
```

但如果你非常确定，在整个项目中不会有同名函数，可以使用 **from** 语句

# Chapter 43: Importing modules

## Section 43.1: Importing a module

Use the **import** statement:

```
>>> import random  
>>> print(random.randint(1, 10))  
4
```

**import** module will import a module and then allow you to reference its objects -- values, functions and classes, for example -- using the module.name syntax. In the above example, the **random** module is imported, which contains the **randint** function. So by importing **random** you can call **randint** with **random.randint**.

You can import a module and assign it to a different name:

```
>>> import random as rn  
>>> print(rn.randint(1, 10))  
4
```

If your python file **main.py** is in the same folder as **custom.py**. You can import it like this:

```
import custom
```

It is also possible to import a function from a module:

```
>>> from math import sin  
>>> sin(1)  
0.8414709848078965
```

To import specific functions deeper down into a module, the dot operator may be used **only** on the left side of the **import** keyword:

```
from urllib.request import urlopen
```

In python, we have two ways to call function from top level. One is **import** and another is **from**. We should use **import** when we have a possibility of name collision. Suppose we have **hello.py** file and **world.py** files having same function named **function**. Then **import** statement will work good.

```
from hello import function  
from world import function
```

```
function() #world's function will be invoked. Not hello's
```

In general **import** will provide you a namespace.

```
import hello  
import world  
  
hello.function() # exclusively hello's function will be invoked  
world.function() # exclusively world's function will be invoked
```

But if you are sure enough, in your whole project there is no way having same function name you should use **from** statement

同一行可以导入多个模块：

```
>>> # 多个模块
>>> import time, sockets, random
>>> # 多个函数
>>> from math import sin, cos, tan
>>> # 多个常量
>>> from math import pi, e

>>> print(pi)
3.141592653589793
>>> print(cos(45))
0.5253219888177297
>>> print(time.time())
1482807222.7240417
```

上述显示的关键字和语法也可以组合使用：

```
>>> from urllib.request import urlopen as geturl, pathname2url as path2url, getproxies
>>> from math import factorial as fact, gamma, atan as arctan
>>> import random.randint, time, sys

>>> print(time.time())
1482807222.7240417
>>> print(arctan(60))
1.554131203080956
>>> filepath = "/dogs/jumping poodle (december).png"
>>> print(path2url(filepath))
/dogs/jumping%20poodle%20%28december%29.png
```

## 第43.2节：\_\_all\_\_ 特殊变量

模块可以有一个名为`__all__`的特殊变量，用来限制使用`from mymodule import *`时导入的变量。

给定以下模块：

```
# mymodule.py

__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

使用`from mymodule import *`时，只有`imported_by_star`会被导入：

```
>>> from mymodule import *
>>> imported_by_star
42
>>> not_imported_by_star
回溯 (最近一次调用最后) :
文件 "<stdin>"，第 1 行，在 <模块>
NameError: 名称 'not_imported_by_star' 未定义
```

但是，`not_imported_by_star` 可以被显式导入：

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
```

Multiple imports can be made on the same line:

```
>>> # Multiple modules
>>> import time, sockets, random
>>> # Multiple functions
>>> from math import sin, cos, tan
>>> # Multiple constants
>>> from math import pi, e

>>> print(pi)
3.141592653589793
>>> print(cos(45))
0.5253219888177297
>>> print(time.time())
1482807222.7240417
```

The keywords and syntax shown above can also be used in combinations:

```
>>> from urllib.request import urlopen as geturl, pathname2url as path2url, getproxies
>>> from math import factorial as fact, gamma, atan as arctan
>>> import random.randint, time, sys

>>> print(time.time())
1482807222.7240417
>>> print(arctan(60))
1.554131203080956
>>> filepath = "/dogs/jumping poodle (december).png"
>>> print(path2url(filepath))
/dogs/jumping%20poodle%20%28december%29.png
```

## Section 43.2: The \_\_all\_\_ special variable

Modules can have a special variable named `__all__` to restrict what variables are imported when using `from mymodule import *`.

Given the following module:

```
# mymodule.py

__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

Only `imported_by_star` is imported when using `from mymodule import *`:

```
>>> from mymodule import *
>>> imported_by_star
42
>>> not_imported_by_star
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'not_imported_by_star' is not defined
```

However, `not_imported_by_star` can be imported explicitly:

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
```

## 第43.3节：从任意文件系统位置导入模块

如果你想导入一个模块，该模块既不是 Python 标准库 中的内置模块，也不是作为附加包存在，你可以通过将包含该模块的目录路径添加到 `sys.path` 来实现。这在主机上存在多个Python环境时可能很有用。

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

重要的是，你要添加的是包含 `mymodule` 的 目录 路径，而不是模块本身的路径。

## 第43.4节：从模块导入所有名称

```
from module_name import *
```

例如：

```
from math import *
sqrt(2)    # 代替 math.sqrt(2)
ceil(2.7)  # 代替 math.ceil(2.7)
```

这将把 `math` 模块中定义的所有名称导入到全局命名空间，除了以下划线开头的名称（表示编写者认为该名称仅供内部使用）。

**警告：**如果已经定义或导入了同名函数，它将被 覆盖。几乎总是推荐只导入特定名称，`from math import sqrt, ceil` 的方式：

```
def sqrt(num):
    print("我不知道 {} 的平方根是多少。".format(num))

sqrt(4)
# 输出：我不知道 4 的平方根是多少。

from math import *
sqrt(4)
# 输出：2.0
```

星号导入仅允许在模块级别进行。尝试在类或函数定义中执行星号导入会导致 `SyntaxError` 错误。

```
def f():
    from math import *
```

和

```
class A:
    from math import *
```

两者都会失败并显示：

## Section 43.3: Import modules from an arbitrary filesystem location

If you want to import a module that doesn't already exist as a built-in module in the [Python Standard Library](#) nor as a side-package, you can do this by adding the path to the directory where your module is found to `sys.path`. This may be useful where multiple python environments exist on a host.

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

It is important that you append the path to the *directory* in which `mymodule` is found, not the path to the module itself.

## Section 43.4: Importing all names from a module

```
from module_name import *
```

for example:

```
from math import *
sqrt(2)    # instead of math.sqrt(2)
ceil(2.7)  # instead of math.ceil(2.7)
```

This will import all names defined in the `math` module into the global namespace, other than names that begin with an underscore (which indicates that the writer feels that it is for internal use only).

**Warning:** If a function with the same name was already defined or imported, it will be **overwritten**. Almost always importing only specific names `from math import sqrt, ceil` is the **recommended way**:

```
def sqrt(num):
    print("I don't know what's the square root of {}".format(num))

sqrt(4)
# Output: I don't know what's the square root of 4.

from math import *
sqrt(4)
# Output: 2.0
```

Starred imports are only allowed at the module level. Attempts to perform them in class or function definitions result in a `SyntaxError`.

```
def f():
    from math import *
```

and

```
class A:
    from math import *
```

both fail with:

SyntaxError: import \* 仅允许在模块级别使用

## 第43.5节：编程式导入

Python 2.x 版本 ≥ 2.7

要通过函数调用导入模块，请使用importlib模块（Python 2.7及以后版本内置）：

```
import importlib  
random = importlib.import_module("random")
```

importlib.import\_module()函数也可以直接导入包的子模块：

```
collections_abc = importlib.import_module("collections.abc")
```

对于较旧版本的Python，请使用imp模块。

Python 2.x 版本 ≤ 2.7

使用函数imp.find\_module和imp.load\_module来执行编程式导入。

[摘自标准库文档](#)

```
import imp, sys  
def import_module(name):  
    fp, pathname, description = imp.find_module(name)  
    try:  
        return imp.load_module(name, fp, pathname, description)  
    finally:  
        if fp:  
            fp.close()
```

不要使用\_\_import\_\_()来编程式导入模块！涉及 sys.modules、fromlist参数等细节很容易被忽视，而importlib.import\_module()会帮你处理这些。

## 第43.6节：导入的PEP8规则

[一些推荐的PEP8导入风格指南](#)：

1. 进口应分行列出：

```
from math import sqrt, ceil      # 不推荐  
from math import sqrt          # 推荐  
from math import ceil
```

2. 在模块顶部按如下顺序导入：

- 标准库导入
- 相关第三方导入
- 本地应用/库特定导入

3. 应避免使用通配符导入，因为这会导致当前命名空间中的名称混淆。如果你确实使用了从模块导入\*时，代码中某个特定名称是否来自模块可能不清楚。这是

SyntaxError: import \* only allowed at module level

## Section 43.5: Programmatic importing

Python 2.x Version ≥ 2.7

To import a module through a function call, use the `importlib` module (included in Python starting in version 2.7):

```
import importlib  
random = importlib.import_module("random")
```

The `importlib.import_module()` function will also import the submodule of a package directly:

```
collections_abc = importlib.import_module("collections.abc")
```

For older versions of Python, use the `imp` module.

Python 2.x Version ≤ 2.7

Use the functions `imp.find_module` and `imp.load_module` to perform a programmatic import.

Taken from [standard library documentation](#)

```
import imp, sys  
def import_module(name):  
    fp, pathname, description = imp.find_module(name)  
    try:  
        return imp.load_module(name, fp, pathname, description)  
    finally:  
        if fp:  
            fp.close()
```

Do NOT use `__import__()` to programmatically import modules! There are subtle details involving `sys.modules`, the `fromlist` argument, etc. that are easy to overlook which `importlib.import_module()` handles for you.

## Section 43.6: PEP8 rules for Imports

[Some recommended PEP8 style guidelines for imports](#):

1. Imports should be on separate lines:

```
from math import sqrt, ceil      # Not recommended  
from math import sqrt          # Recommended  
from math import ceil
```

2. Order imports as follows at the top of the module:

- Standard library imports
- Related third party imports
- Local application/library specific imports

3. Wildcard imports should be avoided as it leads to confusion in names in the current namespace. If you do `from module import *`, it can be unclear if a specific name in your code comes from `module` or not. This is

如果你有多个from module import \*类型的语句，这种情况尤其如此。

4. 避免使用相对导入；改用显式导入。

## 第43.7节：从模块导入特定名称

您可以只导入指定的名称，而不是导入整个模块：

```
from random import randint # 语法 "from 模块名 import 名称1[, 名称2[, ...]]"
print(randint(1, 10))      # 输出: 5
```

需要从random导入，因为Python解释器必须知道从哪个资源导入函数或类，而import randint指定了函数或类本身。

下面是另一个示例（与上面类似）：

```
from math import pi
print(pi)                  # 输出: 3.14159265359
```

以下示例将引发错误，因为我们没有导入模块：

```
random.randrange(1, 10)    # 仅当之前运行过 "import random" 时有效
```

输出：

```
NameError: name 'random' is not defined
```

Python解释器无法理解你所说的random。需要通过添加import random来声明它：

```
import random
random.randrange(1, 10)
```

## 第43.8节：导入子模块

```
from module.submodule import function
```

这会从module.submodule导入function。

## 第43.9节：重新导入模块

在使用交互式解释器时，你可能想重新加载一个模块。如果你正在编辑一个模块并想导入最新版本，或者你对现有模块的某个元素进行了猴子补丁并想恢复更改，这会很有用。

注意，你不能仅仅通过import模块来恢复：

```
import math
math.pi = 3
print(math.pi)      # 3
import math
print(math.pi)      # 3
```

doubly true if you have multiple from module import \*-type statements.

4. Avoid using relative imports; use explicit imports instead.

## Section 43.7: Importing specific names from a module

Instead of importing the complete module you can import only specified names:

```
from random import randint # Syntax "from MODULENAME import NAME1[, NAME2[, ...]]"
print(randint(1, 10))      # Out: 5
```

from random是必要的，因为python解释器必须知道从哪个资源导入函数或类，并且import randint指定了函数或类本身。

下面是另一个示例（与上面类似）：

```
from math import pi
print(pi)                  # Out: 3.14159265359
```

The following example will raise an error, because we haven't imported a module:

```
random.randrange(1, 10)    # works only if "import random" has been run before
```

Outputs:

```
NameError: name 'random' is not defined
```

The python interpreter does not understand what you mean with random. It needs to be declared by adding import random to the example:

```
import random
random.randrange(1, 10)
```

## Section 43.8: Importing submodules

```
from module.submodule import function
```

This imports function from module.submodule.

## Section 43.9: Re-importing a module

When using the interactive interpreter, you might want to reload a module. This can be useful if you're editing a module and want to import the newest version, or if you've monkey-patched an element of an existing module and want to revert your changes.

Note that you can't just import the module again to revert:

```
import math
math.pi = 3
print(math.pi)      # 3
import math
print(math.pi)      # 3
```

这是因为解释器会注册你导入的每个模块。当你尝试重新导入一个模块时，解释器会在注册表中看到它并且不做任何操作。所以重新导入的困难方法是先从注册表中移除对应项后再使用import：

```
print(math.pi)    # 3
import sys
if 'math' in sys.modules: # "math"模块是否已注册？
    del sys.modules['math'] # 如果是这样，删除它。
import math
print(math.pi)    # 3.141592653589793
```

但有一种更直接且简单的方法。

## Python 2

使用 reload 函数：

```
Python 2.x 版本 ≥ 2.3
import math
math.pi = 3
print(math.pi)    # 3
reload(math)
print(math.pi)    # 3.141592653589793
```

## Python 3

reload 函数已移至 importlib：

```
Python 3.x 版本 ≥ 3.0
import math
math.pi = 3
print(math.pi)    # 3
from importlib import reload
reload(math)
print(math.pi)    # 3.141592653589793
```

## 第43.10节：\_\_import\_\_() 函数

\_\_import\_\_() 函数可用于导入名称仅在运行时才知道的模块

```
如果 user_input == "os":
    os = __import__("os")
# 等同于 import os
```

该函数也可用于指定模块的文件路径

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

This is because the interpreter registers every module you import. And when you try to reimport a module, the interpreter sees it in the register and does nothing. So the hard way to reimport is to use `import` after removing the corresponding item from the register:

```
print(math.pi)    # 3
import sys
if 'math' in sys.modules: # Is the ``math`` module in the register?
    del sys.modules['math'] # If so, remove it.
import math
print(math.pi)    # 3.141592653589793
```

But there is more a straightforward and simple way.

## Python 2

Use the `reload` function:

```
Python 2.x Version ≥ 2.3
import math
math.pi = 3
print(math.pi)    # 3
reload(math)
print(math.pi)    # 3.141592653589793
```

## Python 3

The `reload` function has moved to `importlib`:

```
Python 3.x Version ≥ 3.0
import math
math.pi = 3
print(math.pi)    # 3
from importlib import reload
reload(math)
print(math.pi)    # 3.141592653589793
```

## Section 43.10: \_\_import\_\_() function

The `__import__()` function can be used to import modules where the name is only known at runtime

```
if user_input == "os":
    os = __import__("os")
# equivalent to import os
```

This function can also be used to specify the file path to a module

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

# 第44章：模块与包的区别

## 第44.1节：模块

模块是可以导入的单个 Python 文件。使用模块的方式如下：

module.py

```
def hi():
    print("Hello world!")
```

my\_script.py

```
import module
module.hi()
```

在解释器中

```
>>> from module import hi
>>> hi()
# Hello world!
```

## 第44.2节：包

包由多个Python文件（或模块）组成，甚至可以包含用C或C++编写的库。

它不是单个文件，而是一个完整的文件夹结构，可能看起来像这样：

文件夹 包

- `__init__.py`
- `dog.py`
- `hi.py`

`__init__.py`

```
从 package.dog 导入 woof
从 package.hi 导入 hi
```

dog.py

```
定义 woof():
    打印("WOOF!!!")
```

hi.py

```
def hi():
    print("Hello world!")
```

所有 Python 包必须包含一个 `__init__.py` 文件。当你在脚本中导入一个包 (`import package`) 时，`__init__.py` 脚本将被执行，使你能够访问包中的所有函数。在本例中，它允许你使用 `package.hi` 和 `package.woof` 函数。

# Chapter 44: Difference between Module and Package

## Section 44.1: Modules

A module is a single Python file that can be imported. Using a module looks like this:

module.py

```
def hi():
    print("Hello world!")
```

my\_script.py

```
import module
module.hi()
```

in an interpreter

```
>>> from module import hi
>>> hi()
# Hello world!
```

## Section 44.2: Packages

A package is made up of multiple Python files (or modules), and can even include libraries written in C or C++. Instead of being a single file, it is an entire folder structure which might look like this:

Folder package

- `__init__.py`
- `dog.py`
- `hi.py`

`__init__.py`

```
from package.dog import woof
from package.hi import hi
```

dog.py

```
def woof():
    打印("WOOF!!!")
```

hi.py

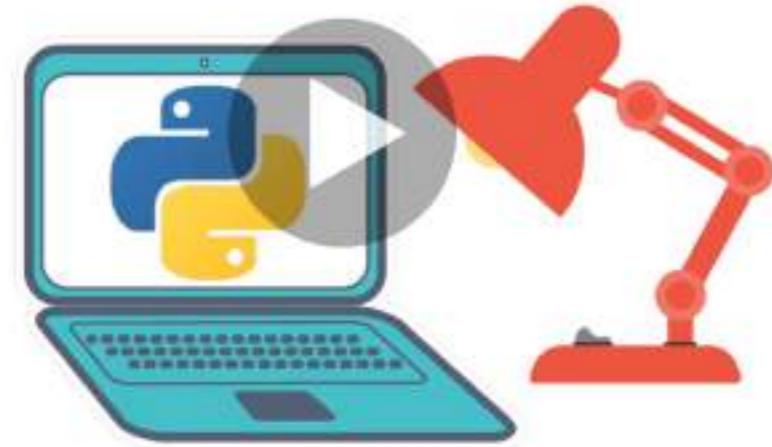
```
def hi():
    print("Hello world!")
```

All Python packages must contain an `__init__.py` file. When you import a package in your script (`import package`), the `__init__.py` script will be run, giving you access to all of the functions in the package. In this case, it allows you to use the `package.hi` and `package.woof` functions.

# 视频：完整的Python 训练营：从零开始 成为Python 3高手

像专业人士一样学习Python！从基础开始，直到  
创建你自己的应用程序和游戏！

## COMPLETE PYTHON 3 BOOTCAMP



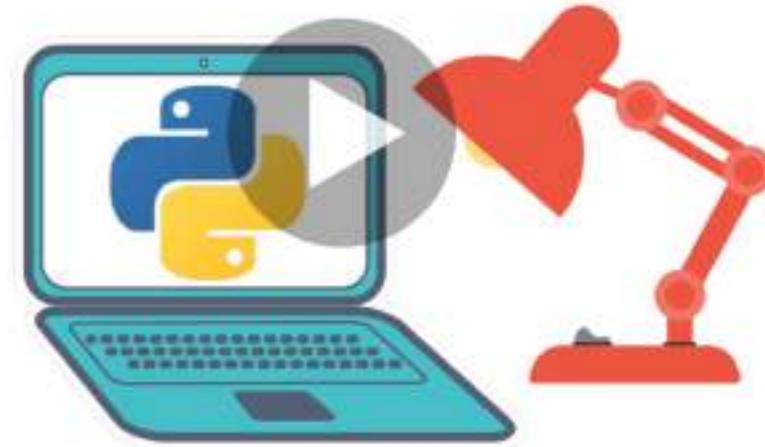
- ✓ 学习专业使用Python，掌握Python 2和Python 3！
- ✓ 用Python创建游戏，比如井字棋和二十一点！
- ✓ 学习高级Python功能，如collections模块和时间戳的使用！
- ✓ 学习使用面向对象编程和类！
- ✓ 理解复杂主题，如装饰器。
- ✓ 理解如何使用Jupyter Notebook并创建.py文件
- ✓ 了解如何在Jupyter Notebook系统中创建图形用户界面（GUI）！
- ✓ 从零开始全面理解Python！

今天观看 →

# VIDEO: Complete Python Bootcamp: Go from zero to hero in Python 3

Learn Python like a Professional! Start from the basics and go all the way to creating your own applications and games!

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ Learn to use Python professionally, learning both Python 2 and Python 3!
- ✓ Create games with Python, like Tic Tac Toe and Blackjack!
- ✓ Learn advanced Python features, like the collections module and how to work with timestamps!
- ✓ Learn to use Object Oriented Programming with classes!
- ✓ Understand complex topics, like decorators.
- ✓ Understand how to use both the Jupyter Notebook and create .py files
- ✓ Get an understanding of how to create GUIs in the Jupyter Notebook system!
- ✓ Build a complete understanding of Python from the ground up!

Watch Today →

# 第45章：数学模块

## 第45.1节：舍入：四舍五入，向下取整，向上取整，截断

除了内置的round函数，math模块还提供了floor、ceil和trunc函数。

```
x = 1.55
y = -1.55

# 四舍五入到最接近的整数
round(x)      # 2
round(y)      # -2

# 第二个参数指定要保留的小数位数 (默认为0)
round(x, 1)   # 1.6
round(y, 1)   # -1.6

# math是一个模块，所以先导入它，然后使用。
import math

# 获取小于x的最大整数
math.floor(x) # 1
math.floor(y) # -2

# 获取大于x的最小整数
math.ceil(x)  # 2
math.ceil(y)  # -1

# 去掉x的小数部分
math.trunc(x) # 1, 等同于正数时的math.floor
math.trunc(y) # -1, 等同于负数时的math.ceil
```

Python 2.x 版本 ≤ 2.7

floor, ceil, trunc 和 round 总是返回 float 类型。

```
round(1.3) # 1.0
```

round 总是将平局情况向远离零的方向舍入。

```
round(0.5) # 1.0
round(1.5) # 2.0
```

Python 3.x 版本 ≥ 3.0

floor, ceil 和 trunc 总是返回 Integral 类型的值，而 round 如果只传入一个参数，则返回 Integral 类型的值。

```
round(1.3)    # 1
round(1.33, 1) # 1.3
```

round 将平局情况舍入到最接近的偶数。这纠正了在进行大量计算时对较大数字的偏差。

```
round(0.5) # 0
round(1.5) # 2
```

警告！

# Chapter 45: Math Module

## Section 45.1: Rounding: round, floor, ceil, trunc

In addition to the built-in `round` function, the `math` module provides the `floor`, `ceil`, and `trunc` functions.

```
x = 1.55
y = -1.55

# round to the nearest integer
round(x)      # 2
round(y)      # -2

# the second argument gives how many decimal places to round to (defaults to 0)
round(x, 1)   # 1.6
round(y, 1)   # -1.6

# math is a module so import it first, then use it.
import math

# get the largest integer less than x
math.floor(x) # 1
math.floor(y) # -2

# get the smallest integer greater than x
math.ceil(x)  # 2
math.ceil(y)  # -1

# drop fractional part of x
math.trunc(x) # 1, equivalent to math.floor for positive numbers
math.trunc(y) # -1, equivalent to math.ceil for negative numbers
```

Python 2.x 版本 ≤ 2.7

floor, ceil, trunc, and round 总是返回 a `float`.

```
round(1.3) # 1.0
```

round 总是 breaks ties away from zero.

```
round(0.5) # 1.0
round(1.5) # 2.0
```

Python 3.x 版本 ≥ 3.0

floor, ceil, and trunc 总是返回 an Integral 值，而 round 返回 an Integral 值 if 叫了带一个参数。

```
round(1.3)    # 1
round(1.33, 1) # 1.3
```

round breaks ties towards the nearest even number. This corrects the bias towards larger numbers when performing a large number of calculations.

```
round(0.5) # 0
round(1.5) # 2
```

Warning!

与任何浮点数表示一样，有些分数无法被精确表示。这可能导致一些意想不到的舍入行为。

```
round(2.675, 2) # 2.67, 而不是2.68 !
```

#### 关于负数的floor、trunc和整数除法的警告

Python（以及C++和Java）对负数的舍入是远离零的。考虑：

```
>>> math.floor(-1.7)  
-2.0  
>>> -5 // 2  
-3
```

## 第45.2节：三角学

### 计算斜边长度

```
math.hypot(2, 4) # 只是 SquareRoot(2**2 + 4**2) 的简写  
# 输出: 4.47213595499958
```

### 角度与弧度的转换

所有math函数都期望输入为弧度，因此你需要将角度转换为弧度：

```
math.radians(45)          # 将45度转换为弧度  
# 输出: 0.7853981633974483
```

所有反三角函数的结果均以弧度返回，因此你可能需要将其转换回角度：

```
math.degrees(math.asin(1))    # 将asin的结果转换为角度  
# 输出: 90.0
```

### 正弦、余弦、正切及其反函数

```
# 正弦和反正弦  
math.sin(math.pi / 2)  
# 输出: 1.0  
math.sin(math.radians(90))    # 90度的正弦值  
# 输出: 1.0
```

```
math.asin(1)
```

```
# 输出: 1.5707963267948966    # "= pi / 2"
```

```
math.asin(1) / math.pi
```

```
# 输出: 0.5
```

```
# 余弦和反余弦：
```

```
math.cos(math.pi / 2)  
# 输出: 6.123233995736766e-17
```

```
# 几乎为零，但不完全是，因为“pi”是有限精度的浮点数！
```

```
math.acos(1)
```

```
# 输出: 0.0
```

```
# 正切和反正切：
```

```
math.tan(math.pi/2)  
# 输出: 1.633123935319537e+16
```

```
# 非常大但不完全是“无穷大”，因为“pi”是有限精度的浮点数
```

Python 3.x 版本 ≥ 3.5

```
math.atan(math.inf)
```

As with any floating-point representation, some fractions *cannot be represented exactly*. This can lead to some unexpected rounding behavior.

```
round(2.675, 2) # 2.67, not 2.68!
```

#### Warning about the floor, trunc, and integer division of negative numbers

Python（and C++ and Java）round away from zero for negative numbers. Consider:

```
>>> math.floor(-1.7)  
-2.0  
>>> -5 // 2  
-3
```

## Section 45.2: Trigonometry

### Calculating the length of the hypotenuse

```
math.hypot(2, 4) # Just a shorthand for SquareRoot(2**2 + 4**2)  
# Out: 4.47213595499958
```

### Converting degrees to/from radians

All math functions expect radians so you need to convert degrees to radians:

```
math.radians(45)          # Convert 45 degrees to radians  
# Out: 0.7853981633974483
```

All results of the inverse trigonometric functions return the result in radians, so you may need to convert it back to degrees:

```
math.degrees(math.asin(1))    # Convert the result of asin to degrees  
# Out: 90.0
```

### Sine, cosine, tangent and inverse functions

```
# Sine and arc sine  
math.sin(math.pi / 2)  
# Out: 1.0  
math.sin(math.radians(90))    # Sine of 90 degrees  
# Out: 1.0
```

```
math.asin(1)  
# Out: 1.5707963267948966    # "= pi / 2"  
math.asin(1) / math.pi  
# Out: 0.5
```

```
# Cosine and arc cosine:  
math.cos(math.pi / 2)  
# Out: 6.123233995736766e-17  
# Almost zero but not exactly because "pi" is a float with limited precision!
```

```
math.acos(1)  
# Out: 0.0
```

```
# Tangent and arc tangent:  
math.tan(math.pi/2)  
# Out: 1.633123935319537e+16  
# Very large but not exactly "Inf" because "pi" is a float with limited precision
```

Python 3.x 版本 ≥ 3.5

```
math.atan(math.inf)
```

```
# 输出: 1.5707963267948966 # 这就是"pi / 2"  
math.atan(float('inf'))  
# 输出: 1.5707963267948966 # 这就是"pi / 2"
```

除了math.atan，还有一个两个参数的math.atan2函数，它可以计算正确的象限并避免除零错误：

```
math.atan2(1, 2)    # 等同于 "math.atan(1/2)"  
# 输出: 0.4636476090008061 # ≈ 26.57 度, 第一象限  
  
math.atan2(-1, -2) # 不等于 "math.atan(-1/-2)" == "math.atan(1/2)"  
# 输出: -2.677945044588987 # ≈ -153.43 度 (或 206.57 度), 第三象限  
  
math.atan2(1, 0)    # math.atan(1/0) 会引发 ZeroDivisionError  
# 输出: 1.5707963267948966 # 这就是"pi / 2"
```

## 双曲正弦、余弦和正切

```
# 双曲正弦函数  
math.sinh(math.pi) # = 11.548739357257746  
math.asinh(1)       # = 0.8813735870195429  
  
# 双曲余弦函数  
math.cosh(math.pi) # = 11.591953275521519  
math.acosh(1)       # = 0.0  
  
# 双曲正切函数  
math.tanh(math.pi) # = 0.99627207622075  
math.atanh(0.5)    # = 0.5493061443340549
```

## 第45.3节：用于更快指数运算的pow函数

使用命令行中的timeit模块：

```
> python -m timeit 'for x in xrange(50000): b = x**3'  
10循环, 3次中最佳: 每次循环51.2毫秒  
> python -m timeit 'from math import pow' 'for x in xrange(50000): b = pow(x,3)'  
100循环, 3次中最佳: 每次循环9.15毫秒
```

内置的\*\*运算符经常很有用，但如果性能至关重要，请使用math.pow。不过需要注意的是，pow即使参数是整数，也会返回浮点数：

```
> from math import pow  
> pow(5,5)  
3125.0
```

## 第45.4节：无穷大和NaN (“不是数字”)

在所有版本的Python中，我们可以如下表示无穷大和NaN (“不是数字”)：

```
pos_inf = float('inf')      # 正无穷大  
neg_inf = float('-inf')     # 负无穷大  
not_a_num = float('nan')    # NaN ("不是数字")
```

在Python 3.5及更高版本中，我们还可以使用定义好的常量math.inf和math.nan：

Python 3.x 版本 ≥ 3.5

```
# Out: 1.5707963267948966 # This is just "pi / 2"  
math.atan(float('inf'))  
# Out: 1.5707963267948966 # This is just "pi / 2"
```

Apart from the math.atan there is also a two-argument math.atan2 function, which computes the correct quadrant and avoids pitfalls of division by zero:

```
math.atan2(1, 2)    # Equivalent to "math.atan(1/2)"  
# Out: 0.4636476090008061 # ≈ 26.57 degrees, 1st quadrant  
  
math.atan2(-1, -2) # Not equal to "math.atan(-1/-2)" == "math.atan(1/2)"  
# Out: -2.677945044588987 # ≈ -153.43 degrees (or 206.57 degrees), 3rd quadrant  
  
math.atan2(1, 0)    # math.atan(1/0) would raise ZeroDivisionError  
# Out: 1.5707963267948966 # This is just "pi / 2"
```

## Hyperbolic sine, cosine and tangent

```
# Hyperbolic sine function  
math.sinh(math.pi) # = 11.548739357257746  
math.asinh(1)       # = 0.8813735870195429  
  
# Hyperbolic cosine function  
math.cosh(math.pi) # = 11.591953275521519  
math.acosh(1)       # = 0.0  
  
# Hyperbolic tangent function  
math.tanh(math.pi) # = 0.99627207622075  
math.atanh(0.5)    # = 0.5493061443340549
```

## Section 45.3: Pow for faster exponentiation

Using the timeit module from the command line:

```
> python -m timeit 'for x in xrange(50000): b = x**3'  
10 loops, best of 3: 51.2 msec per loop  
> python -m timeit 'from math import pow' 'for x in xrange(50000): b = pow(x,3)'  
100 loops, best of 3: 9.15 msec per loop
```

The built-in \*\* operator often comes in handy, but if performance is of the essence, use math.pow. Be sure to note, however, that pow returns floats, even if the arguments are integers:

```
> from math import pow  
> pow(5,5)  
3125.0
```

## Section 45.4: Infinity and NaN ("not a number")

In all versions of Python, we can represent infinity and NaN ("not a number") as follows:

```
pos_inf = float('inf')      # positive infinity  
neg_inf = float('-inf')     # negative infinity  
not_a_num = float('nan')    # NaN ("not a number")
```

In Python 3.5 and higher, we can also use the defined constants math.inf and math.nan:

Python 3.x Version ≥ 3.5

```
pos_inf = math.inf  
neg_inf = -math.inf  
not_a_num = math.nan
```

字符串表示分别显示为inf、-inf和nan：

```
pos_inf, neg_inf, not_a_num  
# 输出: (inf, -inf, nan)
```

我们可以使用isinf方法测试正无穷或负无穷：

```
math.isinf(pos_inf)  
# 输出: True  
  
math.isinf(neg_inf)  
# 输出: True
```

我们也可以通过直接比较来专门测试正无穷或负无穷：

```
pos_inf == float('inf')      # 或者在 Python 3.5+ 中使用 == math.inf  
# 输出: True  
  
neg_inf == float('-inf')    # 或者在 Python 3.5+ 中使用 == -math.inf  
# 输出: True  
  
neg_inf == pos_inf  
# 输出: False
```

Python 3.2 及更高版本也允许检查是否为有限数：

```
Python 3.x 版本 ≥ 3.2  
math.isfinite(pos_inf)  
# 输出: False  
  
math.isfinite(0.0)  
# 输出: True
```

比较运算符对正无穷和负无穷的表现符合预期：

```
import sys  
  
sys.float_info.max  
# 输出: 1.7976931348623157e+308 (此值依赖于系统)  
  
pos_inf > sys.float_info.max  
# 输出: True  
  
neg_inf < -sys.float_info.max  
# 输出: True
```

但是如果算术表达式产生的值大于float类型所能表示的最大值，它将变为无穷大：

```
pos_inf == sys.float_info.max * 1.0000001  
# 输出: True  
  
neg_inf == -sys.float_info.max * 1.0000001
```

```
pos_inf = math.inf  
neg_inf = -math.inf  
not_a_num = math.nan
```

The string representations display as inf and -inf and nan:

```
pos_inf, neg_inf, not_a_num  
# Out: (inf, -inf, nan)
```

We can test for either positive or negative infinity with the isinf method:

```
math.isinf(pos_inf)  
# Out: True  
  
math.isinf(neg_inf)  
# Out: True
```

We can test specifically for positive infinity or for negative infinity by direct comparison:

```
pos_inf == float('inf')      # or == math.inf in Python 3.5+  
# Out: True  
  
neg_inf == float('-inf')    # or == -math.inf in Python 3.5+  
# Out: True  
  
neg_inf == pos_inf  
# Out: False
```

Python 3.2 and higher also allows checking for finiteness:

```
Python 3.x 版本 ≥ 3.2  
math.isfinite(pos_inf)  
# Out: False  
  
math.isfinite(0.0)  
# Out: True
```

Comparison operators work as expected for positive and negative infinity:

```
import sys  
  
sys.float_info.max  
# Out: 1.7976931348623157e+308 (this is system-dependent)  
  
pos_inf > sys.float_info.max  
# Out: True  
  
neg_inf < -sys.float_info.max  
# Out: True
```

But if an arithmetic expression produces a value larger than the maximum that can be represented as a float, it will become infinity:

```
pos_inf == sys.float_info.max * 1.0000001  
# Out: True  
  
neg_inf == -sys.float_info.max * 1.0000001
```

```
# 输出: True
```

然而，除以零并不会得到无穷大（或在适当情况下的负无穷大）结果，而是会引发一个ZeroDivisionError 异常。

```
try:  
x = 1.0 / 0.0  
    print(x)  
except ZeroDivisionError:  
    print("除以零")
```

```
# 输出: 除以零
```

对无穷大的算术运算只会得到无穷大结果，有时会得到NaN：

```
-5.0 * 正无穷 == 负无穷  
# 输出: True
```

```
-5.0 * 负无穷 == 正无穷  
# 输出: True
```

```
正无穷 * 负无穷 == 负无穷  
# 输出: True
```

```
0.0 * 正无穷  
# 输出: nan
```

```
0.0 * 负无穷  
# 输出: nan
```

```
正无穷 / 正无穷  
# 输出: nan
```

NaN 永远不等于任何值，甚至不等于它自身。我们可以用isnan方法来检测它：

```
not_a_num == not_a_num  
# 输出: False
```

```
math.isnan(not_a_num)  
输出: True
```

NaN 总是比较“不等”，但从不小于或大于：

```
not_a_num != 5.0 # 或任何随机值  
# 输出: True
```

```
not_a_num > 5.0 或 not_a_num < 5.0 或 not_a_num == 5.0  
# 输出: False
```

对 NaN 进行算术运算总是得到 NaN。这包括乘以 -1：不存在“负 NaN”。

```
5.0 * not_a_num  
# 输出: nan
```

```
float('nan')  
# 输出: nan
```

Python 3.x 版本 ≥ 3.5

```
-math.nan
```

```
# Out: True
```

However division by zero does not give a result of infinity (or negative infinity where appropriate), rather it raises a ZeroDivisionError exception.

```
try:  
x = 1.0 / 0.0  
    print(x)  
except ZeroDivisionError:  
    print("Division by zero")
```

```
# Out: Division by zero
```

Arithmetic operations on infinity just give infinite results, or sometimes NaN:

```
-5.0 * pos_inf == neg_inf  
# Out: True
```

```
-5.0 * neg_inf == pos_inf  
# Out: True
```

```
pos_inf * neg_inf == neg_inf  
# Out: True
```

```
0.0 * pos_inf  
# Out: nan
```

```
0.0 * neg_inf  
# Out: nan
```

```
pos_inf / pos_inf  
# Out: nan
```

NaN is never equal to anything, not even itself. We can test for it with the isnan method:

```
not_a_num == not_a_num  
# Out: False
```

```
math.isnan(not_a_num)  
Out: True
```

NaN always compares as "not equal", but never less than or greater than:

```
not_a_num != 5.0 # or any random value  
# Out: True
```

```
not_a_num > 5.0 或 not_a_num < 5.0 或 not_a_num == 5.0  
# Out: False
```

Arithmetic operations on NaN always give NaN. This includes multiplication by -1: there is no "negative NaN".

```
5.0 * not_a_num  
# Out: nan
```

```
float('nan')  
# Out: nan
```

Python 3.x 版本 ≥ 3.5

```
-math.nan
```

```
# 输出: nan
```

旧版float的NaN和无穷大与Python 3.5+版本math库常量之间有一个细微的区别：

Python 3.x 版本  $\geq 3.5$

```
math.inf 是 math.inf, math.nan 是 math.nan  
# 输出: (True, True)
```

```
float('inf') 是 float('inf'), float('nan') 是 float('nan')  
# 输出: (False, False)
```

```
# Out: nan
```

There is one subtle difference between the old `float` versions of NaN and infinity and the Python 3.5+ `math` library constants:

Python 3.x Version  $\geq 3.5$

```
math.inf 是 math.inf, math.nan 是 math.nan  
# Out: (True, True)
```

```
float('inf') 是 float('inf'), float('nan') 是 float('nan')  
# Out: (False, False)
```

## 第45.5节：对数

`math.log(x)` 返回 x 的自然对数（以 e 为底）。

```
math.log(math.e) # 1.0  
math.log(1) # 0.0  
math.log(100) # 4.605170185988092
```

`math.log` 在接近1的数字上可能会丢失精度，这是由于浮点数的限制。为了准确计算接近1的对数，使用 `math.log1p`，它计算参数加1的自然对数：

```
math.log(1 + 1e-20) # 0.0  
math.log1p(1e-20) # 1e-20
```

`math.log10` 可用于以10为底的对数：

```
math.log10(10) # 1.0
```

Python 2.x 版本  $\geq 2.3.0$

当使用两个参数时，`math.log(x, base)` 给出 x 在给定 base 下的对数（即  $\log(x) / \log(base)$ ）。

```
math.log(100, 10) # 2.0  
math.log(27, 3) # 3.0  
math.log(1, 10) # 0.0
```

## Section 45.5: Logarithms

`math.log(x)` gives the natural (base e) logarithm of x.

```
math.log(math.e) # 1.0  
math.log(1) # 0.0  
math.log(100) # 4.605170185988092
```

`math.log` can lose precision with numbers close to 1, due to the limitations of floating-point numbers. In order to accurately calculate logs close to 1, use `math.log1p`, which evaluates the natural logarithm of 1 plus the argument:

```
math.log(1 + 1e-20) # 0.0  
math.log1p(1e-20) # 1e-20
```

`math.log10` can be used for logs base 10:

```
math.log10(10) # 1.0
```

Python 2.x Version  $\geq 2.3.0$

When used with two arguments, `math.log(x, base)` gives the logarithm of x in the given base (i.e.  $\log(x) / \log(base)$ ).

```
math.log(100, 10) # 2.0  
math.log(27, 3) # 3.0  
math.log(1, 10) # 0.0
```

## 第45.6节：常量

`math` 模块包含两个常用的数学常数。

- `math.pi` - 数学常数圆周率π
- `math.e` - 数学常数e（自然对数的底数）

```
>>> from math import pi, e  
>>> pi  
3.141592653589793  
>>> e  
2.718281828459045  
>>>
```

Python 3.5及以上版本有表示无穷大和NaN（“不是数字”）的常量。传入字符串给 `float()` 仍然有效。

## Section 45.6: Constants

`math` modules includes two commonly used mathematical constants.

- `math.pi` - The mathematical constant pi
- `math.e` - The mathematical constant e (base of natural logarithm)

```
>>> from math import pi, e  
>>> pi  
3.141592653589793  
>>> e  
2.718281828459045  
>>>
```

Python 3.5 and higher have constants for infinity and NaN ("not a number"). The older syntax of passing a string to `float()` still works.

Python 3.x 版本 ≥ 3.5

```
math.inf == float('inf')
# 输出: True

-math.inf == float('-inf')
# 输出: True

# NaN永远不等于任何值, 甚至不等于它自己
math.nan == float('nan')
# 输出: False
```

## 第45.7节：虚数

Python中的虚数用数字后跟“j”或“J”表示。

```
1j      # 等同于 -1 的平方根。
1j * 1j # = (-1+0j)
```

## 第45.8节：复制符号

在 Python 2.6 及更高版本中, `math.copysign(x, y)` 返回带有 `y` 符号的 `x`。返回值始终是 `float` 类型。

Python 2.x 版本 ≥ 2.6

```
math.copysign(-2, 3)    # 2.0
math.copysign(3, -3)    # -3.0
math.copysign(4, 14.2)  # 4.0
math.copysign(1, -0.0)  # -1.0, 在支持带符号零的平台上
```

## 第45.9节：复数与 cmath 模块

`cmath` 模块类似于 `math` 模块, 但为复平面定义了相应的函数。

首先, 复数是一种数值类型, 是Python语言本身的一部分, 而不是由库类提供的。因此, 对于普通的算术表达式, 我们不需要导入`cmath`模块。

注意我们使用`j` (或`J`) , 而不是`i`。

```
z = 1 + 3j
```

我们必须使用`1j`, 因为`j`会被视为变量名, 而不是数值字面量。

```
1j * 1j
输出: (-1+0j)

1j ** 1j
# 输出: (0.20787957635076193+0j)  # "i 的 i 次方" == math.e ** -(math.pi/2)
```

我们有`real`部分和`imag` (虚数) 部分, 以及复数的`conjugate` (共轭) :

```
# 实部和虚部都是浮点类型
z.real, z.imag
# 输出: (1.0, 3.0)

z.conjugate()
# 输出: (1-3j)  # z.conjugate() == z.real - z.imag * 1j
```

Python 3.x 版本 ≥ 3.5

```
math.inf == float('inf')
# Out: True

-math.inf == float('-inf')
# Out: True

# NaN never compares equal to anything, even itself
math.nan == float('nan')
# Out: False
```

## Section 45.7: Imaginary Numbers

Imaginary numbers in Python are represented by a "j" or "J" trailing the target number.

```
1j      # Equivalent to the square root of -1.
1j * 1j # = (-1+0j)
```

## Section 45.8: Copying signs

In Python 2.6 and higher, `math.copysign(x, y)` returns `x` with the sign of `y`. The returned value is always a `float`.

Python 2.x 版本 ≥ 2.6

```
math.copysign(-2, 3)    # 2.0
math.copysign(3, -3)    # -3.0
math.copysign(4, 14.2)  # 4.0
math.copysign(1, -0.0)  # -1.0, on a platform which supports signed zero
```

## Section 45.9: Complex numbers and the cmath module

The `cmath` module is similar to the `math` module, but defines functions appropriately for the complex plane.

First of all, complex numbers are a numeric type that is part of the Python language itself rather than being provided by a library class. Thus we don't need to `import cmath` for ordinary arithmetic expressions.

Note that we use `j` (or `J`) and not `i`.

```
z = 1 + 3j
```

We must use `1j` since `j` would be the name of a variable rather than a numeric literal.

```
1j * 1j
Out: (-1+0j)

1j ** 1j
# Out: (0.20787957635076193+0j)  # "i to the i" == math.e ** -(math.pi/2)
```

We have the `real` part and the `imag` (imaginary) part, as well as the complex conjugate:

```
# real part and imaginary part are both float type
z.real, z.imag
# Out: (1.0, 3.0)

z.conjugate()
# Out: (1-3j)  # z.conjugate() == z.real - z.imag * 1j
```

内置函数abs和complex也是语言本身的一部分，无需导入：

```
abs(1 + 1j)
# 输出: 1.4142135623730951    # 2的平方根

complex(1)
# 输出: (1+0j)

complex(imag=1)
# 输出: (1j)

complex(1, 1)
# 输出: (1+1j)
```

complex函数可以接受一个字符串，但字符串中不能有空格：

```
complex('1+1j')
# 输出: (1+1j)

complex('1 + 1j')
# 异常: ValueError: complex() 参数是格式错误的字符串
```

但对于大多数函数，我们确实需要模块，例如sqrt：

```
import cmath

cmath.sqrt(-1)
# 输出: 1j
```

自然，sqrt的行为对于复数和实数是不同的。在非复数的math模块中，负数的平方根会引发异常：

```
import math

math.sqrt(-1)
# 异常: ValueError: math 域错误
```

提供了用于在极坐标和直角坐标之间转换的函数：

```
cmath.polar(1 + 1j)
# 输出: (1.4142135623730951, 0.7853981633974483)    # == (sqrt(1 + 1), atan2(1, 1))

abs(1 + 1j), cmath.phase(1 + 1j)
# 输出: (1.4142135623730951, 0.7853981633974483)    # 与之前的计算相同

cmath.rect(math.sqrt(2), math.atan(1))
# 输出: (1.0000000000000002+1.0000000000000002j)
```

复分析的数学领域超出了本示例的范围，但复平面上的许多函数都有“分支切割”，通常沿实轴或虚轴。大多数现代平台支持IEEE 754规定的“带符号零”，这保证了这些函数在分支切割两侧的连续性。以下示例摘自Python文档：

```
cmath.phase(complex(-1.0, 0.0))
# 输出: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
```

The built-in functions `abs` and `complex` are also part of the language itself and don't require any import:

```
abs(1 + 1j)
# Out: 1.4142135623730951    # square root of 2

complex(1)
# Out: (1+0j)

complex(imag=1)
# Out: (1j)

complex(1, 1)
# Out: (1+1j)
```

The `complex` function can take a string, but it can't have spaces:

```
complex('1+1j')
# Out: (1+1j)

complex('1 + 1j')
# Exception: ValueError: complex() arg is a malformed string
```

But for most functions we do need the module, for instance sqrt:

```
import cmath

cmath.sqrt(-1)
# Out: 1j
```

Naturally the behavior of sqrt is different for complex numbers and real numbers. In non-complex `math` the square root of a negative number raises an exception:

```
import math

math.sqrt(-1)
# Exception: ValueError: math domain error
```

Functions are provided to convert to and from polar coordinates:

```
cmath.polar(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # == (sqrt(1 + 1), atan2(1, 1))

abs(1 + 1j), cmath.phase(1 + 1j)
# Out: (1.4142135623730951, 0.7853981633974483)    # same as previous calculation

cmath.rect(math.sqrt(2), math.atan(1))
# Out: (1.0000000000000002+1.0000000000000002j)
```

The mathematical field of complex analysis is beyond the scope of this example, but many functions in the complex plane have a "branch cut", usually along the real axis or the imaginary axis. Most modern platforms support "signed zero" as specified in IEEE 754, which provides continuity of those functions on both sides of the branch cut. The following example is from the Python documentation:

```
cmath.phase(complex(-1.0, 0.0))
# Out: 3.141592653589793

cmath.phase(complex(-1.0, -0.0))
```

```
# 输出: -3.141592653589793
```

cmath模块还提供了许多与math模块直接对应的函数。

除了sqrt之外，还有exp、log、log10、三角函数及其反函数的复数版本

(sin、cos、tan、asin、acos、atan)，以及双曲函数及其反函数 (sinh、cosh、tanh、asinh、acosh、atanh)。但请注意，math.atan2（两参数反正切函数）没有复数对应版本。

```
cmath.log(1+1j)  
# 输出: (0.34657359027997264+0.7853981633974483j)
```

```
cmath.exp(1j * cmath.pi)  
# 输出: (-1+1.2246467991473532e-16j) # e 的 i pi 次方 == -1, 误差范围内
```

提供了常量pi和e。注意它们是float类型，而非complex类型。

```
type(cmath.pi)  
# 输出: <class 'float'>
```

cmath模块还提供了isinf的复数版本，以及（对于Python 3.2及以上版本）isfinite。参见“无穷大和NaN”。如果复数的实部或虚部任一为无穷大，则该复数被视为无穷大。

```
cmath.isinf(complex(float('inf'), 0.0))  
# 输出: True
```

同样，cmath模块提供了isnan的复数版本。参见“无穷大和NaN”。如果复数的实部或虚部任一为“不是数字”，则该复数被视为“不是数字”。

```
cmath.isnan(0.0, float('nan'))  
# 输出: True
```

注意，cmath没有对应于math.inf和math.nan常量（Python 3.5及以上版本）的属性

```
Python 3.x 版本 ≥ 3.5  
cmath.isinf(complex(0.0, math.inf))  
# 输出: True
```

```
cmath.isnan(complex(math.nan, 0.0))  
# 输出: True
```

```
cmath.inf  
# 异常: AttributeError: module 'cmath' has no attribute 'inf'
```

在 Python 3.5 及更高版本中，cmath 和 math 模块中都有一个 isclose 方法。

```
Python 3.x 版本 ≥ 3.5  
z = cmath.rect(*cmath.polar(1+1j))  
  
z  
# 输出: (1.0000000000000002+1.0000000000000002j)  
  
cmath.isclose(z, 1+1j)  
# 真
```

```
# Out: -3.141592653589793
```

The cmath module also provides many functions with direct counterparts from the math module.

In addition to sqrt, there are complex versions of exp, log, log10, the trigonometric functions and their inverses (sin, cos, tan, asin, acos, atan), and the hyperbolic functions and their inverses (sinh, cosh, tanh, asinh, acosh, atanh). Note however there is no complex counterpart of math.atan2, the two-argument form of arctangent.

```
cmath.log(1+1j)  
# Out: (0.34657359027997264+0.7853981633974483j)
```

```
cmath.exp(1j * cmath.pi)  
# Out: (-1+1.2246467991473532e-16j) # e to the i pi == -1, within rounding error
```

The constants pi and e are provided. Note these are float and not complex.

```
type(cmath.pi)  
# Out: <class 'float'>
```

The cmath module also provides complex versions of isinf, and (for Python 3.2+) isfinite. See "Infinity and NaN". A complex number is considered infinite if either its real part or its imaginary part is infinite.

```
cmath.isinf(complex(float('inf'), 0.0))  
# Out: True
```

Likewise, the cmath module provides a complex version of isnan. See "Infinity and NaN". A complex number is considered "not a number" if either its real part or its imaginary part is "not a number".

```
cmath.isnan(0.0, float('nan'))  
# Out: True
```

Note there is no cmath counterpart of the math.inf and math.nan constants (from Python 3.5 and higher)

```
Python 3.x 版本 ≥ 3.5  
cmath.isinf(complex(0.0, math.inf))  
# Out: True
```

```
cmath.isnan(complex(math.nan, 0.0))  
# Out: True
```

```
cmath.inf  
# 异常: AttributeError: module 'cmath' has no attribute 'inf'
```

In Python 3.5 and higher, there is an isclose method in both cmath and math modules.

```
Python 3.x 版本 ≥ 3.5  
z = cmath.rect(*cmath.polar(1+1j))  
  
z  
# Out: (1.0000000000000002+1.0000000000000002j)  
  
cmath.isclose(z, 1+1j)  
# True
```

# 第46章：复数数学

## 第46.1节：高级复数运算

模块cmath包含用于复数的附加函数。

```
import cmath
```

该模块可以计算复数的相位，单位为弧度：

```
z = 2+3j # 一个复数  
cmath.phase(z) # 0.982793723247329
```

它允许复数的笛卡尔（直角坐标）和极坐标表示之间的转换：

```
cmath.polar(z) # (3.605551275463989, 0.982793723247329)  
cmath.rect(2, cmath.pi/2) # (0+2j)
```

该模块包含复数版本的

- 指数函数和对数函数（如常，log 是自然对数，log10 是常用对数）：

```
cmath.exp(z) # (-7.315110094901103+1.0427436562359045j)  
cmath.log(z) # (1.2824746787307684+0.982793723247329j)  
cmath.log10(-100) # (2+1.3643763538418412j)
```

- 平方根：

```
cmath.sqrt(z) # (1.6741492280355401+0.8959774761298381j)
```

- 三角函数及其反函数：

```
cmath.sin(z) # (9.15449914691143-4.168906959966565j)  
cmath.cos(z) # (-4.189625690968807-9.109227893755337j)  
cmath.tan(z) # (-0.003764025641504249+1.00323862735361j)  
cmath.asin(z) # (0.5706527843210994+1.9833870299165355j)  
cmath.acos(z) # (1.0001435424737972-1.9833870299165355j)  
cmath.atan(z) # (1.4099210495965755+0.22907268296853878j)  
cmath.sin(z)**2 + cmath.cos(z)**2 # (1+0j)
```

- 双曲函数及其反函数：

```
cmath.sinh(z) # (-3.59056458998578+0.5309210862485197j)  
cmath.cosh(z) # (-3.7245455049153224+0.5118225699873846j)  
cmath.tanh(z) # (0.965385879022133-0.009884375038322495j)  
cmath.asinh(z) # (0.5706527843210994+1.9833870299165355j)  
cmath.acosh(z) # (1.9833870299165355+1.0001435424737972j)  
cmath.atanh(z) # (0.14694666622552977+1.3389725222944935j)  
cmath.cosh(z)**2 - cmath.sin(z)**2 # (1+0j)  
cmath.cosh((0+1j)*z) - cmath.cos(z) # 0j
```

# Chapter 46: Complex math

## Section 46.1: Advanced complex arithmetic

The module `cmath` includes additional functions to use complex numbers.

```
import cmath
```

This module can calculate the phase of a complex number, in radians:

```
z = 2+3j # A complex number  
cmath.phase(z) # 0.982793723247329
```

It allows the conversion between the cartesian (rectangular) and polar representations of complex numbers:

```
cmath.polar(z) # (3.605551275463989, 0.982793723247329)  
cmath.rect(2, cmath.pi/2) # (0+2j)
```

The module contains the complex version of

- Exponential and logarithmic functions (as usual, log is the natural logarithm and log10 the decimal logarithm):

```
cmath.exp(z) # (-7.315110094901103+1.0427436562359045j)  
cmath.log(z) # (1.2824746787307684+0.982793723247329j)  
cmath.log10(-100) # (2+1.3643763538418412j)
```

- Square roots:

```
cmath.sqrt(z) # (1.6741492280355401+0.8959774761298381j)
```

- Trigonometric functions and their inverses:

```
cmath.sin(z) # (9.15449914691143-4.168906959966565j)  
cmath.cos(z) # (-4.189625690968807-9.109227893755337j)  
cmath.tan(z) # (-0.003764025641504249+1.00323862735361j)  
cmath.asin(z) # (0.5706527843210994+1.9833870299165355j)  
cmath.acos(z) # (1.0001435424737972-1.9833870299165355j)  
cmath.atan(z) # (1.4099210495965755+0.22907268296853878j)  
cmath.sin(z)**2 + cmath.cos(z)**2 # (1+0j)
```

- Hyperbolic functions and their inverses:

```
cmath.sinh(z) # (-3.59056458998578+0.5309210862485197j)  
cmath.cosh(z) # (-3.7245455049153224+0.5118225699873846j)  
cmath.tanh(z) # (0.965385879022133-0.009884375038322495j)  
cmath.asinh(z) # (0.5706527843210994+1.9833870299165355j)  
cmath.acosh(z) # (1.9833870299165355+1.0001435424737972j)  
cmath.atanh(z) # (0.14694666622552977+1.3389725222944935j)  
cmath.cosh(z)**2 - cmath.sin(z)**2 # (1+0j)  
cmath.cosh((0+1j)*z) - cmath.cos(z) # 0j
```

## 第46.2节：基本复数运算

Python 内置支持复数运算。虚数单位用 `j` 表示：

```
z = 2+3j # 一个复数  
w = 1-7j # 另一个复数
```

复数可以进行加法、减法、乘法、除法和指数运算：

```
z + w # (3-4j)  
z - w # (1+10j)  
z * w # (23-11j)  
z / w # (-0.38+0.34j)  
z**3 # (-46+9j)
```

Python 还可以提取复数的实部和虚部，并计算它们的绝对值和共轭：

```
z.real # 2.0  
z.imag # 3.0  
abs(z) # 3.605551275463989  
z.conjugate() # (2-3j)
```

## Section 46.2: Basic complex arithmetic

Python has built-in support for complex arithmetic. The imaginary unit is denoted by `j`:

```
z = 2+3j # A complex number  
w = 1-7j # Another complex number
```

Complex numbers can be summed, subtracted, multiplied, divided and exponentiated:

```
z + w # (3-4j)  
z - w # (1+10j)  
z * w # (23-11j)  
z / w # (-0.38+0.34j)  
z**3 # (-46+9j)
```

Python can also extract the real and imaginary parts of complex numbers, and calculate their absolute value and conjugate:

```
z.real # 2.0  
z.imag # 3.0  
abs(z) # 3.605551275463989  
z.conjugate() # (2-3j)
```

# 第47章：集合模块

内置的collections包提供了几种专门的、灵活的集合类型，这些类型既具有高性能，又为通用集合类型dict、list、tuple和set提供了替代方案。该模块还定义了描述不同集合功能类型（例如MutableSet和ItemsView）。

## 第47.1节：collections.Counter

Counter 是 dict 的子类，允许你轻松计数对象。它提供了用于处理你正在计数的对象频率的实用方法。

```
import collections  
counts = collections.Counter([1,2,3])
```

上述代码创建了一个名为 counts 的对象，该对象包含传递给构造函数的所有元素的频率。  
此示例的值为 Counter({1: 1, 2: 1, 3: 1})

### 构造函数示例

#### 字母计数器

```
>>> collections.Counter('Happy Birthday')  
Counter({'a': 2, 'p': 2, 'y': 2, 'i': 1, 'r': 1, 'B': 1, ' ': 1, 'H': 1, 'd': 1, 'h': 1, 't': 1})
```

#### 单词计数器

```
>>> collections.Counter('我是山姆 山姆 我是 那个山姆-我是！我不喜欢那个山姆-我'.split())  
Counter({'I': 3, 'Sam': 2, 'Sam-I-am': 2, 'That': 2, 'am': 2, 'do': 1, 'Sam-I-am!': 1, 'that': 1, 'not': 1, 'like': 1})
```

#### 配方

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

#### 获取单个元素的计数

```
>>> c['a']  
4
```

#### 设置单个元素的计数

```
>>> c['c'] = -3  
>>> c  
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

#### 获取计数器中元素的总数 (4 + 2 + 0 - 3)

```
>>> sum(c.itervalues()) # 负数也会被计数！  
3
```

#### 获取元素（仅保留计数器值为正的元素）

# Chapter 47: Collections module

The built-in `collections` package provides several specialized, flexible collection types that are both high-performance and provide alternatives to the general collection types of `dict`, `list`, `tuple` and `set`. The module also defines abstract base classes describing different types of collection functionality (such as `MutableSet` and `ItemsView`).

## Section 47.1: collections.Counter

`Counter` is a dict sub class that allows you to easily count objects. It has utility methods for working with the frequencies of the objects that you are counting.

```
import collections  
counts = collections.Counter([1,2,3])
```

the above code creates an object, counts, which has the frequencies of all the elements passed to the constructor.  
This example has the value Counter({1: 1, 2: 1, 3: 1})

### Constructor examples

#### Letter Counter

```
>>> collections.Counter('Happy Birthday')  
Counter({'a': 2, 'p': 2, 'y': 2, 'i': 1, 'r': 1, 'B': 1, ' ': 1, 'H': 1, 'd': 1, 'h': 1, 't': 1})
```

#### Word Counter

```
>>> collections.Counter('I am Sam Sam I am That Sam-I-am That Sam-I-am! I do not like that Sam-I-am'.split())  
Counter({'I': 3, 'Sam': 2, 'Sam-I-am': 2, 'That': 2, 'am': 2, 'do': 1, 'Sam-I-am!': 1, 'that': 1, 'not': 1, 'like': 1})
```

### Recipes

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

#### Get count of individual element

```
>>> c['a']  
4
```

#### Set count of individual element

```
>>> c['c'] = -3  
>>> c  
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

#### Get total number of elements in counter (4 + 2 + 0 - 3)

```
>>> sum(c.itervalues()) # negative numbers are counted!  
3
```

#### Get elements (only those with positive counter are kept)

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

移除计数为0或负值的键

```
>>> c = collections.Counter()
Counter({'a': 4, 'b': 2})
```

移除所有元素

```
>>> c.clear()
>>> c
Counter()
```

添加或移除单个元素

```
>>> c.update({'a': 3, 'b': 3})
>>> c.update({'a': 2, 'c': 2}) # 如果存在则添加, 不存在则设置
>>> c
Counter({'a': 5, 'b': 3, 'c': 2})
>>> c.subtract({'a': 3, 'b': 3, 'c': 3}) # 执行减法(允许负值)
>>> c
Counter({'a': 2, 'b': 0, 'c': -1})
```

## 第47.2节 : collections.OrderedDict

Python字典中键的顺序是任意的：它们不受添加顺序的控制。

例如：

```
>>> d = {'foo': 5, 'bar': 6}
>>> print(d)
{'foo': 5, 'bar': 6}
>>> d['baz'] = 7
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6}
>>> d['foobar'] = 8
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6, 'foobar': 8}
``
```

(上述任意顺序意味着你运行上述代码时，结果可能与这里显示的不同。)

键出现的顺序就是它们被迭代的顺序，例如使用for循环时。

collections.OrderedDict类提供了保留键顺序的字典对象。 OrderedDict可以如下面所示通过一系列有序项（这里是键值对元组列表）创建：

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('foo', 5), ('bar', 6)])
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6)])
>>> d['baz'] = 7
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7)])
>>> d['foobar'] = 8
```

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

Remove keys with 0 or negative value

```
>>> c = collections.Counter()
Counter({'a': 4, 'b': 2})
```

Remove everything

```
>>> c.clear()
>>> c
Counter()
```

Add remove individual elements

```
>>> c.update({'a': 3, 'b': 3})
>>> c.update({'a': 2, 'c': 2}) # adds to existing, sets if they don't exist
>>> c
Counter({'a': 5, 'b': 3, 'c': 2})
>>> c.subtract({'a': 3, 'b': 3, 'c': 3}) # subtracts (negative values are allowed)
>>> c
Counter({'a': 2, 'b': 0, 'c': -1})
```

## Section 47.2: collections.OrderedDict

The order of keys in Python dictionaries is arbitrary: they are not governed by the order in which you add them.

For example:

```
>>> d = {'foo': 5, 'bar': 6}
>>> print(d)
{'foo': 5, 'bar': 6}
>>> d['baz'] = 7
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6}
>>> d['foobar'] = 8
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6, 'foobar': 8}
``
```

(The arbitrary ordering implied above means that you may get different results with the above code to that shown here.)

The order in which the keys appear is the order which they would be iterated over, e.g. using a for loop.

The `collections.OrderedDict` class provides dictionary objects that retain the order of keys. OrderedDicts can be created as shown below with a series of ordered items (here, a list of tuple key-value pairs):

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('foo', 5), ('bar', 6)])
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6)])
>>> d['baz'] = 7
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7)])
>>> d['foobar'] = 8
```

```
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

或者我们可以创建一个空的OrderedDict然后添加项：

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

遍历一个OrderedDict允许按添加的顺序访问键。

如果我们给一个已存在的键赋予一个新值，会发生什么？

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

该键在OrderedDict中保持其原有的位置。

## 第47.3节：collections.defaultdict

`collections.defaultdict(default_factory)` 返回一个 `dict` 的子类，该子类对缺失的键有默认值。参数应是一个函数，当无参数调用时返回默认值。如果没有传入参数，默认值为None。

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

返回一个 `defaultdict` 的引用，该 `defaultdict` 会使用其 `default_factory` 方法创建字符串对象。

使用`defaultdict`的典型用法是使用内置类型之一，如`str`、`int`、`list`或 `dict`作为`default_factory`，因为这些类型在无参数调用时返回空类型：

```
>>> str()
''
>>> int()
0
>>> list
[]
```

调用不存在键的`defaultdict`不会像普通字典那样产生错误。

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

另一个使用`int`的例子：

```
>>> fruit_counts = defaultdict(int)
>>> fruit_counts['apple'] += 2 # 不会发生错误
>>> fruit_counts
default_dict(int, {'apple': 2})
>>> fruit_counts['banana'] # 不会发生错误
```

```
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

Or we can create an empty `OrderedDict` and then add items:

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

Iterating through an `OrderedDict` allows key access in the order they were added.

What happens if we assign a new value to an existing key?

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

The key retains its original place in the `OrderedDict`.

## Section 47.3: collections.defaultdict

`collections.defaultdict(default_factory)` returns a subclass of `dict` that has a default value for missing keys. The argument should be a function that returns the default value when called with no arguments. If there is nothing passed, it defaults to `None`.

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

returns a reference to a `defaultdict` that will create a string object with its `default_factory` method.

A typical usage of `defaultdict` is to use one of the builtin types such as `str`, `int`, `list` or `dict` as the `default_factory`, since these return empty types when called with no arguments:

```
>>> str()
''
>>> int()
0
>>> list
[]
```

Calling the `defaultdict` with a key that does not exist does not produce an error as it would in a normal dictionary.

```
>>> state_capitals['Alaska']
''
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ''})
```

Another example with `int`:

```
>>> fruit_counts = defaultdict(int)
>>> fruit_counts['apple'] += 2 # No errors should occur
>>> fruit_counts
default_dict(int, {'apple': 2})
>>> fruit_counts['banana'] # No errors should occur
```

```
0  
>>> fruit_counts # 创建了一个新键  
defaultdict(int, {'apple': 2, 'banana': 0})
```

普通字典方法使用默认字典

```
>>> state_capitals['阿拉巴马'] = '蒙哥马利'  
>>> state_capitals  
defaultdict(<class 'str'>, {'阿拉巴马': '蒙哥马利', '阿拉斯加': ''})
```

使用list作为default\_factory会为每个新键创建一个列表。

```
>>> s = [('北卡罗来纳', '罗利'), ('弗吉尼亚', '里士满'), ('华盛顿', '西雅图'), ('北卡罗来纳', '阿什维尔')]  
>>> dd = collections.defaultdict(list)  
>>> for k, v in s:  
...     dd[k].append(v)  
>>> dd  
defaultdict(<class 'list'>,  
{'弗吉尼亚': ['里士满'],  
'北卡罗来纳': ['罗利', '阿什维尔'],  
'华盛顿': ['西雅图']})
```

## 第47.4节 : collections.namedtuple

使用namedtuple定义一个新类型Person，方法如下：

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

第二个参数是元组将拥有的属性列表。你也可以将这些属性列为以空格或逗号分隔的字符串：

```
Person = namedtuple('Person', 'age, height, name')
```

或者

```
Person = namedtuple('Person', 'age height name')
```

定义后，可以通过调用该对象并传入必要参数来实例化命名元组，例如：

```
dave = Person(30, 178, 'Dave')
```

也可以使用命名参数：

```
jack = Person(age=30, height=178, name='Jack S.')
```

现在你可以访问命名元组的属性：

```
print(jack.age) # 30  
print(jack.name) # 'Jack S.'
```

namedtuple构造函数的第一个参数（在我们的示例中为'Person'）是typename。通常构造函数名和类型名相同，但它们也可以不同：

```
Human = namedtuple('Person', 'age, height, name')  
dave = Human(30, 178, 'Dave')
```

```
0  
>>> fruit_counts # A new key is created  
defaultdict(int, {'apple': 2, 'banana': 0})
```

Normal dictionary methods work with the default dictionary

```
>>> state_capitals['Alabama'] = 'Montgomery'  
>>> state_capitals  
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ''})
```

Using list as the default\_factory will create a list for each new key.

```
>>> s = [('NC', 'Raleigh'), ('VA', 'Richmond'), ('WA', 'Seattle'), ('NC', 'Asheville')]  
>>> dd = collections.defaultdict(list)  
>>> for k, v in s:  
...     dd[k].append(v)  
>>> dd  
defaultdict(<class 'list'>,  
{'VA': ['Richmond'],  
'NC': ['Raleigh', 'Asheville'],  
'WA': ['Seattle']})
```

## Section 47.4: collections.namedtuple

Define a new type Person using [namedtuple](#) like this:

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

The second argument is the list of attributes that the tuple will have. You can list these attributes also as either space or comma separated string:

```
Person = namedtuple('Person', 'age, height, name')
```

or

```
Person = namedtuple('Person', 'age height name')
```

Once defined, a named tuple can be instantiated by calling the object with the necessary parameters, e.g.:

```
dave = Person(30, 178, 'Dave')
```

Named arguments can also be used:

```
jack = Person(age=30, height=178, name='Jack S.')
```

Now you can access the attributes of the namedtuple:

```
print(jack.age) # 30  
print(jack.name) # 'Jack S.'
```

The first argument to the namedtuple constructor (in our example 'Person') is the typename. It is typical to use the same word for the constructor and the typename, but they can be different:

```
Human = namedtuple('Person', 'age, height, name')  
dave = Human(30, 178, 'Dave')
```

```
print(dave) # yields: Person(age=30, height=178, name='Dave')
```

## 第47.5节 : collections.deque

返回一个新的deque对象，使用来自可迭代对象的数据从左到右（使用append()）初始化。如果未指定可迭代对象，则新的deque为空。

Deque是栈和队列的泛化（名称发音为“deck”，是“double-endedqueue”的缩写）。Deque支持线程安全、内存高效的从两端追加和弹出操作，且在任一方向上的性能大致为O(1)。

虽然列表对象支持类似操作，但它们针对快速的固定长度操作进行了优化，对于pop(0)和insert(0, v)操作会产生O(n)的内存移动开销，因为这些操作会改变底层数据表示的大小和位置。

版本2.4新增。

如果未指定 maxlen 或其值为 None，deque 可以增长到任意长度。否则，deque 的长度被限制为指定的最大长度。一旦有界长度的 deque 已满，当添加新元素时，会从另一端丢弃相应数量的元素。有界长度的 deque 提供了类似 Unix 中 tail 过滤器的功能。它们也适用于跟踪事务和其他只关注最新活动的数据池。

版本2.6更改：添加了 maxlen 参数。

```
>>> from collections import deque
>>> d = deque('ghi')          # 创建一个包含三个元素的新双端队列
>>> for elem in d:           # 遍历双端队列中的元素
...     print elem.upper()
G
H
I
>
>
d.append('j')                # 在右侧添加一个新元素
>>> d.appendleft('f')        # 在左侧添加一个新元素
>>> d                         # 显示双端队列的表示形式
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                  # 返回并移除最右侧的元素
'j'
>>> d.popleft()              # 返回并移除最左侧的元素
'f'
>>> list(d)                  # 列出双端队列的内容
['g', 'h', 'i']
>>> d[0]                      # 查看最左侧的元素
'g'
>>> d[-1]                     # 查看最右侧的元素
'i'

>>> list(reversed(d))         # 反向列出双端队列的内容
['i', 'h', 'g']
>>> 'h' in d                  # 在双端队列中搜索
True
>>> d.extend('jkl')           # 一次添加多个元素
>>> d                         # 右旋转
deque(['g', 'h', 'i', 'j', 'k', 'l'])
```

```
print(dave) # yields: Person(age=30, height=178, name='Dave')
```

## Section 47.5: collections.deque

Returns a new deque object initialized left-to-right (using append()) with data from iterable. If iterable is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

New in version 2.4.

If maxlen is not specified or is `None`, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the tail filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Changed in version 2.6: Added maxlen parameter.

```
>>> from collections import deque
>>> d = deque('ghi')          # make a new deque with three items
>>> for elem in d:           # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')            # add a new entry to the right side
>>> d.appendleft('f')         # add a new entry to the left side
>>> d                         # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                  # return and remove the rightmost item
'j'
>>> d.popleft()              # return and remove the leftmost item
'f'
>>> list(d)                  # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                      # peek at leftmost item
'g'
>>> d[-1]                     # peek at rightmost item
'i'

>>> list(reversed(d))         # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                  # search the deque
True
>>> d.extend('jkl')           # add multiple elements at once
>>> d                         # right rotation
deque(['g', 'h', 'i', 'j', 'k', 'l'])
```

```

deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)          # 左旋转
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))    # 创建一个新的反向deque
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()            # 清空deque
>>> d.pop()               # 无法从空deque中弹出
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <toplevel>
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')   # extendleft() 会反转输入顺序
>>> d
deque(['c', 'b', 'a'])

```

Source: <https://docs.python.org/2/library/collections.html>

## 第47.6节 : collections.ChainMap

ChainMap 是在 版本3.3中新增的

返回一个新的 ChainMap 对象，给定多个 maps。该对象将多个字典或其他映射组合在一起，创建一个单一的、可更新的视图。

ChainMap 对于管理嵌套上下文和覆盖非常有用。在Python领域的一个例子是在Django模板引擎中实现的 Context 类。它对于快速链接多个映射非常有用，使结果可以作为一个整体来处理。通常比创建一个新的字典并多次调用 update() 要快得多。

任何时候当存在一系列查找值时，都可能适合使用 ChainMap。一个例子是同时拥有用户指定的值和默认值字典。另一个例子是在Web使用中（如Django或Flask）发现的 POST 和 GET 参数映射。通过使用 ChainMap，可以返回两个不同字典的组合视图。

maps 参数列表的顺序是从先查找到后查找。查找操作会依次搜索底层映射，直到找到键为止。相比之下，写入、更新和删除操作只作用于第一个映射。

```

import collections

# 定义两个字典，至少有一些键是重叠的。
dict1 = {'apple': 1, 'banana': 2}
dict2 = {'coconut': 1, 'date': 1, 'apple': 3}

# 创建两个ChainMap，字典的顺序不同。
combined_dict = collections.ChainMap(dict1, dict2)
reverse_ordered_dict = collections.ChainMap(dict2, dict1)

```

请注意顺序对后续查找中首先找到的值的影响

```

for k, v in combined_dict.items():
    print(k, v)

date 1
apple 1
banana 2

```

```

deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)          # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))    # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()            # empty the deque
>>> d.pop()               # cannot pop from an empty deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <toplevel>
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')   # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])

```

Source: <https://docs.python.org/2/library/collections.html>

## Section 47.6: collections.ChainMap

ChainMap is new in **version 3.3**

Returns a new ChainMap object given a number of maps. This object groups multiple dicts or other mappings together to create a single, updateable view.

ChainMaps are useful managing nested contexts and overlays. An example in the python world is found in the implementation of the Context class in Django's template engine. It is useful for quickly linking a number of mappings so that the result can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple update() calls.

Anytime one has a chain of lookup values there can be a case for ChainMap. An example includes having both user specified values and a dictionary of default values. Another example is the POST and GET parameter maps found in web use, e.g. Django or Flask. Through the use of ChainMap one returns a combined view of two distinct dictionaries.

The maps parameter list is ordered from first-searched to last-searched. Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

```

import collections

# define two dictionaries with at least some keys overlapping.
dict1 = {'apple': 1, 'banana': 2}
dict2 = {'coconut': 1, 'date': 1, 'apple': 3}

# create two ChainMaps with different ordering of those dicts.
combined_dict = collections.ChainMap(dict1, dict2)
reverse_ordered_dict = collections.ChainMap(dict2, dict1)

```

Note the impact of order on which value is found first in the subsequent lookup

```

for k, v in combined_dict.items():
    print(k, v)

date 1
apple 1
banana 2

```

```
coconut 1
```

```
for k, v in reverse_ordered_dict.items():
    print(k, v)
```

```
date 1
apple 3
banana 2
coconut 1
```

```
coconut 1
```

```
for k, v in reverse_ordered_dict.items():
    print(k, v)
```

```
date 1
apple 3
banana 2
coconut 1
```

# 第48章：操作符模块

## 第48.1节：Itemgetter

使用itemgetter按值对字典的键值对进行分组：

```
from itertools import groupby
from operator import itemgetter
adict = {'a': 1, 'b': 5, 'c': 1}

dict((i, dict(v)) for i, v in groupby(adict.items(), itemgetter(1)))
# 输出: {1: {'a': 1, 'c': 1}, 5: {'b': 5}}
```

这等价于（但更快）的一个 lambda 函数，如下所示：

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

或者先按第二个元素，再按第一个元素对元组列表进行排序：

```
alist_of_tuples = [(5, 2), (1, 3), (2, 2)]
sorted(alist_of_tuples, key=itemgetter(1, 0))
# 输出: [(2, 2), (5, 2), (1, 3)]
```

## 第48.2节：作为中缀运算符替代的运算符

对于每个中缀运算符，例如+，都有一个operator函数（operator.add对应+）：

```
1 + 1
# 输出: 2
from operator import add
add(1, 1)
# 输出: 2
```

尽管主要文档说明算术运算符只允许数值输入，但实际上可能是可能的：

```
from operator import mul
mul('a', 10)
# 输出: 'aaaaaaaaaa'
mul([3], 3)
# 输出: [3, 3, 3]
```

另请参见：[官方Python文档中操作到运算符函数的映射](#)。

## 第48.3节：Methodcaller

替代这个显式调用方法的lambda函数：

```
alist = ['狼', '羊', '鸭']
list(filter(lambda x: x.startswith('d'), alist))      # 仅保留以 'd' 开头的元素
# 输出: ['鸭']
```

可以使用一个执行相同操作的操作符函数：

```
from operator import methodcaller
```

# Chapter 48: Operator module

## Section 48.1: Itemgetter

Grouping the key-value pairs of a dictionary by the value with itemgetter:

```
from itertools import groupby
from operator import itemgetter
adict = {'a': 1, 'b': 5, 'c': 1}

dict((i, dict(v)) for i, v in groupby(adict.items(), itemgetter(1)))
# Output: {1: {'a': 1, 'c': 1}, 5: {'b': 5}}
```

which is equivalent (but faster) to a lambda function like this:

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

Or sorting a list of tuples by the second element first the first element as secondary:

```
alist_of_tuples = [(5, 2), (1, 3), (2, 2)]
sorted(alist_of_tuples, key=itemgetter(1, 0))
# Output: [(2, 2), (5, 2), (1, 3)]
```

## Section 48.2: Operators as alternative to an infix operator

For every infix operator, e.g. + there is an operator-function (operator.add for +):

```
1 + 1
# Output: 2
from operator import add
add(1, 1)
# Output: 2
```

even though the main documentation states that for the arithmetic operators only numerical input is allowed it is possible:

```
from operator import mul
mul('a', 10)
# Output: 'aaaaaaaaaa'
mul([3], 3)
# Output: [3, 3, 3]
```

See also: [mapping from operation to operator function in the official Python documentation](#).

## Section 48.3: Methodcaller

Instead of this lambda-function that calls the method explicitly:

```
alist = ['wolf', 'sheep', 'duck']
list(filter(lambda x: x.startswith('d'), alist))      # Keep only elements that start with 'd'
# Output: ['duck']
```

one could use a operator-function that does the same:

```
from operator import methodcaller
```

```
list(filter(methodcaller('startswith', 'd'), alist)) # 功能相同但更快。  
# 输出: ['鸭']
```

```
list(filter(methodcaller('startswith', 'd'), alist)) # Does the same but is faster.  
# Output: ['duck']
```

# 第49章：JSON模块

## 第49.1节：将数据存储到文件中

下面的代码片段将存储在 `d` 中的数据编码为JSON，并存储到文件中（将 `filename` 替换为实际的文件名）。

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

## 第49.2节：从文件中检索数据

下面的代码片段打开一个JSON编码的文件（将`filename`替换为实际的文件名），并返回存储在文件中的对象。

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

## 第49.3节：格式化JSON输出

假设我们有以下数据：

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

仅仅将其作为 JSON 转储在这里并没有做任何特别的处理：

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

### 设置缩进以获得更美观的输出

如果我们想要美观打印，可以设置一个`indent`大小：

```
>>> print(json.dumps(data, indent=2))
{
    "cats": [
        {
            "name": "Tubbs",
            "color": "white"
        },
        {
            "name": "Pepper",
            "color": "black"
        }
    ]
}
```

# Chapter 49: JSON Module

## Section 49.1: Storing data in a file

The following snippet encodes the data stored in `d` into JSON and stores it in a file (replace `filename` with the actual name of the file).

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

## Section 49.2: Retrieving data from a file

The following snippet opens a JSON encoded file (replace `filename` with the actual name of the file) and returns the object that is stored in the file.

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

## Section 49.3: Formatting JSON output

Let's say we have the following data:

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Just dumping this as JSON does not do anything special here:

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

### Setting indentation to get prettier output

If we want pretty printing, we can set an `indent` size:

```
>>> print(json.dumps(data, indent=2))
{
    "cats": [
        {
            "name": "Tubbs",
            "color": "white"
        },
        {
            "name": "Pepper",
            "color": "black"
        }
    ]
}
```

## 按字母顺序排序键以获得一致的输出

默认情况下，输出中键的顺序是不确定的。我们可以按字母顺序获取它们，以确保我们总是得到相同的输出：

```
>>> print(json.dumps(data, sort_keys=True))
{"cats": [{"color": "white", "name": "Tubbs"}, {"color": "black", "name": "Pepper"}]}
```

## 去除空白以获得紧凑的输出

我们可能想去除不必要的空格，这可以通过设置与默认值不同的分隔符字符串来实现，默认值为'，' 和 ':':

```
>>> print(json.dumps(data, separators=(',', ':')))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

## 第49.4节：`load` 与 `loads`，`dump` 与 `dumps`

json模块包含用于从Unicode字符串读取和写入，以及从文件读取和写入的函数。这些函数通过名称末尾的 s来区分。在这些示例中，我们使用了StringIO对象，但相同的函数也适用于任何类文件对象。

这里我们使用基于字符串的函数：

```
import json

data = {u"foo": u"bar", u"baz": []}
json_string = json.dumps(data)
# u'{"foo": "bar", "baz": []}'
json.loads(json_string)
# {u"foo": u"bar", u"baz": []}
```

这里我们使用基于文件的函数：

```
import json

from io import StringIO

json_file = StringIO()
data = {u"foo": u"bar", u"baz": []}
json.dump(data, json_file)
json_file.seek(0) # 在读取前回到文件开头
json_file_content = json_file.read()
# u'{"foo": "bar", "baz": []}'
json_file.seek(0) # 在读取前回到文件开头
json.load(json_file)
# {u"foo": u"bar", u"baz": []}
```

如你所见，主要区别在于转储json数据时必须传入文件句柄，而不是捕获返回值。同样值得注意的是，读取或写入前必须将文件指针移到开头，以避免数据损坏。打开文件时，光标默认位于位置0，因此下面的写法也可行：

```
import json

json_file_path = './data.json'
data = {u"foo": u"bar", u"baz": []}
```

## Sorting keys alphabetically to get consistent output

By default the order of keys in the output is undefined. We can get them in alphabetical order to make sure we always get the same output:

```
>>> print(json.dumps(data, sort_keys=True))
{"cats": [{"color": "white", "name": "Tubbs"}, {"color": "black", "name": "Pepper"}]}
```

## Getting rid of whitespace to get compact output

We might want to get rid of the unnecessary spaces, which is done by setting separator strings different from the default ', ' and ':':

```
>>> print(json.dumps(data, separators=(',', ':')))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

## Section 49.4: `load` vs `loads`, `dump` vs `dumps`

The json module contains functions for both reading and writing to and from unicode strings, and reading and writing to and from files. These are differentiated by a trailing s in the function name. In these examples we use a StringIO object, but the same functions would apply for any file-like object.

Here we use the string-based functions:

```
import json

data = {u"foo": u"bar", u"baz": []}
json_string = json.dumps(data)
# u'{"foo": "bar", "baz": []}'
json.loads(json_string)
# {u"foo": u"bar", u"baz": []}
```

And here we use the file-based functions:

```
import json

from io import StringIO

json_file = StringIO()
data = {u"foo": u"bar", u"baz": []}
json.dump(data, json_file)
json_file.seek(0) # Seek back to the start of the file before reading
json_file_content = json_file.read()
# u'{"foo": "bar", "baz": []}'
json_file.seek(0) # Seek back to the start of the file before reading
json.load(json_file)
# {u"foo": u"bar", u"baz": []}
```

As you can see the main difference is that when dumping json data you must pass the file handle to the function, as opposed to capturing the return value. Also worth noting is that you must seek to the start of the file before reading or writing, in order to avoid data corruption. When opening a file the cursor is placed at position 0, so the below would also work:

```
import json

json_file_path = './data.json'
data = {u"foo": u"bar", u"baz": []}
```

```

with open(json_file_path, 'w') as json_file:
    json.dump(data, json_file)

with open(json_file_path) as json_file:
    json_file_content = json_file.read()
    # u'{"foo": "bar", "baz": []}'

with open(json_file_path) as json_file:
    json.load(json_file)
    # {u"foo": u"bar", u"baz": []}

```

同时掌握两种处理json数据的方法，可以让你以惯用且高效的方式处理基于json构建的格式，例如pyspark的逐行json：

```

# 从文件加载
data = [json.loads(line) for line in open(file_path).readlines()]

# 写入到文件
with open(file_path, 'w') as json_file:
    for item in data:
        json.dump(item, json_file)
        json_file.write("\n")

```

## 第49.5节：从命令行调用 `json.tool` 来美化打印JSON输出

给定一个类似于 "foo.json" 的JSON文件：

```
{"foo": {"bar": {"baz": 1}}}
```

我们可以直接从命令行调用该模块（将文件名作为参数传入）来美化打印它：

```
$ python -m json.tool foo.json
{
    "foo": {
        "bar": {
            "baz": 1
        }
    }
}
```

该模块也可以从标准输入读取数据，因此（在Bash中）我们同样可以这样做：

```
$ cat foo.json | python -m json.tool
```

## 第49.6节：JSON编码自定义对象

如果我们只是尝试以下操作：

```

import json
from datetime import datetime
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}
print(json.dumps(data))

```

我们得到一个错误，提示TypeError: datetime.datetime(2016, 9, 26, 4, 44) 无法进行 JSON序列化。

为了能够正确序列化datetime对象，我们需要编写自定义代码来转换它：

```

with open(json_file_path, 'w') as json_file:
    json.dump(data, json_file)

with open(json_file_path) as json_file:
    json_file_content = json_file.read()
    # u'{"foo": "bar", "baz": []}'

with open(json_file_path) as json_file:
    json.load(json_file)
    # {u"foo": u"bar", u"baz": []}

```

Having both ways of dealing with json data allows you to idiomatically and efficiently work with formats which build upon json, such as pyspark's json-per-line:

```

# loading from a file
data = [json.loads(line) for line in open(file_path).readlines()]

# dumping to a file
with open(file_path, 'w') as json_file:
    for item in data:
        json.dump(item, json_file)
        json_file.write('\n')

```

## Section 49.5: Calling `json.tool` from the command line to pretty-print JSON output

Given some JSON file "foo.json" like:

```
{"foo": {"bar": {"baz": 1}}}
```

we can call the module directly from the command line (passing the filename as an argument) to pretty-print it:

```
$ python -m json.tool foo.json
{
    "foo": {
        "bar": {
            "baz": 1
        }
    }
}
```

The module will also take input from STDOUT, so (in Bash) we equally could do:

```
$ cat foo.json | python -m json.tool
```

## Section 49.6: JSON encoding custom objects

If we just try the following:

```

import json
from datetime import datetime
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}
print(json.dumps(data))

```

we get an error saying `TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializable.`

To be able to serialize the datetime object properly, we need to write custom code for how to convert it:

```

class DatetimeJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        try:
            return obj.isoformat()
        except AttributeError:
            # obj没有isoformat方法；让内置的JSON编码器处理它
            return super(DatetimeJSONEncoder, self).default(obj)

```

然后使用这个编码器类替代json.dumps：

```

encoder = DatetimeJSONEncoder()
print(encoder.encode(data))
# 输出 {"datetime": "2016-09-26T04:44:00"}

```

## 第49.7节：从Python字典创建JSON

```

import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)

```

上述代码片段将返回以下内容：

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

## 第49.8节：从JSON创建Python字典

```

import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)

```

上述代码片段将返回以下内容：

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

```

class DatetimeJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        try:
            return obj.isoformat()
        except AttributeError:
            # obj has no isoformat method; let the builtin JSON encoder handle it
            return super(DatetimeJSONEncoder, self).default(obj)

```

and then use this encoder class instead of json.dumps:

```

encoder = DatetimeJSONEncoder()
print(encoder.encode(data))
# prints {"datetime": "2016-09-26T04:44:00"}

```

## Section 49.7: Creating JSON from Python dict

```

import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)

```

The above snippet will return the following:

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

## Section 49.8: Creating Python dict from JSON

```

import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)

```

The above snippet will return the following:

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

# 视频：Python 数据 科学与机器 学习训练营

学习如何使用 NumPy、Pandas、Seaborn、  
Matplotlib、Plotly、Scikit-Learn、机器学习、  
Tensorflow 等！



- ✓ 使用 Python 进行数据科学和机器学习
- ✓ 使用 Spark 进行大数据分析
- ✓ 实现机器学习算法
- ✓ 学习使用 NumPy 处理数值数据
- ✓ 学习使用 Pandas 进行数据分析
- ✓ 学习使用 Matplotlib 进行 Python 绘图
- ✓ 学习使用 Seaborn 进行统计图表绘制
- ✓ 使用 Plotly 进行交互式动态可视化
- ✓ 使用 SciKit-Learn 进行机器学习任务
- ✓ K-均值聚类
- ✓ 逻辑回归
- ✓ 线性回归
- ✓ 随机森林和决策树
- ✓ 神经网络
- ✓ 支持向量机今天观看→

# VIDEO: Python for Data Science and Machine Learning Bootcamp

Learn how to use NumPy, Pandas, Seaborn,  
Matplotlib , Plotly, Scikit-Learn , Machine Learning,  
Tensorflow, and more!



- ✓ Use Python for Data Science and Machine Learning
- ✓ Use Spark for Big Data Analysis
- ✓ Implement Machine Learning Algorithms
- ✓ Learn to use NumPy for Numerical Data
- ✓ Learn to use Pandas for Data Analysis
- ✓ Learn to use Matplotlib for Python Plotting
- ✓ Learn to use Seaborn for statistical plots
- ✓ Use Plotly for interactive dynamic visualizations
- ✓ Use SciKit-Learn for Machine Learning Tasks
- ✓ K-Means Clustering
- ✓ Logistic Regression
- ✓ Linear Regression
- ✓ Random Forest and Decision Trees
- ✓ Neural Networks
- ✓ Support Vector Machines

**Watch Today →**

# 第50章：Sqlite3模块

## 第50.1节：Sqlite3 - 不需要单独的服务器进程

sqlite3模块由Gerhard Häring编写。要使用该模块，必须首先创建一个表示数据库的Connection对象  
这里数据将存储在example.db文件中：

```
import sqlite3
conn = sqlite3.connect('example.db')
```

你也可以提供特殊名称 :memory: 来创建一个内存中的数据库。一旦你有了连接对象，你可以  
创建一个游标对象并调用它的 execute() 方法来执行 SQL 命令：

```
c = conn.cursor()

# 创建表
c.execute("CREATE TABLE stocks
(date text, trans text, symbol text, qty real, price real)")

# 插入一行数据
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# 保存 (提交) 更改
conn.commit()

# 如果我们完成了操作，也可以关闭连接。
# 只要确保所有更改已提交，否则更改将丢失。
conn.close()
```

## 第50.2节：从数据库获取值及错误处理

从SQLite3数据库获取值。

打印由select查询返回的行值

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute("SELECT * from table_name where id=cust_id")
for row in c:
    print row # 将是一个列表
```

使用 fetchone() 方法获取单个匹配项

```
print c.fetchone()
```

对于多行，使用 fetchall() 方法

```
a=c.fetchall() #这类似于之前使用的 list(cursor) 方法
for row in a:
    print row
```

错误处理可以使用内置函数 sqlite3.Error 来完成

# Chapter 50: Sqlite3 Module

## Section 50.1: Sqlite3 - Not require separate server process

The sqlite3 module was written by Gerhard Häring. To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the example.db file:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name :memory: to create a database in RAM. Once you have a Connection, you can create a Cursor object and call its execute() method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
            (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

## Section 50.2: Getting the values from the database and Error handling

Fetching the values from the SQLite3 database.

Print row values returned by select query

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute("SELECT * from table_name where id=cust_id")
for row in c:
    print row # will be a list
```

To fetch single matching fetchone() method

```
print c.fetchone()
```

For multiple rows use fetchall() method

```
a=c.fetchall() #which is similar to list(cursor) method used previously
for row in a:
    print row
```

Error handling can be done using sqlite3.Error built in function

```
try:  
    #SQL 代码  
except sqlite3.Error as e:  
    print "发生错误:", e.args[0]
```

```
try:  
    #SQL Code  
except sqlite3.Error as e:  
    print "An error occurred:", e.args[0]
```

# 第51章：os 模块

## 参数

	详情
路径	指向文件的路径。路径分隔符可以由 <code>os.path.sep</code> 确定。
模式	所需权限，八进制表示（例如0700）

该模块提供了一种使用操作系统相关功能的跨平台方法。

## 第51.1节：makedirs - 递归创建目录

给定一个本地目录，内容如下：

```
└-- dir1
    ├── subdir1
    └── subdir2
```

我们想在一个尚不存在的新目录dir2下创建相同的subdir1和subdir2子目录。

```
import os
```

```
os.makedirs("./dir2/subdir1")
os.makedirs("./dir2/subdir2")
```

运行此操作的结果是

```
└-- dir1
    ├── subdir1
    └── subdir2
└-- dir2
    ├── subdir1
    └── subdir2
```

dir2 仅在第一次需要创建 subdir1 时被创建。

如果我们改用 `os.mkdir`，会抛出异常，因为 dir2 还不存在。

```
os.mkdir("./dir2/subdir1")
OSError: [Errno 2] 没有这样的文件或目录: './dir2/subdir1'
```

如果目标目录已经存在，`os.makedirs` 会报错。如果我们再次运行它：

```
OSError: [Errno 17] 文件已存在: './dir2/subdir1'
```

不过，这可以通过捕获异常并检查目录是否已创建来轻松解决。

```
try:
    os.makedirs("./dir2/subdir1")
except OSError:
    if not os.path.isdir("./dir2/subdir1"):
        raise

try:
    os.makedirs("./dir2/subdir2")
except OSError:
```

# Chapter 51: The os Module

## Parameter

Path	A path to a file. The path separator may be determined by <code>os.path.sep</code> .
Mode	The desired permission, in octal (e.g. <code>0700</code> )

This module provides a portable way of using operating system dependent functionality.

## Section 51.1: makedirs - recursive directory creation

Given a local directory with the following contents:

```
└-- dir1
    ├── subdir1
    └── subdir2
```

We want to create the same `subdir1`, `subdir2` under a new directory `dir2`, which does not exist yet.

```
import os
```

```
os.makedirs("./dir2/subdir1")
os.makedirs("./dir2/subdir2")
```

Running this results in

```
└-- dir1
    ├── subdir1
    └── subdir2
└-- dir2
    ├── subdir1
    └── subdir2
```

`dir2` 仅在第一次需要创建 `subdir1` 时被创建。

If we had used `os.mkdir` instead, we would have had an exception because `dir2` would not have existed yet.

```
os.mkdir("./dir2/subdir1")
OSError: [Errno 2] No such file or directory: './dir2/subdir1'
```

`os.makedirs` 不会喜欢如果目标目录已经存在。如果我们再次运行它：

```
OSError: [Errno 17] File exists: './dir2/subdir1'
```

然而，这可以通过捕获异常并检查目录是否已创建来轻松解决。

```
try:
    os.makedirs("./dir2/subdir1")
except OSError:
    if not os.path.isdir("./dir2/subdir1"):
        raise

try:
    os.makedirs("./dir2/subdir2")
except OSError:
```

```
if not os.path.isdir("./dir2/subdir2"):  
    raise
```

## 第51.2节：创建目录

```
os.mkdir('newdir')
```

如果需要指定权限，可以使用可选的mode参数：

```
os.mkdir('newdir', mode=0700)
```

## 第51.3节：获取当前目录

使用os.getcwd()函数：

```
print(os.getcwd())
```

## 第51.4节：确定操作系统名称

os模块提供了一个接口，用于确定代码当前运行的操作系统类型。

```
os.name
```

在Python 3中，这可以返回以下之一：

- posix
- nt
- ce
- java

可以从sys.platform获取更详细的信息

## 第51.5节：删除目录

删除路径为path的目录：

```
os.rmdir(path)
```

你不应该使用os.remove()来删除目录。该函数用于文件，若用于目录将导致OSError

## 第51.6节：跟随符号链接（POSIX）

有时你需要确定符号链接的目标。os.readlink可以做到这一点：

```
print(os.readlink(path_to_symlink))
```

## 第51.7节：更改文件权限

```
os.chmod(path, mode)
```

其中mode是所需的权限，采用八进制表示。

```
if not os.path.isdir("./dir2/subdir2"):  
    raise
```

## Section 51.2: Create a directory

```
os.mkdir('newdir')
```

If you need to specify permissions, you can use the optional mode argument:

```
os.mkdir('newdir', mode=0700)
```

## Section 51.3: Get current directory

Use the os.getcwd() function:

```
print(os.getcwd())
```

## Section 51.4: Determine the name of the operating system

The os module provides an interface to determine what type of operating system the code is currently running on.

```
os.name
```

This can return one of the following in Python 3:

- posix
- nt
- ce
- java

More detailed information can be retrieved from [sys.platform](#)

## Section 51.5: Remove a directory

Remove the directory at path:

```
os.rmdir(path)
```

You should not use os.remove() to remove a directory. That function is for files and using it on directories will result in an OSError

## Section 51.6: Follow a symlink (POSIX)

Sometimes you need to determine the target of a symlink. os.readlink will do this:

```
print(os.readlink(path_to_symlink))
```

## Section 51.7: Change permissions on a file

```
os.chmod(path, mode)
```

where mode is the desired permission, in octal.

## 第52章：locale模块

### 第52.1节：使用locale模块格式化美元货币

```
import locale

locale.setlocale(locale.LC_ALL, '')
Out[2]: 'English_United States.1252'

locale.currency(762559748.49)
Out[3]: '$762559748.49'

locale.currency(762559748.49, grouping=True)
Out[4]: '$762,559,748.49'
```

## Chapter 52: The locale Module

### Section 52.1: Currency Formatting US Dollars Using the locale Module

```
import locale

locale.setlocale(locale.LC_ALL, '')
Out[2]: 'English_United States.1252'

locale.currency(762559748.49)
Out[3]: '$762559748.49'

locale.currency(762559748.49, grouping=True)
Out[4]: '$762,559,748.49'
```

# 第53章：Itertools模块

## 第53.1节：Itertools模块中的Combinations方法

itertools.combinations 将返回一个列表的k组合序列的生成器。

换句话说：它将返回输入列表所有可能的k元组合的元组生成器。

例如：

如果你有一个列表：

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 2))
print b
```

输出：

```
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

上述输出是一个生成器被转换成的元组列表，包含输入列表a的所有可能的成对组合

你也可以找到所有的3组合：

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 3))
print b
```

输出：

```
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4),
 (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5),
 (2, 4, 5), (3, 4, 5)]
```

## 第53.2节：itertools.dropwhile

itertools.dropwhile 使你能够在条件首次变为False后，从序列中获取元素。

```
def is_even(x):
    return x % 2 == 0

lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.dropwhile(is_even, lst))

print(result)
```

这将输出[13, 14, 22, 23, 44]。

(这个例子与使用`akewhile`的例子相同，但使用了`dropwhile`。)

注意，第一个不满足谓词（即返回布尔值的函数）`is_even`的数字是13。  
在此之前的所有元素都被丢弃了。

# Chapter 53: Itertools Module

## Section 53.1: Combinations method in Itertools Module

`itertools.combinations` will return a generator of the  $k$ -combination sequence of a list.

**In other words:** It will return a generator of tuples of all the possible  $k$ -wise combinations of the input list.

**For Example:**

If you have a list:

```
a = [1, 2, 3, 4, 5]
b = list(itertools.combinations(a, 2))
print b
```

Output:

```
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

The above output is a generator converted to a list of tuples of all the possible pair-wise combinations of the input list a

**You can also find all the 3-combinations:**

```
a = [1, 2, 3, 4, 5]
b = list(itertools.combinations(a, 3))
print b
```

Output:

```
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4),
 (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5),
 (2, 4, 5), (3, 4, 5)]
```

## Section 53.2: itertools.dropwhile

`itertools.dropwhile` enables you to take items from a sequence after a condition first becomes `False`.

```
def is_even(x):
    return x % 2 == 0
```

```
lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.dropwhile(is_even, lst))

print(result)
```

This outputs [13, 14, 22, 23, 44].

(This example is same as the example for `takewhile` but using `dropwhile`.)

Note that, the first number that violates the predicate (i.e.: the function returning a Boolean value) `is_even` is, 13.  
All the elements before that, are discarded.

dropwhile产生的输出与下面代码生成的输出类似。

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

由 takewhile 和 dropwhile 产生的结果连接起来会生成原始的可迭代对象。

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

## 第53.3节：将两个迭代器压缩（zip）直到两者都耗尽

类似于内置函数 zip()，itertools.zip\_longest 会继续迭代，直到两个可迭代对象中较短的那个结束之后。

```
from itertools import zip_longest
a = [i for i in range(5)] # 长度为5
b = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # 长度为7
for i in zip_longest(a, b):
    x, y = i # 注意 zip_longest 返回的是元组形式的值
    print(x, y)
```

可以传入一个可选的 fillvalue 参数（默认为 ''），用法如下：

```
for i in zip_longest(a, b, fillvalue='Hogwash!'):
    x, y = i # 注意 zip_longest 返回的是元组形式的值
    print(x, y)
```

在 Python 2.6 和 2.7 中，该函数名为 itertools.izip\_longest。

## 第53.4节：对生成器进行切片

Itertools 中的 "islice" 允许你对生成器进行切片：

```
results = fetch_paged_results() # 返回一个生成器
limit = 20 # 只想要前20个结果
for data in itertools.islice(results, limit):
    print(data)
```

通常你不能对生成器进行切片：

```
def gen():
    n = 0
    当 n < 20时：
        n += 1
        产量 n

for part in gen()[:3]:
    打印(部分)
```

将给出

The output produced by dropwhile is similar to the output generated from the code below.

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

The concatenation of results produced by takewhile and dropwhile produces the original iterable.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

## Section 53.3: Zipping two iterators until they are both exhausted

Similar to the built-in function zip(), `itertools.zip_longest` will continue iterating beyond the end of the shorter of two iterables.

```
from itertools import zip_longest
a = [i for i in range(5)] # Length is 5
b = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # Length is 7
for i in zip_longest(a, b):
    x, y = i # Note that zip longest returns the values as a tuple
    print(x, y)
```

An optional fillvalue argument can be passed (defaults to '') like so:

```
for i in zip_longest(a, b, fillvalue='Hogwash!'):
    x, y = i # Note that zip longest returns the values as a tuple
    print(x, y)
```

In Python 2.6 and 2.7, this function is called `itertools.izip_longest`.

## Section 53.4: Take a slice of a generator

Itertools "islice" allows you to slice a generator:

```
results = fetch_paged_results() # returns a generator
limit = 20 # Only want the first 20 results
for data in itertools.islice(results, limit):
    print(data)
```

Normally you cannot slice a generator:

```
def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in gen()[:3]:
    print(part)
```

Will give

```
回溯（最近一次调用最后）：  
文件 "gen.py", 第 6 行, 在 <module>  
  for part in gen()[:3]:  
类型错误: 'generator' 对象不可下标访问
```

不过，这样是可行的：

```
import itertools  
  
def gen():  
    n = 0  
    当 n < 20时：  
        n += 1  
        产量 n  
  
for part in itertools.islice(gen(), 3):  
    打印(部分)
```

注意，像普通切片一样，你也可以使用 start、stop 和 step 参数：

```
itertools.islice(iterator, 1, 30, 3)
```

## 第53.5节：使用函数对可迭代对象中的项目进行分组

从需要分组的可迭代对象开始

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

生成分组生成器，按每个元组的第二个元素分组：

```
def testGroupBy(lst):  
    groups = itertools.groupby(lst, key=lambda x: x[1])  
    for key, group in groups:  
        print(key, list(group))  
  
testGroupBy(lst)  
  
# 5 [(‘a’, 5, 6)]  
# 2 [(‘b’, 2, 4), (‘a’, 2, 5), (‘c’, 2, 6)]
```

只有连续元素的组会被分组。调用 groupby 之前，可能需要按相同的键进行排序  
例如，（最后一个元素被更改）

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 5, 6)]  
testGroupBy(lst)  
  
# 5 [(‘a’, 5, 6)]  
# 2 [(‘b’, 2, 4), (‘a’, 2, 5)]  
# 5 [(‘c’, 5, 6)]
```

groupby 返回的分组是一个迭代器，在下一次迭代之前该迭代器将失效。例如，如果你想按键排序分组，下面的代码将无法正常工作。下面的分组 5 是空的，因为当获取分组 2 时，它使分组 5 失效了。

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]  
groups = itertools.groupby(lst, key=lambda x: x[1])  
for key, group in sorted(groups):
```

```
Traceback (most recent call last):  
  File "gen.py", line 6, in <module>  
    for part in gen()[:3]:  
TypeError: 'generator' object is not subscriptable
```

However, this works:

```
import itertools  
  
def gen():  
    n = 0  
    while n < 20:  
        n += 1  
        yield n  
  
for part in itertools.islice(gen(), 3):  
    print(part)
```

Note that like a regular slice, you can also use start, stop and step arguments:

```
itertools.islice(iterator, 1, 30, 3)
```

## Section 53.5: Grouping items from an iterable object using a function

Start with an iterable which needs to be grouped

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

Generate the grouped generator, grouping by the second element in each tuple:

```
def testGroupBy(lst):  
    groups = itertools.groupby(lst, key=lambda x: x[1])  
    for key, group in groups:  
        print(key, list(group))  
  
testGroupBy(lst)  
  
# 5 [(‘a’, 5, 6)]  
# 2 [(‘b’, 2, 4), (‘a’, 2, 5), (‘c’, 2, 6)]
```

Only groups of consecutive elements are grouped. You may need to sort by the same key before calling groupby  
For E.g, (Last element is changed)

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 5, 6)]  
testGroupBy(lst)  
  
# 5 [(‘a’, 5, 6)]  
# 2 [(‘b’, 2, 4), (‘a’, 2, 5)]  
# 5 [(‘c’, 5, 6)]
```

The group returned by groupby is an iterator that will be invalid before next iteration. E.g the following will not work if you want the groups to be sorted by key. Group 5 is empty below because when group 2 is fetched it invalidates 5

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]  
groups = itertools.groupby(lst, key=lambda x: x[1])  
for key, group in sorted(groups):
```

```
print(key, list(group))
```

```
# 2 [('c', 2, 6)]  
# 5 []
```

要正确进行排序，应先从迭代器创建一个列表，然后再排序

```
groups = itertools.groupby(lst, key=lambda x: x[1])  
for key, group in sorted((key, list(group)) for key, group in groups):  
    print(key, list(group))  
  
# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]  
# 5 [('a', 5, 6)]
```

## 第 53.6 节 : itertools.takewhile

`itertools.takewhile` 允许你从序列中取元素，直到条件首次变为False。

```
def is_even(x):  
    return x % 2 == 0  
  
lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]  
result = list(itertools.takewhile(is_even, lst))  
  
print(result)
```

这输出为[0, 2, 4, 12, 18]。

请注意，第一个不满足谓词（即返回布尔值的函数）`is_even`的数字是13。一旦`takewhile`遇到一个对给定谓词返回False的值，它就会停止。

`takewhile`产生的输出与下面代码生成的输出类似。

```
def takewhile(predicate, iterable):  
    for x in iterable:  
        if predicate(x):  
            yield x  
        else:  
            break
```

注意：`takewhile`和`dropwhile`产生的结果连接起来就是原始的可迭代对象。

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

## 第53.7节 : itertools.permutations

`itertools.permutations`返回一个生成器，生成可迭代对象中元素的连续长度为r的排列。

```
a = [1,2,3]  
list(itertools.permutations(a))  
# [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]  
  
list(itertools.permutations(a, 2))  
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

```
print(key, list(group))
```

```
# 2 [('c', 2, 6)]  
# 5 []
```

To correctly do sorting, create a list from the iterator before sorting

```
groups = itertools.groupby(lst, key=lambda x: x[1])  
for key, group in sorted((key, list(group)) for key, group in groups):  
    print(key, list(group))  
  
# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]  
# 5 [('a', 5, 6)]
```

## Section 53.6: `itertools.takewhile`

`itertools.takewhile` enables you to take items from a sequence until a condition first becomes `False`.

```
def is_even(x):  
    return x % 2 == 0  
  
lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]  
result = list(itertools.takewhile(is_even, lst))  
  
print(result)
```

This outputs [0, 2, 4, 12, 18].

Note that, the first number that violates the predicate (i.e.: the function returning a Boolean value) `is_even` is, 13. Once `takewhile` encounters a value that produces `False` for the given predicate, it breaks out.

The **output produced** by `takewhile` is similar to the output generated from the code below.

```
def takewhile(predicate, iterable):  
    for x in iterable:  
        if predicate(x):  
            yield x  
        else:  
            break
```

**Note:** The concatenation of results produced by `takewhile` and `dropwhile` produces the original iterable.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

## Section 53.7: `itertools.permutations`

`itertools.permutations` returns a generator with successive r-length permutations of elements in the iterable.

```
a = [1,2,3]  
list(itertools.permutations(a))  
# [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]  
  
list(itertools.permutations(a, 2))  
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

如果列表 a 有重复元素，生成的排列也会有重复元素，你可以使用 `set` 来获取唯一的排列：

```
a = [1,2,1]
list(itertools.permutations(a))
# [(1, 2, 1), (1, 1, 2), (2, 1, 1), (2, 1, 1), (1, 1, 2), (1, 2, 1)]
set(itertools.permutations(a))
# {(1, 1, 2), (1, 2, 1), (2, 1, 1)}
```

## 第53.8节：itertools.repeat

重复某个元素n次：

```
>>> import itertools
>>> for i in itertools.repeat('over-and-over', 3):
...     print(i)
over-and-over
over-and-over
over-and-over
```

## 第53.9节：获取可迭代对象中数字的累积和

Python 3.x 版本 ≥ 3.2

`accumulate` 生成数字的累积和（或积）。

```
>>> import itertools as it
>>> import operator

>>> list(it.accumulate([1,2,3,4,5]))
[1, 3, 6, 10, 15]

>>> list(it.accumulate([1,2,3,4,5], func=operator.mul))
[1, 2, 6, 24, 120]
```

## 第53.10节：循环迭代器中的元素

`cycle` 是一个无限迭代器。

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

因此，使用此方法时请注意设置边界，以避免无限循环。示例：

```
>>> # 在固定范围内迭代 cycle 中的每个元素
>>> cycle_iterator = it.cycle('abc123')
>>> [next(cycle_iterator) for i in range(0, 10)]
['a', 'b', 'c', '1', '2', '3', 'a', 'b', 'c', '1']
```

## 第53.11节：itertools.product

此函数允许你迭代多个可迭代对象的笛卡尔积。

if the list a has duplicate elements, the resulting permutations will have duplicate elements, you can use `set` to get unique permutations:

```
a = [1,2,1]
list(itertools.permutations(a))
# [(1, 2, 1), (1, 1, 2), (2, 1, 1), (2, 1, 1), (1, 1, 2), (1, 2, 1)]
set(itertools.permutations(a))
# {(1, 1, 2), (1, 2, 1), (2, 1, 1)}
```

## Section 53.8: itertools.repeat

Repeat something n times:

```
>>> import itertools
>>> for i in itertools.repeat('over-and-over', 3):
...     print(i)
over-and-over
over-and-over
over-and-over
```

## Section 53.9: Get an accumulated sum of numbers in an iterable

Python 3.x Version ≥ 3.2

`accumulate` yields a cumulative sum (or product) of numbers.

```
>>> import itertools as it
>>> import operator

>>> list(it.accumulate([1,2,3,4,5]))
[1, 3, 6, 10, 15]

>>> list(it.accumulate([1,2,3,4,5], func=operator.mul))
[1, 2, 6, 24, 120]
```

## Section 53.10: Cycle through elements in an iterator

`cycle` is an infinite iterator.

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

Therefore, take care to give boundaries when using this to avoid an infinite loop. Example:

```
>>> # Iterate over each element in cycle for a fixed range
>>> cycle_iterator = it.cycle('abc123')
>>> [next(cycle_iterator) for i in range(0, 10)]
['a', 'b', 'c', '1', '2', '3', 'a', 'b', 'c', '1']
```

## Section 53.11: itertools.product

This function lets you iterate over the Cartesian product of a list of iterables.

例如,

```
for x, y in itertools.product(xrange(10), xrange(10)):  
    print x, y
```

等同于

```
for x in xrange(10):  
    for y in xrange(10):  
        print x, y
```

像所有接受可变数量参数的Python函数一样，我们可以使用\*操作符将列表传递给itertools.product以进行拆包。

因此,

```
its = [xrange(10)] * 2  
for x,y in itertools.product(*its):  
    print x, y
```

产生与前面两个示例相同的结果。

```
>>> from itertools import product  
>>> a=[1,2,3,4]  
>>> b=['a','b','c']  
>>> product(a,b)  
<itertools.product object at 0x000000002712F78>  
>>> for i in product(a,b):  
...     打印 i  
...  
(1, 'a')  
(1, 'b')  
(1, 'c')  
(2, 'a')  
(2, 'b')  
(2, 'c')  
(3, 'a')  
(3, 'b')  
(3, 'c')  
(4, 'a')  
(4, 'b')  
(4, 'c')
```

## 第53.12节：itertools.count

简介：

这个简单的函数生成无限的数字序列。例如...

```
对于 number 在 itertools.count()中：  
    如果 number > 20 :  
        break  
    打印(number)
```

注意我们必须中断，否则它会无限打印！

输出：

For example,

```
for x, y in itertools.product(xrange(10), xrange(10)):  
    print x, y
```

is equivalent to

```
for x in xrange(10):  
    for y in xrange(10):  
        print x, y
```

Like all python functions that accept a variable number of arguments, we can pass a list to itertools.product for unpacking, with the \* operator.

Thus,

```
its = [xrange(10)] * 2  
for x,y in itertools.product(*its):  
    print x, y
```

produces the same results as both of the previous examples.

```
>>> from itertools import product  
>>> a=[1,2,3,4]  
>>> b=['a','b','c']  
>>> product(a,b)  
<itertools.product object at 0x000000002712F78>  
>>> for i in product(a,b):  
...     print i  
...  
(1, 'a')  
(1, 'b')  
(1, 'c')  
(2, 'a')  
(2, 'b')  
(2, 'c')  
(3, 'a')  
(3, 'b')  
(3, 'c')  
(4, 'a')  
(4, 'b')  
(4, 'c')
```

## Section 53.12: itertools.count

**Introduction:**

This simple function generates infinite series of numbers. For example...

```
for number in itertools.count():  
    if number > 20:  
        break  
    print(number)
```

Note that we must break or it prints forever!

Output:

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
参数:  
c  
o  
u  
n  
t  
(
```

接受两个参数，

和 `:`

```
for number in itertools.count(start=10, step=4):  
    print(number)  
    if number > 20:  
        break
```

输出：

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

#### Arguments:

`count()` takes two arguments, `start` and `step`:

```
for number in itertools.count(start=10, step=4):  
    print(number)  
    if number > 20:  
        break
```

Output:

```
10  
14  
18  
22
```

## Section 53.13: Chaining multiple iterators together

Use `itertools.chain` to create a single generator which will yield the values from several generators in sequence.

```
from itertools import chain  
a = (x for x in ['1', '2', '3', '4'])  
b = (x for x in ['x', 'y', 'z'])  
''.join(chain(a, b))
```

Results in:

```
'1 2 3 4 x y z'
```

As an alternate constructor, you can use the classmethod `chain.from_iterable` which takes as its single parameter an iterable of iterables. To get the same result as above:

```
''.join(chain.from_iterable([a, b]))
```

While `chain` can take an arbitrary number of arguments, `chain.from_iterable` is the only way to chain an *infinite* number of iterables.

# 第54章：Asyncio模块

## 第54.1节：协程和委托语法

在Python 3.5+发布之前，asyncio模块使用生成器来模拟异步调用，因此其语法与当前Python 3.5版本不同。

Python 3.x 版本 ≥ 3.5

Python 3.5引入了async和await关键字。注意await func()调用周围没有括号。

```
import asyncio

async def main():
    print(await func())

async def func():
    # 执行耗时操作...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x 版本 ≥ 3.3 版本 < 3.5

在 Python 3.5 之前，使用 @asyncio.coroutine 装饰器来定义协程。yield from 表达式用于生成器委托。注意 yield from func() 周围的括号。

```
import asyncio

@asyncio.coroutine
def main():
    print((yield from func()))

@asyncio.coroutine
def func():
    # 执行耗时操作...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x 版本 ≥ 3.5

下面是一个示例，展示了如何异步运行两个函数：

```
import asyncio

async def cor1():
    print("cor1 开始")
    for i in range(10):
        await asyncio.sleep(1.5)
        print("cor1", i)

async def cor2():
    print("cor2 开始")
    for i in range(15):
        await asyncio.sleep(1)
```

# Chapter 54: Asyncio Module

## Section 54.1: Coroutine and Delegation Syntax

Before Python 3.5+ was released, the `asyncio` module used generators to mimic asynchronous calls and thus had a different syntax than the current Python 3.5 release.

Python 3.x Version ≥ 3.5

Python 3.5 introduced the `async` and `await` keywords. Note the lack of parentheses around the `await func()` call.

```
import asyncio

async def main():
    print(await func())

async def func():
    # Do time intensive stuff...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x Version ≥ 3.3 Version < 3.5

Before Python 3.5, the `@asyncio.coroutine` decorator was used to define a coroutine. The `yield from` expression was used for generator delegation. Note the parentheses around the `yield from func()`.

```
import asyncio

@asyncio.coroutine
def main():
    print((yield from func()))

@asyncio.coroutine
def func():
    # Do time intensive stuff...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Python 3.x Version ≥ 3.5

Here is an example that shows how two functions can be run asynchronously:

```
import asyncio

async def cor1():
    print("cor1 start")
    for i in range(10):
        await asyncio.sleep(1.5)
        print("cor1", i)

async def cor2():
    print("cor2 start")
    for i in range(15):
        await asyncio.sleep(1)
```

```
print("cor2", i)

loop = asyncio.get_event_loop()
cors = asyncio.wait([cor1(), cor2()])
loop.run_until_complete(cors)
```

## 第54.2节：异步执行器

注意：使用Python 3.5及以上版本的async/await语法

asyncio支持使用concurrent.futures中提供的Executor对象来异步调度任务。事件循环具有 run\_in\_executor()函数，该函数接受一个Executor对象、一个Callable以及该Callable的参数。

为Executor调度任务

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # 执行耗时操作...
    return a + b

async def main(loop):
    executor = ThreadPoolExecutor()
    result = await loop.run_in_executor(executor, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

每个事件循环也有一个“默认”的Executor槽，可以分配给一个Executor。要分配一个Executor并从循环中调度任务，可以使用set\_default\_executor()方法。

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # 执行耗时操作...
    return a + b

async def main(loop):
    # 注意：将 `None` 作为第一个参数表示使用 `default` 执行器 (Executor)。
    result = await loop.run_in_executor(None, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.set_default_executor(ThreadPoolExecutor())
    loop.run_until_complete(main(loop))
```

在 concurrent.futures 中有两种主要的 Executor 类型，分别是 ThreadPoolExecutor 和 ProcessPoolExecutor。ThreadPoolExecutor 包含一个线程池，可以通过构造函数手动设置线程数，否则默认是机器核心数乘以 5。ThreadPoolExecutor 使用线程池来执行分配给它的任务，通常更适合 CPU 密集型操作，而非 I/O 密集型操作。与此相对的是 ProcessPoolExecutor，它为每个分配的任务生成一个新的进程。ProcessPoolExecutor 只能接受可序列化 (picklable) 的任务和参数。

```
print("cor2", i)

loop = asyncio.get_event_loop()
cors = asyncio.wait([cor1(), cor2()])
loop.run_until_complete(cors)
```

## Section 54.2: Asynchronous Executors

Note: Uses the Python 3.5+ async/await syntax

asyncio supports the use of Executor objects found in concurrent.futures for scheduling tasks asynchronously. Event loops have the function run\_in\_executor() which takes an Executor object, a Callable, and the Callable's parameters.

Scheduling a task for an Executor

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b

async def main(loop):
    executor = ThreadPoolExecutor()
    result = await loop.run_in_executor(executor, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

Each event loop also has a "default" Executor slot that can be assigned to an Executor. To assign an Executor and schedule tasks from the loop you use the set\_default\_executor() method.

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Do time intensive stuff...
    return a + b

async def main(loop):
    # NOTE: Using `None` as the first parameter designates the `default` Executor.
    result = await loop.run_in_executor(None, func, "Hello,", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.set_default_executor(ThreadPoolExecutor())
    loop.run_until_complete(main(loop))
```

There are two main types of Executor in concurrent.futures, the ThreadPoolExecutor and the ProcessPoolExecutor. The ThreadPoolExecutor contains a pool of threads which can either be manually set to a specific number of threads through the constructor or defaults to the number of cores on the machine times 5. The ThreadPoolExecutor uses the pool of threads to execute tasks assigned to it and is generally better at CPU-bound operations rather than I/O bound operations. Contrast that to the ProcessPoolExecutor which spawns a new process for each task assigned to it. The ProcessPoolExecutor can only take tasks and parameters that are

不可序列化的任务最常见的是对象的方法。如果必须将对象的方法作为任务调度到 Executor 中，必须使用 ThreadPoolExecutor。

## 第54.3节：使用 UVLoop

uvloop 是基于 libuv (nodejs 使用的库) 实现的 asyncio.AbstractEventLoop。它兼容 99% 的 asyncio 功能，并且比传统的 asyncio.EventLoop 快得多。uvloop 目前不支持 Windows，安装命令为 pip install uvloop。

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # 在这里执行你的操作 ...
```

也可以通过将 EventLoopPolicy 设置为 uvloop 中的策略来更改事件循环工厂。

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()
```

## 第54.4节：同步原语：事件

### 概念

使用 Event 来同步多个协程的调度。

简单来说，事件就像赛跑中的发令枪：它让选手们从起跑块上出发。

### 示例

```
import asyncio

# 事件触发函数
def trigger(event):
    print('事件已设置')
event.set() # 唤醒等待的协程

# 事件消费者
async def consumer_a(event):
    consumer_name = '消费者 A'
    print('{} 等待中'.format(consumer_name))
    await event.wait()
    print('{} 已触发'.format(consumer_name))

async def consumer_b(event):
    consumer_name = '消费者 B'
    print('{} 等待中'.format(consumer_name))
    await event.wait()
    print('{} 已触发'.format(consumer_name))

# 事件
event = asyncio.Event()

# 将协程包装成一个 future
```

pickable。The most common non-pickable tasks are the methods of objects. If you must schedule an object's method as a task in an Executor you must use a ThreadPoolExecutor.

## Section 54.3: Using UVLoop

uvloop 是一个基于 libuv (Used by nodejs) 实现的 asyncio.AbstractEventLoop。它兼容 99% 的 asyncio 功能，并且比传统的 asyncio.EventLoop 快得多。uvloop 目前不支持 Windows，安装命令为 pip install uvloop。

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # Do your stuff here ...
```

One can also change the event loop factory by setting the EventLoopPolicy to the one in uvloop.

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()
```

## Section 54.4: Synchronization Primitive: Event

### Concept

Use an Event to **synchronize the scheduling of multiple coroutines**.

Put simply, an event is like the gun shot at a running race: it lets the runners off the starting blocks.

### Example

```
import asyncio

# event trigger function
def trigger(event):
    print('EVENT SET')
    event.set() # wake up coroutines waiting

# event consumers
async def consumer_a(event):
    consumer_name = 'Consumer A'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

async def consumer_b(event):
    consumer_name = 'Consumer B'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

# event
event = asyncio.Event()

# wrap coroutines in one future
```

```

main_future = asyncio.wait([consumer_a(event),
                           consumer_b(event)])

# 事件循环
event_loop = asyncio.get_event_loop()
event_loop.call_later(0.1, functools.partial(trigger, event)) # 0.1秒后触发事件

# 完成 main_future
done, pending = event_loop.run_until_complete(main_future)

```

输出：

```

消费者B等待中
消费者A等待中
事件已设置
消费者B已触发
消费者A已触发

```

## 第54.5节：一个简单的Websocket

这里我们使用asyncio制作一个简单的回声websocket。我们定义了用于连接服务器和发送/接收消息的协程。websocket的通信运行在一个main协程中，该协程由事件循环运行。此示例修改自一个先前的帖子。

```

import asyncio
import aiohttp

session = aiohttp.ClientSession() # 处理上下文管理器
class EchoWebsocket:

    async def connect(self):
        self.websocket = await session.ws_connect("wss://echo.websocket.org")

    async def send(self, message):
        self.websocket.send_str(message)

    async def receive(self):
        result = (await self.websocket.receive())
        return result.data

    async def main():
        echo = EchoWebsocket()
        await echo.connect()
        await echo.send("Hello World!")
        print(await echo.receive()) # "Hello World!"

if __name__ == '__main__':
    # The main loop
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())

```

## 第54.6节：关于asyncio的常见误解

关于asyncio最常见的误解可能是它允许你并行运行任何任务——绕过GIL（全局解释器锁），从而并行执行阻塞任务（在不同线程上）。实际上并不可以！

```

main_future = asyncio.wait([consumer_a(event),
                           consumer_b(event)])

# event loop
event_loop = asyncio.get_event_loop()
event_loop.call_later(0.1, functools.partial(trigger, event)) # trigger event in 0.1 sec

# complete main_future
done, pending = event_loop.run_until_complete(main_future)

```

Output:

```

Consumer B waiting
Consumer A waiting
EVENT SET
Consumer B triggered
Consumer A triggered

```

## Section 54.5: A Simple Websocket

Here we make a simple echo websocket using asyncio. We define coroutines for connecting to a server and sending/receiving messages. The communications of the websocket are run in a main coroutine, which is run by an event loop. This example is modified from a [prior post](#).

```

import asyncio
import aiohttp

session = aiohttp.ClientSession() # handles the context manager
class EchoWebsocket:

    async def connect(self):
        self.websocket = await session.ws_connect("wss://echo.websocket.org")

    async def send(self, message):
        self.websocket.send_str(message)

    async def receive(self):
        result = (await self.websocket.receive())
        return result.data

    async def main():
        echo = EchoWebsocket()
        await echo.connect()
        await echo.send("Hello World!")
        print(await echo.receive()) # "Hello World!"

if __name__ == '__main__':
    # The main loop
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())

```

## Section 54.6: Common Misconception about asyncio

probably the most common misconception about asyncio is that it lets you run any task in parallel - sidestepping the GIL (global interpreter lock) and therefore execute blocking jobs in parallel (on separate threads). it does **not!**

`asyncio` (以及与 `asyncio` 协作构建的库) 基于协程：协程是（协作式地）将控制流交给调用函数的函数。请注意上面示例中的 `asyncio.sleep`。这是一个非阻塞协程的例子，它在“后台”等待，并在调用时（使用 `await`）将控制流交给调用函数。`time.sleep` 是一个阻塞函数的例子。程序的执行流程会在此处停止，只有在 `time.sleep` 完成后才会返回。

一个真实的例子是 `requests` 库，目前它只包含阻塞函数。如果你在 `asyncio` 中调用它的任何函数，是不会有并发的。另一方面，`aiohttp` 是专门为 `asyncio` 设计的。它的协程可以并发运行。

- 如果你需要并行运行的长时间 CPU 密集型任务，`asyncio` 并不适合你。对此你需要使用线程（`threads`）或多进程（`multiprocessing`）。
- 如果你有 IO 密集型任务运行，你可以使用 `asyncio` 来并发运行它们。

`asyncio` (and libraries that are built to collaborate with `asyncio`) build on coroutines: functions that (collaboratively) yield the control flow back to the calling function. note `asyncio.sleep` in the examples above. this is an example of a non-blocking coroutine that waits 'in the background' and gives the control flow back to the calling function (when called with `await`). `time.sleep` is an example of a blocking function. the execution flow of the program will just stop there and only return after `time.sleep` has finished.

a real-live example is the `requests` library which consists (for the time being) on blocking functions only. there is no concurrency if you call any of its functions within `asyncio`. `aiohttp` on the other hand was built with `asyncio` in mind. its coroutines will run concurrently.

- if you have long-running CPU-bound tasks you would like to run in parallel `asyncio` is **not** for you. for that you need `threads` or `multiprocessing`.
- if you have IO-bound jobs running, you *may* run them concurrently using `asyncio`.

# 视频：机器学习A-Z：动手Python数据科学

向两位数据科学专家学习如何用Python创建机器学习算法。包含代码模板。



- ✓ 精通Python上的机器学习
- ✓ 对多种机器学习模型有良好的直觉
- ✓ 做出准确的预测
- ✓ 进行强有力的分析
- ✓ 构建稳健的机器学习模型
- ✓ 为您的业务创造强大的附加价值
- ✓ 将机器学习用于个人目的
- ✓ 处理强化学习、自然语言处理和深度学习等特定主题
- ✓ 掌握降维等高级技术
- ✓ 了解针对每种问题应选择哪种机器学习模型
- ✓ 构建一支强大的机器学习模型军团，并知道如何组合它们来解决任何问题

今天观看 →

# VIDEO: Machine Learning A-Z: Hands-On Python In Data Science

Learn to create Machine Learning Algorithms in Python from two Data Science experts. Code templates included.



- ✓ Master Machine Learning on Python
- ✓ Have a great intuition of many Machine Learning models
- ✓ Make accurate predictions
- ✓ Make powerful analysis
- ✓ Make robust Machine Learning models
- ✓ Create strong added value to your business
- ✓ Use Machine Learning for personal purpose
- ✓ Handle specific topics like Reinforcement Learning, NLP and Deep Learning
- ✓ Handle advanced techniques like Dimensionality Reduction
- ✓ Know which Machine Learning model to choose for each type of problem
- ✓ Build an army of powerful Machine Learning models and know how to combine them to solve any problem

Watch Today →

# 第 55 章：随机模块

## 第 55.1 节：创建随机用户密码

为了创建随机用户密码，我们可以使用 `string` 模块中提供的符号。具体来说，`punctuation` 用于标点符号，`ascii_letters` 用于字母，`digits` 用于数字：

```
from string import punctuation, ascii_letters, digits
```

然后我们可以将所有这些符号组合成一个名为 `symbols` 的名称：

```
symbols = ascii_letters + digits + punctuation
```

删除其中任意一项即可创建元素更少的符号池。

之后，我们可以使用 `random.SystemRandom` 来生成密码。对于长度为 10 的密码：

```
secure_random = random.SystemRandom()
password = ''.join(secure_random.choice(symbols) for i in range(10))
print(password) # '^@g;J?M6e'
```

请注意，`random` 模块立即提供的其他例程—例如 `random.choice`、  
`random.randint` 等—不适合用于密码学目的。

这些例程在幕后使用的是 Mersenne Twister 伪随机数生成器 (PRNG)，无法满足 CSPRNG 的要求。因此，特别是你不应使用它们来生成计划使用的密码。应始终使用如上所示的 `SystemRandom` 实例。

Python 3.x 版本  $\geq 3.6$

从 Python 3.6 开始，提供了 `secrets` 模块，暴露了密码学安全的功能。

引用官方文档，生成“一个包含至少一个小写字母、至少一个大写字母和至少三个数字的十字符字母数字密码”，你可以：

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
        and any(c.isupper() for c in password)
        and sum(c.isdigit() for c in password) >= 3):
        break
```

## 第 55.2 节：创建密码学安全的随机数

默认情况下，Python 的 `random` 模块使用梅森旋转算法 (Mersenne Twister) PRNG 生成随机数，虽然适用于模拟等领域，但在更高安全要求的环境中无法满足安全需求。

为了创建密码学安全的伪随机数，可以使用 `SystemRandom`，它通过使用 `os.urandom`，能够作为密码学安全的伪随机数生成器 (CPRNG)。

# Chapter 55: Random module

## Section 55.1: Creating a random user password

In order to create a random user password we can use the symbols provided in the `string` module. Specifically punctuation for punctuation symbols, `ascii_letters` for letters and `digits` for digits:

```
from string import punctuation, ascii_letters, digits
```

We can then combine all these symbols in a name named `symbols`:

```
symbols = ascii_letters + digits + punctuation
```

Remove either of these to create a pool of symbols with fewer elements.

After this, we can use `random.SystemRandom` to generate a password. For a 10 length password:

```
secure_random = random.SystemRandom()
password = ''.join(secure_random.choice(symbols) for i in range(10))
print(password) # '^@g;J?M6e'
```

Note that other routines made immediately available by the `random` module — such as `random.choice`, `random.randint`, etc. — are *unsuitable for cryptographic purposes*.

Behind the curtains, these routines use the [Mersenne Twister PRNG](#), which does not satisfy the requirements of a [CSPRNG](#). Thus, in particular, you should not use any of them to generate passwords you plan to use. Always use an instance of `SystemRandom` as shown above.

Python 3.x Version  $\geq 3.6$

Starting from Python 3.6, the `secrets` module is available, which exposes cryptographically safe functionality.

Quoting the [official documentation](#), to generate “a ten-character alphanumeric password with at least one lowercase character, at least one uppercase character, and at least three digits,” you could:

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
        and any(c.isupper() for c in password)
        and sum(c.isdigit() for c in password) >= 3):
        break
```

## Section 55.2: Create cryptographically secure random numbers

By default the Python random module use the Mersenne Twister [PRNG](#) to generate random numbers, which, although suitable in domains like simulations, fails to meet security requirements in more demanding environments.

In order to create a cryptographically secure pseudorandom number, one can use `SystemRandom` which, by using `os.urandom`, is able to act as a Cryptographically secure pseudorandom number generator, [CPRNG](#).

最简单的使用方法是初始化SystemRandom类。其提供的方法与random模块导出的方法类似。

```
from random import SystemRandom
secure_rand_gen = SystemRandom()
```

为了创建一个范围在[0, 20]内的10个int的随机序列，可以简单调用randrange()：

```
print([secure_rand_gen.randrange(10) for i in range(10)])
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

要在给定范围内生成随机整数，可以使用 randint：

```
print(secure_rand_gen.randint(0, 20))
# 5
```

对于所有其他方法也是如此。接口完全相同，唯一的变化是底层的数字生成器。

你也可以直接使用 `os.urandom` 来获取密码学安全的随机字节。

## 第55.3节：随机数和序列：洗牌、选择和抽样

```
import random
shuffle()
```

你可以使用 `random.shuffle()` 来打乱/随机化一个 可变且可索引 的序列中的元素。例如一个 列表：

```
laughs = ["Hi", "Ho", "He"]

random.shuffle(laughs)      # 就地洗牌！不要写成：laughs = random.shuffle(laughs)

print(laughs)
# 输出: ["He", "Hi", "Ho"] # 输出可能会有所不同！
```

`choice()`

从任意序列中随机选择一个元素：

```
print(random.choice(laughs))
# 输出: He           # 输出可能会有所不同！
```

`sample()`

类似于 `choice`，它从任意序列中随机选择元素，但你可以指定选择多少个：

```
#          /--序列--/ --数量--
print(random.sample(laughs, 1)) # 选择一个元素
# 输出: ['Ho']                 # 输出可能会有所不同！
```

它不会选择相同的元素两次：

```
print(random.sample(laughs, 3)) # 从序列中选择3个随机元素。
# 输出: ['Ho', 'He', 'Hi']     # 输出可能有所不同！
```

The easiest way to use it simply involves initializing the `SystemRandom` class. The methods provided are similar to the ones exported by the `random` module.

```
from random import SystemRandom
secure_rand_gen = SystemRandom()
```

In order to create a random sequence of 10 `ints` in range [0, 20], one can simply call `randrange()`:

```
print([secure_rand_gen.randrange(10) for i in range(10)])
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

To create a random integer in a given range, one can use `randint`:

```
print(secure_rand_gen.randint(0, 20))
# 5
```

and, accordingly for all other methods. The interface is exactly the same, the only change is the underlying number generator.

You can also use `os.urandom` directly to obtain cryptographically secure random bytes.

## Section 55.3: Random and sequences: shuffle, choice and sample

```
import random
shuffle()
```

You can use `random.shuffle()` to mix up/randomize the items in a **mutable and indexable** sequence. For example a `list`:

```
laughs = ["Hi", "Ho", "He"]

random.shuffle(laughs)      # Shuffles in-place! Don't do: laughs = random.shuffle(laughs)

print(laughs)
# Out: ["He", "Hi", "Ho"] # Output may vary!
```

`choice()`

Takes a random element from an arbitrary **sequence**:

```
print(random.choice(laughs))
# Out: He           # Output may vary!
```

`sample()`

Like `choice` it takes random elements from an arbitrary **sequence** but you can specify how many:

```
#          /--sequence--/ --number--
print(random.sample(laughs, 1)) # Take one element
# Out: ['Ho']                 # Output may vary!
```

it will not take the same element twice:

```
print(random.sample(laughs, 3)) # Take 3 random element from the sequence.
# Out: ['Ho', 'He', 'Hi']      # Output may vary!
```

```
print(random.sample(laughs, 4)) # 从3个元素的序列中随机取4个元素。
```

ValueError: Sample larger than population

## 第55.4节：创建随机整数和浮点数：randint、randrange、random和uniform

```
import random
```

**randint()**

返回一个介于 x 和 y (含) 之间的随机整数：

```
random.randint(x, y)
```

例如，获取一个介于1 和8 之间的随机数：

```
random.randint(1, 8) # 输出：8
```

**randrange()**

random.randrange 的语法与 range 相同，与 random.randint 不同，最后一个值不包含在内：

```
random.randrange(100)      # 生成0到99之间的随机整数
```

```
random.randrange(20, 50)    # 生成20到49之间的随机整数
```

```
random.randrange(10, 20, 3) # 生成10到19之间步长为3的随机整数 (10, 13, 16和19)
```

```
print(random.sample(laughs, 4)) # Take 4 random element from the 3-item sequence.
```

ValueError: Sample larger than population

## Section 55.4: Creating random integers and floats: randint, randrange, random, and uniform

```
import random
```

**randint()**

Returns a random integer between x and y (inclusive):

```
random.randint(x, y)
```

For example getting a random number between 1 and 8:

```
random.randint(1, 8) # Out: 8
```

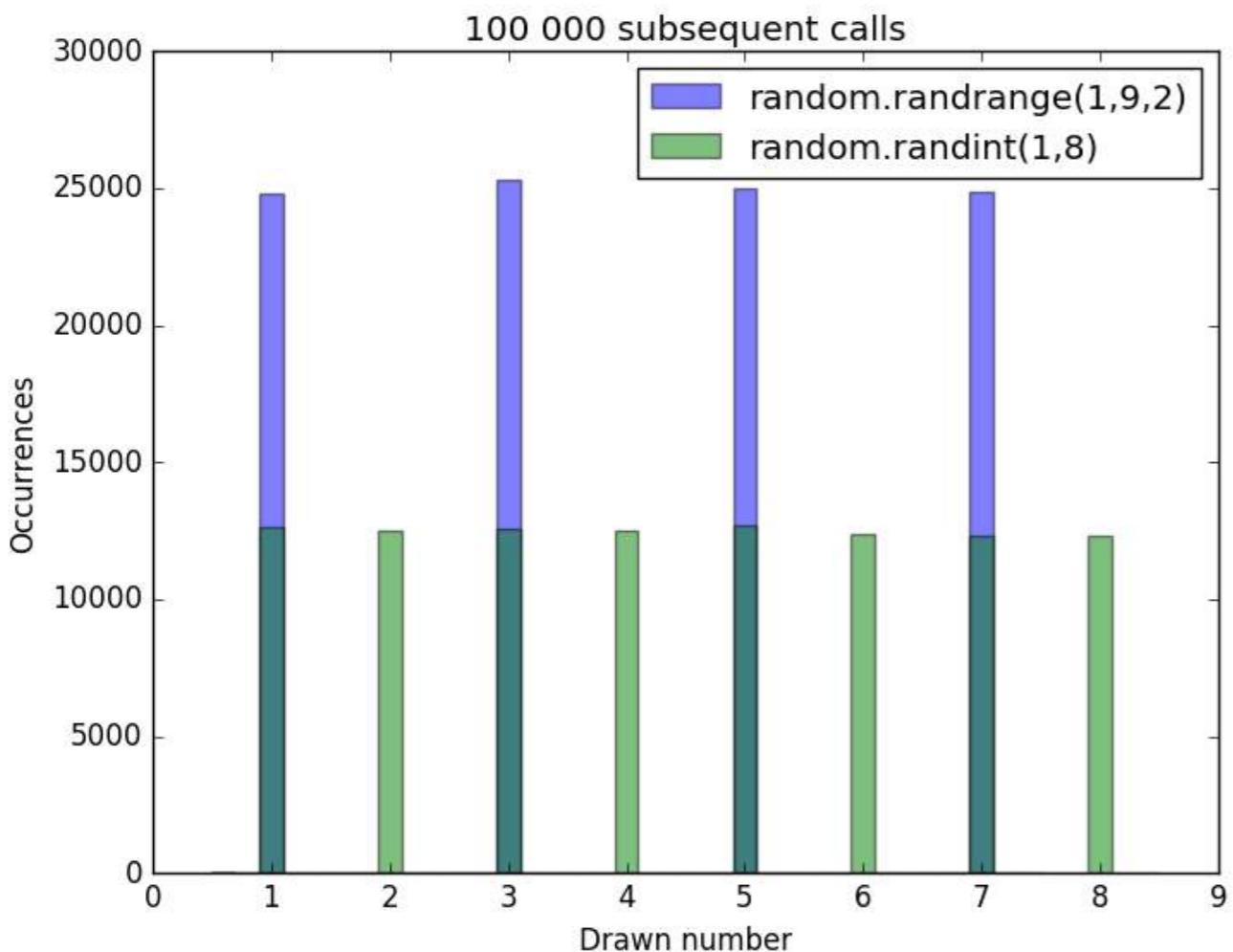
**randrange()**

random.randrange has the same syntax as range and unlike random.randint, the last value is **not** inclusive:

```
random.randrange(100)      # Random integer between 0 and 99
```

```
random.randrange(20, 50)    # Random integer between 20 and 49
```

```
random.randrange(10, 20, 3) # Random integer between 10 and 19 with step 3 (10, 13, 16 and 19)
```



## random

返回一个0到1之间的随机浮点数：

```
random.random() # 输出: 0.66486093215306317
```

## uniform

返回一个在 `x` 和 `y` (包含) 之间的随机浮点数：

```
random.uniform(1, 8) # 输出: 3.726062641730108
```

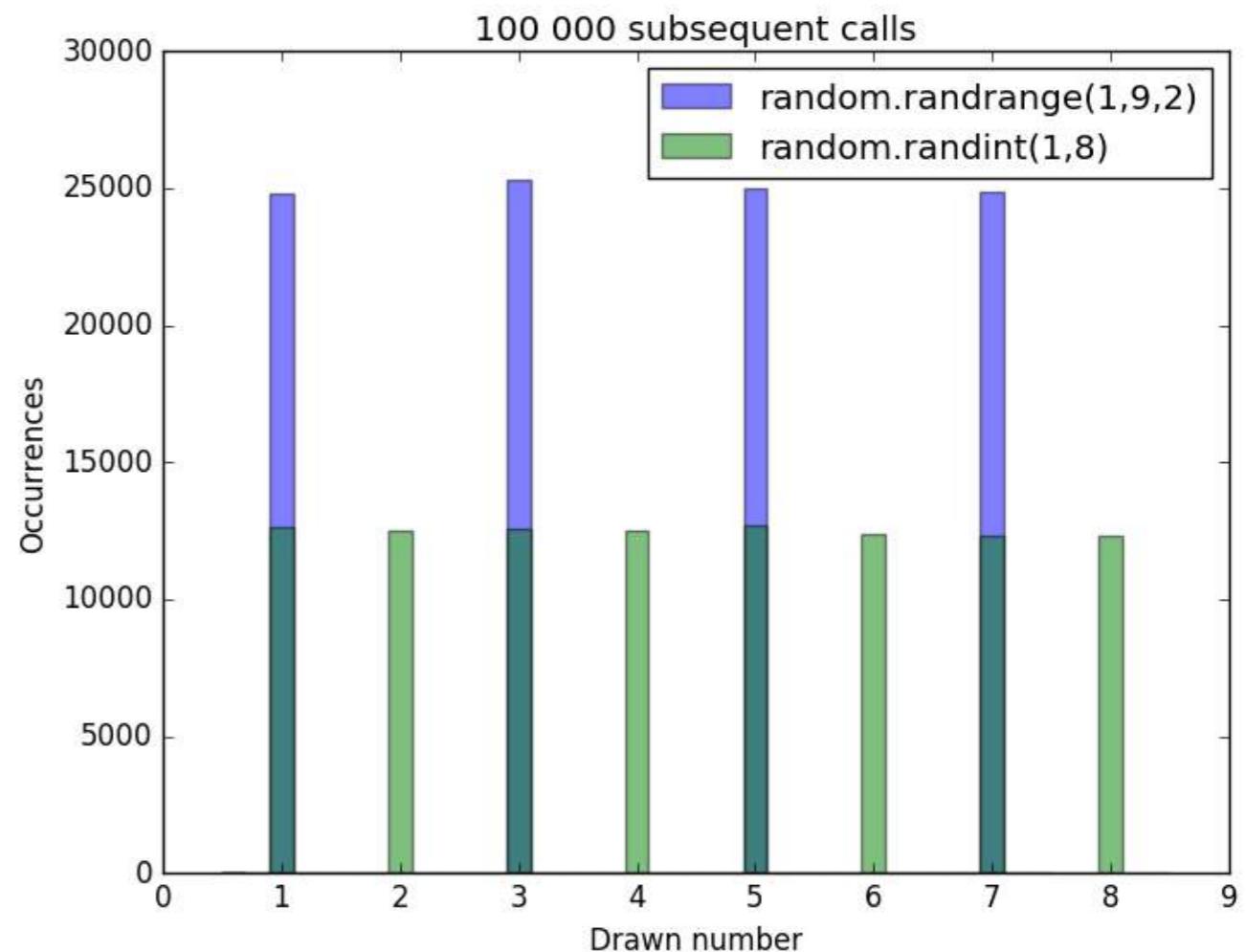
## 第55.5节：可复现的随机数：种子和状态

设置特定的种子将创建一个固定的随机数序列：

```
random.seed(5)          # 创建固定状态
print(random.randrange(0, 10)) # 获取0到9之间的随机整数
# 输出: 9
print(random.randrange(0, 10))
# Out: 4
```

重置种子将再次创建相同的“随机”序列：

```
random.seed(5)          # 将随机模块重置为相同的固定状态。
print(random.randrange(0, 10))
```



## random

Returns a random floating point number between 0 and 1:

```
random.random() # Out: 0.66486093215306317
```

## uniform

Returns a random floating point number between `x` and `y` (inclusive):

```
random.uniform(1, 8) # Out: 3.726062641730108
```

## Section 55.5: Reproducible random numbers: Seed and State

Setting a specific Seed will create a fixed random-number series:

```
random.seed(5)          # Create a fixed state
print(random.randrange(0, 10)) # Get a random integer between 0 and 9
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

Resetting the seed will create the same "random" sequence again:

```
random.seed(5)          # Reset the random module to the same fixed state.
print(random.randrange(0, 10))
```

```
# 输出: 9  
print(random.randrange(0, 10))  
# Out: 4
```

由于种子是固定的，这些结果总是9和4。如果不需要特定的数字，只需要值相同，也可以使用getstate和setstate来恢复到之前的状态：

```
save_state = random.getstate() # 获取当前状态  
print(random.randrange(0, 10))  
# 输出: 5  
print(random.randrange(0, 10))  
# 输出: 8  
  
random.setstate(save_state) # 重置为保存的状态  
print(random.randrange(0, 10))  
# 输出: 5  
print(random.randrange(0, 10))  
# 输出: 8
```

要伪随机化序列，可以用None作为seed：

```
random.seed(None)
```

或者调用seed方法且不带参数：

```
random.seed()
```

## 第55.6节：随机二元决策

```
import random  
  
概率 = 0.3  
  
如果 random.random() < 概率:  
    打印("概率为0.3的决策")  
否则:  
    打印("概率为0.7的决策")
```

```
# Out: 9  
print(random.randrange(0, 10))  
# Out: 4
```

Since the seed is fixed these results are always 9 and 4. If having specific numbers is not required only that the values will be the same one can also just use getstate and setstate to recover to a previous state:

```
save_state = random.getstate() # Get the current state  
print(random.randrange(0, 10))  
# Out: 5  
print(random.randrange(0, 10))  
# Out: 8  
  
random.setstate(save_state) # Reset to saved state  
print(random.randrange(0, 10))  
# Out: 5  
print(random.randrange(0, 10))  
# Out: 8
```

To pseudo-randomize the sequence again you seed with None:

```
random.seed(None)
```

Or call the seed method with no arguments:

```
random.seed()
```

## Section 55.6: Random Binary Decision

```
import random  
  
probability = 0.3  
  
if random.random() < probability:  
    print("Decision with probability 0.3")  
else:  
    print("Decision with probability 0.7")
```

# 第56章：functools模块

## 第56.1节：partial

partial函数从另一个函数创建部分函数应用。它用于绑定函数的部分参数（或关键字参数）值，并生成一个可调用对象，该对象不包含已定义的参数。

```
>>> 从 functools 导入 partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = '将 base16 字符串转换为整数'
>>> unhex('ca11ab1e')
3390155550
```

partial()，顾名思义，允许对函数进行部分求值。让我们看下面的例子：

```
在 [2]: from functools import partial
在 [3]: def f(a, b, c, x):
...:     return 1000*a + 100*b + 10*c + x
...
在 [4]: g = partial(f, 1, 1, 1)
在 [5]: print g(2)
1112
```

当 g 被创建时，f 接受四个参数 (a, b, c, x)，并对前三个参数 a, b, c 进行部分求值。f 的求值在调用 g 时完成，即 g(2)，将第四个参数传递给 f。

可以将 partial 理解为一个移位寄存器；一次向某个函数推入一个参数。对于数据以流形式输入且无法一次传入多个参数的情况，partial 非常实用。

## 第56.2节：cmp\_to\_key

Python 修改了其排序方法，改为接受一个键函数。该函数接受一个值并返回一个用于排序数组的键。

旧的比较函数通常接受两个值，并返回-1、0或+1，分别表示第一个参数小于、等于或大于第二个参数。这与新的键函数不兼容。

这就是functools.cmp\_to\_key的用武之地：

```
>>> import functools
>>> import locale
>>> sorted(["A", "S", "F", "D"], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

示例取自并改编自Python标准库文档。

## 第56.3节：lru\_cache

@lru\_cache装饰器可以用来包装一个计算量大、开销高的函数，使用最近最少使用（Least Recently Used）缓存。这允许函数调用被记忆化，使得未来使用相同参数的调用可以立即返回，而无需重新计算。

# Chapter 56: Functools Module

## Section 56.1: partial

The partial function creates partial function application from another function. It is used to bind values to some of the function's arguments (or keyword arguments) and produce a callable without the already defined arguments.

```
>>> from functools import partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = 'Convert base16 string to int'
>>> unhex('ca11ab1e')
3390155550
```

partial()，as the name suggests, allows a partial evaluation of a function. Let's look at following example:

```
In [2]: from functools import partial
In [3]: def f(a, b, c, x):
...:     return 1000*a + 100*b + 10*c + x
...
In [4]: g = partial(f, 1, 1, 1)
In [5]: print g(2)
1112
```

When g is created, f, which takes four arguments(a, b, c, x), is also partially evaluated for the first three arguments, a, b, c. Evaluation of f is completed when g is called, g(2), which passes the fourth argument to f.

One way to think of partial is a shift register; pushing in one argument at the time into some function. partial comes handy for cases where data is coming in as stream and we cannot pass more than one argument.

## Section 56.2: cmp\_to\_key

Python changed its sorting methods to accept a key function. Those functions take a value and return a key which is used to sort the arrays.

Old comparison functions used to take two values and return -1, 0 or +1 if the first argument is small, equal or greater than the second argument respectively. This is incompatible to the new key-function.

That's where functools.cmp\_to\_key comes in:

```
>>> import functools
>>> import locale
>>> sorted(["A", "S", "F", "D"], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

Example taken and adapted from the [Python Standard Library Documentation](#).

## Section 56.3: lru\_cache

The @lru\_cache decorator can be used wrap an expensive, computationally-intensive function with a [Least Recently Used](#) cache. This allows function calls to be memoized, so that future calls with the same parameters can return instantly instead of having to be recomputed.

```
@lru_cache(maxsize=None) # 无界缓存
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

>>> fibonacci(15)
```

在上面的例子中，fibonacci(3) 的值只计算了一次，而如果 fibonacci 没有 LRU 缓存，fibonacci(3) 将被计算超过 230 次。因此，@lru\_cache 对于递归函数或动态规划特别有用，因为一个开销较大的函数可能会被多次调用，且参数完全相同。

@lru\_cache 有两个参数

- maxsize：保存调用的数量。当唯一调用次数超过 maxsize 时，LRU 缓存将移除最近最少使用的调用。
- typed (3.3 版本新增)：用于判断不同类型但等价的参数是否属于不同的缓存记录（即 3.0 和 3 是否算作不同参数）的标志

我们也可以查看缓存统计信息：

```
>>> fib.cache_info()
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)
```

注意：由于 @lru\_cache 使用字典来缓存结果，函数的所有参数必须是可哈希的，缓存才能正常工作。

[官方 Python 文档关于 @lru\\_cache。@lru\\_cache 于 3.2 版本中添加。](#)

## 第 56.4 节：total\_ordering

当我们想创建一个可排序的类时，通常需要定义方法 `__eq__()`、`__lt__()`、`__le__()`、`__gt__()` 和 `__ge__()`。

应用于类的 `total_ordering` 装饰器允许只定义 `__eq__()` 和 `__lt__()`、`__le__()`、`__gt__()`、`__ge__()` 中的一个，仍然可以对该类执行所有排序操作。

```
@total_ordering
class Employee:
    ...

    def __eq__(self, other):
        return ((self.surname, self.name) == (other.surname, other.name))

    def __lt__(self, other):
        return ((self.surname, self.name) < (other.surname, other.name))
```

该装饰器使用提供的方法和代数运算的组合来推导其他比较方法。例如，如果我们定义了 `__lt__()` 和 `__eq__()` 并想推导 `__gt__()`，我们可以简单地检查 `not __lt__()` 且 `not __eq__()`。

注意：`total_ordering` 函数仅自 Python 2.7 起可用。

```
@lru_cache(maxsize=None) # Boundless cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

>>> fibonacci(15)
```

In the example above, the value of `fibonacci(3)` is only calculated once, whereas if `fibonacci` didn't have an LRU cache, `fibonacci(3)` would have been computed upwards of 230 times. Hence, `@lru_cache` is especially great for recursive functions or dynamic programming, where an expensive function could be called multiple times with the same exact parameters.

`@lru_cache` has two arguments

- maxsize: Number of calls to save. When the number of unique calls exceeds `maxsize`, the LRU cache will remove the least recently used calls.
- typed (added in 3.3): Flag for determining if equivalent arguments of different types belong to different cache records (i.e. if `3.0` and `3` count as different arguments)

We can see cache stats too:

```
>>> fib.cache_info()
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)
```

NOTE: Since `@lru_cache` uses dictionaries to cache results, all parameters for the function must be hashable for the cache to work.

[Official Python docs for @lru\\_cache.](#) `@lru_cache` was added in 3.2.

## Section 56.4: total\_ordering

When we want to create an orderable class, normally we need to define the methods `__eq__()`、`__lt__()`、`__le__()`、`__gt__()` 和 `__ge__()`。

The `total_ordering` decorator, applied to a class, permits the definition of `__eq__()` and only one between `__lt__()`、`__le__()`、`__gt__()` 和 `__ge__()`，and still allow all the ordering operations on the class.

```
@total_ordering
class Employee:
    ...

    def __eq__(self, other):
        return ((self.surname, self.name) == (other.surname, other.name))

    def __lt__(self, other):
        return ((self.surname, self.name) < (other.surname, other.name))
```

The decorator uses a composition of the provided methods and algebraic operations to derive the other comparison methods. For example if we defined `__lt__()` and `__eq__()` and we want to derive `__gt__()`, we can simply check `not __lt__()` and `not __eq__()`.

**Note:** The `total_ordering` function is only available since Python 2.7.

## 第56.5节：reduce

在Python 3.x中，此处已解释的reduce函数已从内置函数中移除，必须现在从functools模块中导入。

```
from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))
```

## Section 56.5: reduce

In Python 3.x, the `reduce` function already explained here has been removed from the built-ins and must now be imported from `functools`.

```
from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))
```

# 第57章：dis模块

## 第57.1节：什么是Python字节码？

Python是一种混合解释器。运行程序时，它首先将程序汇编成字节码，然后可以在Python解释器（也称为Python虚拟机）中运行。标准库中的dis模块可以通过反汇编类、方法、函数和代码对象，使Python字节码变得可读。

```
>>> def hello():
...     print "Hello, World"
...
>>> dis.dis(hello)
 2      0 LOAD_CONST      1 ('Hello, World')
 3 PRINT_ITEM
 4 PRINT_NEWLINE
 5 LOAD_CONST      0 (None)
 8 RETURN_VALUE
```

Python 解释器是基于栈的，采用先进后出（LIFO）系统。

Python 汇编语言（字节码）中的每个操作码（opcode）都会从栈中取固定数量的项，并将固定数量的项放回栈中。如果栈中没有足够的项供某个操作码使用，Python 解释器将崩溃，可能不会显示错误信息。

## 第57.2节：dis模块中的常量

```
EXTENDED_ARG = 145 # 所有大于此值的操作码有2个操作数
HAVE_ARGUMENT = 90 # 所有大于此值的操作码至少有1个操作数

cmp_op = ('<', '<=', '==', '!=', '>', '>=',
          'in', 'not in', 'is', 'is ...'
# 比较器ID列表。索引用作某些操作码的操作数
```

```
# 这些列表中的所有操作码的操作数均具有相应的类型
hascompare = [107]
hasconst = [100]
hasfree = [135, 136, 137]
hasjabs = [111, 112, 113, 114, 115, 119]
hasjrel = [93, 110, 120, 121, 122, 143]
haslocal = [124, 125, 126]
hasname = [90, 91, 95, 96, 97, 98, 101, 106, 108, 109, 116]

# 操作码到ID的映射
opmap = {'BINARY_ADD': 23, 'BINARY_AND': 64, 'BINARY_DIVIDE': 21, 'BIN...
# ID到操作码的映射
opname = ['STOP_CODE', 'POP_TOP', 'ROT_TWO', 'ROT_THREE', 'DUP_TOP', '...
```

## 第57.3节：反汇编模块

要反汇编一个Python模块，首先需要将其转换成.pyc文件（Python编译文件）。为此，运行

```
python -m compileall <file>.py
```

然后在解释器中运行

```
import dis
import marshal
```

# Chapter 57: The dis module

## Section 57.1: What is Python bytecode?

Python is a hybrid interpreter. When running a program, it first assembles it into *bytecode* which can then be run in the Python interpreter (also called a *Python virtual machine*). The `dis` module in the standard library can be used to make the Python bytecode human-readable by disassembling classes, methods, functions, and code objects.

```
>>> def hello():
...     print "Hello, World"
...
>>> dis.dis(hello)
 2      0 LOAD_CONST      1 ('Hello, World')
 3 PRINT_ITEM
 4 PRINT_NEWLINE
 5 LOAD_CONST      0 (None)
 8 RETURN_VALUE
```

The Python interpreter is stack-based and uses a first-in last-out system.

Each operation code (opcode) in the Python assembly language (the bytecode) takes a fixed number of items from the stack and returns a fixed number of items to the stack. If there aren't enough items on the stack for an opcode, the Python interpreter will crash, possibly without an error message.

## Section 57.2: Constants in the dis module

```
EXTENDED_ARG = 145 # All opcodes greater than this have 2 operands
HAVE_ARGUMENT = 90 # All opcodes greater than this have at least 1 operands

cmp_op = ('<', '<=', '==', '!=', '>', '>=',
          'in', 'not in', 'is', 'is ...'
# A list of comparator id's. The indices are used as operands in some opcodes

# All opcodes in these lists have the respective types as their operands
hascompare = [107]
hasconst = [100]
hasfree = [135, 136, 137]
hasjabs = [111, 112, 113, 114, 115, 119]
hasjrel = [93, 110, 120, 121, 122, 143]
haslocal = [124, 125, 126]
hasname = [90, 91, 95, 96, 97, 98, 101, 106, 108, 109, 116]

# A map of opcodes to ids
opmap = {'BINARY_ADD': 23, 'BINARY_AND': 64, 'BINARY_DIVIDE': 21, 'BIN...
# A map of ids to opcodes
opname = ['STOP_CODE', 'POP_TOP', 'ROT_TWO', 'ROT_THREE', 'DUP_TOP', '...
```

## Section 57.3: Disassembling modules

To disassemble a Python module, first this has to be turned into a .pyc file (Python compiled). To do this, run

```
python -m compileall <file>.py
```

Then in an interpreter, run

```
import dis
import marshal
```

```
with open("<file>.pyc", "rb") as code_f:  
    code_f.read(8) # 魔数和修改时间  
    code = marshal.load(code_f) # 返回一个可以反汇编的代码对象  
    dis.dis(code) # 输出反汇编结果
```

这将编译一个Python模块并使用 `dis` 输出字节码指令。该模块从未被导入，因此可以安全地用于不受信任的代码。

```
with open("<file>.pyc", "rb") as code_f:  
    code_f.read(8) # Magic number and modification time  
    code = marshal.load(code_f) # Returns a code object which can be disassembled  
    dis.dis(code) # Output the disassembly
```

This will compile a Python module and output the bytecode instructions with `dis`. The module is never imported so it is safe to use with untrusted code.

# 第58章：base64模块

参数	描述
base64.b64encode(s, altchars=None)	
s	类似字节的对象
altchars	一个长度为2或以上的类字节对象，用于在创建Base64字母表时替换'+'和'='字符。 多余的字符将被忽略。
base64.b64decode(s, altchars=None, validate=False)	
s	类似字节的对象
altchars	一个长度为2或以上的类字节对象，用于在创建Base64字母表时替换'+'和'='字符。 多余的字符将被忽略。
validate	如果validate为True，则在填充检查之前，不属于普通Base64字母表或替代字母表的字符不会被丢弃
base64.standard_b64encode(s)	
s	类似字节的对象
base64.standard_b64decode(s)	
s	类似字节的对象
base64.urlsafe_b64encode(s)	
s	类似字节的对象
base64.urlsafe_b64decode(s)	
s	类似字节的对象
b32encode(s)	
s	类似字节的对象
b32decode(s)	
s	类似字节的对象
base64.b16encode(s)	
s	类似字节的对象
base64.b16decode(s)	
s	类似字节的对象
base64.a85encode(b, foldspaces=False, wrapcol=0, pad=False, adobe=False)	
b	类似字节的对象
foldspaces	如果 foldspaces 为 True，字符 'y' 将被用来替代连续的4个空格。
wrapcol	换行前的字符数（0 表示无换行）
填充	如果 pad 为 True，字节将在编码前填充至4的倍数
adobe	如果 adobe 为 True，编码的序列将使用 Adobe 产品中使用的 '<~' 和 '~>' 进行框定
base64.a85decode(b, foldspaces=False, adobe=False, ignorechars=b'\r\n')	
b	类似字节的对象
foldspaces	如果 foldspaces 为 True，字符 'y' 将被用来替代连续的4个空格。

# Chapter 58: The base64 Module

Parameter	Description
base64.b64encode(s, altchars=None)	
s	A bytes-like object
altchars	A bytes-like object of length 2+ of characters to replace the '+' and '=' characters when creating the Base64 alphabet. Extra characters are ignored.
base64.b64decode(s, altchars=None, validate=False)	
s	A bytes-like object
altchars	A bytes-like object of length 2+ of characters to replace the '+' and '=' characters when creating the Base64 alphabet. Extra characters are ignored.
validate	If validate is True, the characters not in the normal Base64 alphabet or the alternative alphabet are <b>not</b> discarded before the padding check
base64.standard_b64encode(s)	
s	A bytes-like object
base64.standard_b64decode(s)	
s	A bytes-like object
base64.urlsafe_b64encode(s)	
s	A bytes-like object
base64.urlsafe_b64decode(s)	
s	A bytes-like object
b32encode(s)	
s	A bytes-like object
b32decode(s)	
s	A bytes-like object
base64.b16encode(s)	
s	A bytes-like object
base64.b16decode(s)	
s	A bytes-like object
base64.a85encode(b, foldspaces=False, wrapcol=0, pad=False, adobe=False)	
b	A bytes-like object
foldspaces	If foldspaces is True, the character 'y' will be used instead of 4 consecutive spaces.
wrapcol	The number characters before a newline (0 implies no newlines)
pad	If pad is True, the bytes are padded to a multiple of 4 before encoding
adobe	If adobe is True, the encoded sequenced with be framed with '<~' and '~>' as used with Adobe products
base64.a85decode(b, foldspaces=False, adobe=False, ignorechars=b'\t\r\n')	
b	A bytes-like object
foldspaces	If foldspaces is True, the character 'y' will be used instead of 4 consecutive spaces.

```
adobe
```

```
ignorechars
```

```
base64.b85encode(b, pad=False)
```

```
b
```

```
填充
```

```
base64.b85decode(b)
```

```
b
```

如果 adobe 为 True，编码的序列将使用 Adobe 产品中使用的 '<~' 和 "~~>' 进行框定

在编码过程中要忽略的类字节对象字符

类似字节的对象

如果 pad 为 True，字节将在编码前填充至4的倍数

类似字节的对象

Base64 编码是一种将二进制编码为使用基数 64 的 ASCII 字符串格式的常用方案。base64 模块是标准库的一部分，这意味着它随 Python 一起安装。理解字节和字符串对于本主题至关重要，可以在 [here](#) 复习。本主题解释了如何使用 base64 模块的各种功能和数字基数。

## 第 58.1 节：Base64 的编码和解码

要在脚本中包含 base64 模块，必须先导入它：

```
import base64
```

base64 编码和解码函数都需要一个类似字节的对象。为了将我们的字符串转换为字节，必须使用 Python 内置的 encode 函数进行编码。最常用的是UTF-8编码，然而所有这些标准编码（包括带有不同字符的语言）完整列表可以在官方 Python 文档中找到。下面是将字符串编码为字节的示例：

```
s = "Hello World!"  
b = s.encode("UTF-8")
```

上一行代码的输出将是：

```
b'Hello World!'
```

前缀b用于表示该值是一个字节对象。

要对这些字节进行 Base64 编码，我们使用base64.b64encode()函数：

```
import base64  
s = "Hello World!"  
b = s.encode("UTF-8")  
e = base64.b64encode(b)  
print(e)
```

该代码将输出以下内容：

```
b'SGVsbG8gV29ybGQh'
```

这些内容仍然是字节对象。要从这些字节中获取字符串，我们可以使用 Python 的decode()方法，配合 UTF-8 编码：

```
import base64  
s = "Hello World!"  
b = s.encode("UTF-8")  
e = base64.b64encode(b)
```

```
adobe
```

```
ignorechars
```

```
base64.b85encode(b, pad=False)
```

```
b
```

```
pad
```

```
base64.b85decode(b)
```

```
b
```

If adobe is True, the encoded sequenced will be framed with '<~' and "~~>' as used with Adobe products

A bytes-like object of characters to ignore in the encoding process

A bytes-like object

If pad is True, the bytes are padded to a multiple of 4 before encoding

A bytes-like object

Base 64 encoding represents a common scheme for encoding binary into ASCII string format using radix 64. The base64 module is part of the standard library, which means it installs along with Python. Understanding of bytes and strings is critical to this topic and can be reviewed [here](#). This topic explains how to use the various features and number bases of the base64 module.

## Section 58.1: Encoding and Decoding Base64

To include the base64 module in your script, you must import it first:

```
import base64
```

The base64 encode and decode functions both require a [bytes-like object](#). To get our string into bytes, we must encode it using Python's built in encode function. Most commonly, the [UTF-8](#) encoding is used, however a full list of these standard encodings (including languages with different characters) can be found [here](#) in the official Python Documentation. Below is an example of encoding a string into bytes:

```
s = "Hello World!"  
b = s.encode("UTF-8")
```

The output of the last line would be:

```
b'Hello World!'
```

The b prefix is used to denote the value is a bytes object.

To Base64 encode these bytes, we use the [base64.b64encode\(\)](#) function:

```
import base64  
s = "Hello World!"  
b = s.encode("UTF-8")  
e = base64.b64encode(b)  
print(e)
```

That code would output the following:

```
b'SGVsbG8gV29ybGQh'
```

which is still in the bytes object. To get a string out of these bytes, we can use Python's decode() method with the [UTF-8](#) encoding:

```
import base64  
s = "Hello World!"  
b = s.encode("UTF-8")  
e = base64.b64encode(b)
```

```
s1 = e.decode("UTF-8")
print(s1)
```

输出将会是：

```
SGVsbG8gV29ybGQh
```

如果我们想先编码字符串然后解码，可以使用base64.b64decode()方法：

```
import base64
# 创建一个字符串
s = "Hello World!"
# 将字符串编码为字节
b = s.encode("UTF-8")
# 对字节进行Base64编码
e = base64.b64encode(b)
# 将Base64字节解码为字符串
s1 = e.decode("UTF-8")
# 打印Base64编码的字符串
print("Base64 encoded:", s1)
# 将Base64编码的字符串编码为字节
b1 = s1.encode("UTF-8")
# 解码Base64字节
d = base64.b64decode(b1)
# 将字节解码为字符串
s2 = d.decode("UTF-8")
print(s2)
```

正如你所预期的，输出将是原始字符串：

```
Base64 encoded: SGVsbG8gV29ybGQh
Hello World!
```

## 第58.2节：Base32的编码与解码

base64 模块还包括用于 Base32 的编码和解码函数。这些函数与 Base64 函数非常相似：

```
import base64
# 创建一个字符串
s = "Hello World!"
# 将字符串编码为字节
b = s.encode("UTF-8")
# 对字节进行Base32编码
e = base64.b32encode(b)
# 解码Base32字节为字符串
s1 = e.decode("UTF-8")
# 打印Base32编码的字符串
print("Base32 Encoded:", s1)
# 将Base32编码的字符串编码为字节
b1 = s1.encode("UTF-8")
# 解码Base32字节
d = base64.b32decode(b1)
# 将字节解码为字符串
s2 = d.decode("UTF-8")
print(s2)
```

这将产生以下输出：

```
s1 = e.decode("UTF-8")
print(s1)
```

The output would then be:

```
SGVsbG8gV29ybGQh
```

If we wanted to encode the string and then decode we could use the `base64.b64decode()` method:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base64 Encode the bytes
e = base64.b64encode(b)
# Decoding the Base64 bytes to string
s1 = e.decode("UTF-8")
# Printing Base64 encoded string
print("Base64 Encoded:", s1)
# Encoding the Base64 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base64 bytes
d = base64.b64decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

As you may have expected, the output would be the original string:

```
Base64 Encoded: SGVsbG8gV29ybGQh
Hello World!
```

## Section 58.2: Encoding and Decoding Base32

The `base64` module also includes encoding and decoding functions for Base32. These functions are very similar to the Base64 functions:

```
import base64
# Creating a string
s = "Hello World!"
# Encoding the string into bytes
b = s.encode("UTF-8")
# Base32 Encode the bytes
e = base64.b32encode(b)
# Decoding the Base32 bytes to string
s1 = e.decode("UTF-8")
# Printing Base32 encoded string
print("Base32 Encoded:", s1)
# Encoding the Base32 encoded string into bytes
b1 = s1.encode("UTF-8")
# Decoding the Base32 bytes
d = base64.b32decode(b1)
# Decoding the bytes to string
s2 = d.decode("UTF-8")
print(s2)
```

This would produce the following output:

```
Base32 Encoded: JBSWY3DPEBLW64TMMQQQ====  
Hello World!
```

## 第58.3节：Base16的编码与解码

base64 模块还包括用于 Base16 的编码和解码函数。Base16 最常被称为十六进制。这些函数与 Base64 和 Base32 函数非常相似：

```
import base64  
# 创建一个字符串  
s = "Hello World!"  
# 将字符串编码为字节  
b = s.encode("UTF-8")  
# Base16 编码字节  
e = base64.b16encode(b)  
# 解码 Base16 字节为字符串  
s1 = e.decode("UTF-8")  
# 打印 Base16 编码的字符串  
print("Base16 编码:", s1)  
# 将 Base16 编码的字符串编码为字节  
b1 = s1.encode("UTF-8")  
# 解码 Base16 字节  
d = base64.b16decode(b1)  
# 将字节解码为字符串  
s2 = d.decode("UTF-8")  
print(s2)
```

这将产生以下输出：

```
Base16 编码: 48656C6C6F20576F726C6421  
Hello World!
```

## 第 58.4 节：ASCII85 的编码和解码

Adobe 创建了自己的编码，称为 ASCII85，类似于 Base85，但有所不同。该编码经常用于 Adobe PDF 文件中。这些函数在 Python 3.4 版本中发布。否则，这些函数

base64.a85encode() 和 base64.a85decode() 类似于之前的用法：

```
import base64  
# 创建一个字符串  
s = "Hello World!"  
# 将字符串编码为字节  
b = s.encode("UTF-8")  
# ASCII85 编码字节  
e = base64.a85encode(b)  
# 将 ASCII85 字节解码为字符串  
s1 = e.decode("UTF-8")  
# 打印 ASCII85 编码的字符串  
print("ASCII85 编码:", s1)  
# 将 ASCII85 编码的字符串编码为字节  
b1 = s1.encode("UTF-8")  
# 解码 ASCII85 字节  
d = base64.a85decode(b1)  
# 将字节解码为字符串  
s2 = d.decode("UTF-8")
```

```
Base32 Encoded: JBSWY3DPEBLW64TMMQQQ====  
Hello World!
```

## Section 58.3: Encoding and Decoding Base16

The base64 module also includes encoding and decoding functions for Base16. Base 16 is most commonly referred to as **hexadecimal**. These functions are very similar to the both the Base64 and Base32 functions:

```
import base64  
# Creating a string  
s = "Hello World!"  
# Encoding the string into bytes  
b = s.encode("UTF-8")  
# Base16 Encode the bytes  
e = base64.b16encode(b)  
# Decoding the Base16 bytes to string  
s1 = e.decode("UTF-8")  
# Printing Base16 encoded string  
print("Base16 Encoded:", s1)  
# Encoding the Base16 encoded string into bytes  
b1 = s1.encode("UTF-8")  
# Decoding the Base16 bytes  
d = base64.b16decode(b1)  
# Decoding the bytes to string  
s2 = d.decode("UTF-8")  
print(s2)
```

This would produce the following output:

```
Base16 Encoded: 48656C6C6F20576F726C6421  
Hello World!
```

## Section 58.4: Encoding and Decoding ASCII85

Adobe created its own encoding called **ASCII85** which is similar to Base85, but has its differences. This encoding is used frequently in Adobe PDF files. These functions were released in Python version 3.4. Otherwise, the functions `base64.a85encode()` and `base64.a85decode()` are similar to the previous:

```
import base64  
# Creating a string  
s = "Hello World!"  
# Encoding the string into bytes  
b = s.encode("UTF-8")  
# ASCII85 Encode the bytes  
e = base64.a85encode(b)  
# Decoding the ASCII85 bytes to string  
s1 = e.decode("UTF-8")  
# Printing ASCII85 encoded string  
print("ASCII85 Encoded:", s1)  
# Encoding the ASCII85 encoded string into bytes  
b1 = s1.encode("UTF-8")  
# Decoding the ASCII85 bytes  
d = base64.a85decode(b1)  
# Decoding the bytes to string  
s2 = d.decode("UTF-8")
```

```
print(s2)
```

输出如下内容：

```
ASCII85 编码：87cURD]i,"Ebo80  
Hello World!
```

## 第58.5节：Base85的编码与解码

就像Base64、Base32和Base16函数一样，Base85的编码和解码函数是 `base64.b85encode()` 和 `base64.b85decode()`：

```
import base64  
# 创建一个字符串  
s = "Hello World!"  
# 将字符串编码为字节  
b = s.encode("UTF-8")  
# 对字节进行Base85编码  
e = base64.b85encode(b)  
# 将Base85字节解码为字符串  
s1 = e.decode("UTF-8")  
# 打印Base85编码的字符串  
print("Base85 Encoded:", s1)  
# 将Base85编码的字符串编码为字节  
b1 = s1.encode("UTF-8")  
# 解码Base85字节  
d = base64.b85decode(b1)  
# 将字节解码为字符串  
s2 = d.decode("UTF-8")  
print(s2)
```

输出如下内容：

```
Base85 编码：NM&qnZy;B1a%^NF  
你好，世界！
```

```
print(s2)
```

This outputs the following:

```
ASCII85 Encoded: 87cURD]i,"Ebo80  
Hello World!
```

## Section 58.5: Encoding and Decoding Base85

Just like the Base64, Base32, and Base16 functions, the Base85 encoding and decoding functions are `base64.b85encode()` and `base64.b85decode()`:

```
import base64  
# Creating a string  
s = "Hello World!"  
# Encoding the string into bytes  
b = s.encode("UTF-8")  
# Base85 Encode the bytes  
e = base64.b85encode(b)  
# Decoding the Base85 bytes to string  
s1 = e.decode("UTF-8")  
# Printing Base85 encoded string  
print("Base85 Encoded:", s1)  
# Encoding the Base85 encoded string into bytes  
b1 = s1.encode("UTF-8")  
# Decoding the Base85 bytes  
d = base64.b85decode(b1)  
# Decoding the bytes to string  
s2 = d.decode("UTF-8")  
print(s2)
```

which outputs the following:

```
Base85 Encoded: NM&qnZy;B1a%^NF  
Hello World!
```

# 第59章：队列模块

Queue 模块实现了多生产者、多消费者队列。它在多线程编程中尤为有用，当信息必须在线程之间安全交换时。  
queue 模块提供三种类型的队列，分别是：1. Queue 2. LifoQueue 3. PriorityQueue 可能出现的异常有：1. Full (队列溢出) 2. Empty (队列下溢)

## 第59.1节：简单示例

```
from Queue import Queue

question_queue = Queue()

for x in range(1,10):
    temp_dict = ('key', x)
    question_queue.put(temp_dict)

while(not question_queue.empty()):
    item = question_queue.get()
    print(str(item))
```

输出：

```
('key', 1)
('key', 2)
('key', 3)
('key', 4)
('key', 5)
('key', 6)
('key', 7)
('key', 8)
('key', 9)
```

# Chapter 59: Queue Module

The Queue module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. There are three types of queues provided by the queue module, which are as follows: 1. Queue 2. LifoQueue 3. PriorityQueue Exception which could be come: 1. Full (queue overflow) 2. Empty (queue underflow)

## Section 59.1: Simple example

```
from Queue import Queue

question_queue = Queue()

for x in range(1,10):
    temp_dict = ('key', x)
    question_queue.put(temp_dict)

while(not question_queue.empty()):
    item = question_queue.get()
    print(str(item))
```

Output:

```
('key', 1)
('key', 2)
('key', 3)
('key', 4)
('key', 5)
('key', 6)
('key', 7)
('key', 8)
('key', 9)
```

# 视频：机器学习 、数据科学和使用 Pyth on的深度学习

完整的动手机器学习教程，涵盖数据科学、Tensorflow、人工智能和神经网络



- ✓ 使用Tensorflow和Keras构建人工神经网络
- ✓ 使用深度学习对图像、数据和情感进行分类
- ✓ 使用线性回归、多项式回归和多元回归进行预测
- ✓ 使用Matplotlib和Seaborn进行数据可视化
- ✓ 利用Apache Spark的MLlib实现大规模机器学习
- ✓ 理解强化学习——以及如何构建吃豆人机器人
- ✓ 使用K-Means聚类、支持向量机（SVM）、K近邻（KNN）、决策树、朴素贝叶斯和主成分分析（PCA）对数据进行分类
- ✓ 使用训练/测试和K折交叉验证选择和调优模型
- ✓ 使用基于物品和基于用户的协同过滤构建电影推荐系统

今天观看 →

# VIDEO: Machine Learning, Data Science and Deep Learning with Python

Complete hands-on machine learning tutorial with data science, Tensorflow, artificial intelligence, and neural networks



- ✓ Build artificial neural networks with Tensorflow and Keras
- ✓ Classify images, data, and sentiments using deep learning
- ✓ Make predictions using linear regression, polynomial regression, and multivariate regression
- ✓ Data Visualization with Matplotlib and Seaborn
- ✓ Implement machine learning at massive scale with Apache Spark's MLlib
- ✓ Understand reinforcement learning - and how to build a Pac-Man bot
- ✓ Classify data using K-Means clustering, Support Vector Machines (SVM), KNN, Decision Trees, Naive Bayes, and PCA
- ✓ Use train/test and K-Fold cross validation to choose and tune your models
- ✓ Build a movie recommender system using item-based and user-based collaborative filtering

Watch Today →

# 第60章：双端队列模块

## 参数

	详情
可迭代对象	创建一个双端队列，初始元素从另一个可迭代对象复制而来。
maxlen	限制双端队列的最大长度，当添加新元素时会推出旧元素。

## 第60.1节：双端队列的基本使用

该类中主要用的方法是popleft和appendleft

```
from collections import deque

d = deque([1, 2, 3])
p = d.popleft()      # p = 1, d = deque([2, 3])
d.appendleft(5)      # d = deque([5, 2, 3])
```

## 第60.2节：双端队列中可用的方法

创建空双端队列：

```
dl = deque() # deque([]) 创建空的双端队列
```

创建带有一些元素的双端队列：

```
dl = deque([1, 2, 3, 4]) # deque([1, 2, 3, 4])
```

向双端队列添加元素：

```
dl.append(5) # deque([1, 2, 3, 4, 5])
```

向双端队列左侧添加元素：

```
dl.appendleft(0) # deque([0, 1, 2, 3, 4, 5])
```

向双端队列添加元素列表：

```
dl.extend([6, 7]) # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

从左侧添加元素列表到双端队列：

```
dl.extendleft([-2, -1]) # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

使用.pop()方法会自然地从右侧移除一个元素：

```
dl.pop() # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

使用.popleft()方法从左侧移除一个元素：

```
dl.popleft() # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

通过值移除元素：

# Chapter 60: Deque Module

## Parameter

	Details
iterable	Creates the deque with initial elements copied from another iterable.
maxlen	Limits how large the deque can be, pushing out old elements as new are added.

## Section 60.1: Basic deque using

The main methods that are useful with this class are popleft and appendleft

```
from collections import deque

d = deque([1, 2, 3])
p = d.popleft()      # p = 1, d = deque([2, 3])
d.appendleft(5)      # d = deque([5, 2, 3])
```

## Section 60.2: Available methods in deque

Creating empty deque:

```
dl = deque() # deque([]) creating empty deque
```

Creating deque with some elements:

```
dl = deque([1, 2, 3, 4]) # deque([1, 2, 3, 4])
```

Adding element to deque:

```
dl.append(5) # deque([1, 2, 3, 4, 5])
```

Adding element left side of deque:

```
dl.appendleft(0) # deque([0, 1, 2, 3, 4, 5])
```

Adding list of elements to deque:

```
dl.extend([6, 7]) # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

Adding list of elements to from the left side:

```
dl.extendleft([-2, -1]) # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

Using .pop() element will naturally remove an item from the right side:

```
dl.pop() # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

Using .popleft() element to remove an item from the left side:

```
dl.popleft() # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

Remove element by its value:

```
dl.remove(1) # deque([-2, 0, 2, 3, 4, 5, 6])
```

反转deque中元素的顺序：

```
dl.reverse() # deque([6, 5, 4, 3, 2, 0, -2])
```

## 第60.3节：限制deque大小

在创建双端队列时使用maxlen参数来限制双端队列的大小：

```
from collections import deque
d = deque(maxlen=3) # 仅保存3个元素
d.append(1) # deque([1])
d.append(2) # deque([1, 2])
d.append(3) # deque([1, 2, 3])
d.append(4) # deque([2, 3, 4]) (因为最大长度是3, 1被移除)
```

## 第60.4节：广度优先搜索

Deque是Python中唯一具有快速队列操作的数据结构。（注意queue.Queue通常不适用，因为它是用于线程间通信的。）队列的一个基本用例是广度优先搜索。

```
from collections import deque

def bfs(graph, root):
    distances = {}
    distances[root] = 0
    q = deque([root])
    while q:
        # 最早看到的（但尚未访问的）节点将是最左边的那个。
        current = q.popleft()
        for neighbor in graph[current]:
            if neighbor not in distances:
                distances[neighbor] = distances[current] + 1
                # 当我们看到一个新节点时，将其添加到队列的右侧。
                q.append(neighbor)
    return distances
```

假设我们有一个简单的有向图：

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

我们现在可以找到某些起始位置的距离：

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

```
dl.remove(1) # deque([-2, 0, 2, 3, 4, 5, 6])
```

Reverse the order of the elements in deque:

```
dl.reverse() # deque([6, 5, 4, 3, 2, 0, -2])
```

## Section 60.3: limit deque size

Use the maxlen parameter while creating a deque to limit the size of the deque:

```
from collections import deque
d = deque(maxlen=3) # only holds 3 items
d.append(1) # deque([1])
d.append(2) # deque([1, 2])
d.append(3) # deque([1, 2, 3])
d.append(4) # deque([2, 3, 4]) (1 is removed because its maxlen is 3)
```

## Section 60.4: Breadth First Search

The Deque is the only Python data structure with fast [Queue operations](#). (Note queue.Queue isn't normally suitable, since it's meant for communication between threads.) A basic use case of a Queue is the [breadth first search](#).

```
from collections import deque

def bfs(graph, root):
    distances = {}
    distances[root] = 0
    q = deque([root])
    while q:
        # The oldest seen (but not yet visited) node will be the left most one.
        current = q.popleft()
        for neighbor in graph[current]:
            if neighbor not in distances:
                distances[neighbor] = distances[current] + 1
                # When we see a new node, we add it to the right side of the queue.
                q.append(neighbor)
    return distances
```

Say we have a simple directed graph:

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

We can now find the distances from some starting position:

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

# 第61章：Webbrowser模块

参数	详情
<code>webbrowser.open()</code>	
<code>url</code>	要在网页浏览器中打开的URL
<code>new</code>	0 在现有标签页中打开URL，1 在新窗口中打开，2 在新标签页中打开
<code>autoraise</code>	如果设置为True，窗口将被移到其他窗口之上
<code>webbrowser.open_new()</code>	
<code>url</code>	要在网页浏览器中打开的URL
<code>webbrowser.open_new_tab()</code>	
<code>url</code>	要在网页浏览器中打开的URL
<code>webbrowser.get()</code>	
<code>使用</code>	要使用的浏览器
<code>webbrowser.register()</code>	
<code>url</code>	浏览器名称
<code>构造函数</code>	可执行浏览器的路径 (help) _____
<code>实例</code>	从 <code>webbrowser.get()</code> 方法返回的一个网页浏览器实例

根据Python的标准文档，`webbrowser`模块提供了一个高级接口，允许向用户显示基于网页的文档。本文介绍并演示了`webbrowser`模块的正确用法。

## 第61.1节：使用默认浏览器打开URL

要简单地打开一个URL，使用`webbrowser.open()`方法：

```
import webbrowser  
webbrowser.open("http://stackoverflow.com")
```

如果当前有浏览器窗口打开，该方法将在指定的URL处打开一个新标签页。如果没有窗口打开，该方法将打开操作系统的默认浏览器并导航到参数中的URL。`open`方法支持以下参数：

- `url` - 要在浏览器中打开的URL（字符串）**[必需]**
- `new` - 0表示在现有标签页打开，1表示打开新窗口，2表示打开新标签页（整数）**[默认0]**
- `autoraise` - 如果设置为True，窗口将被置于其他应用窗口之上（布尔值）**[默认False]**

注意，`new`和`autoraise`参数很少起作用，因为大多数现代浏览器拒绝这些命令。

`Webbrowser`还可以尝试使用`open_new`方法在新窗口中打开URL：

```
import webbrowser  
webbrowser.open_new("http://stackoverflow.com")
```

此方法通常被现代浏览器忽略，URL 通常会在新标签页中打开。模块可以尝试使用`open_new_tab`方法打开新标签页：

```
import webbrowser  
webbrowser.open_new_tab("http://stackoverflow.com")
```

# Chapter 61: Webbrowser Module

Parameter	Details
<code>webbrowser.open()</code>	
<code>url</code>	the URL to open in the web browser
<code>new</code>	0 opens the URL in the existing tab, 1 opens in a new window, 2 opens in new tab
<code>autoraise</code>	if set to True, the window will be moved on top of the other windows
<code>webbrowser.open_new()</code>	
<code>url</code>	the URL to open in the web browser
<code>webbrowser.open_new_tab()</code>	
<code>url</code>	the URL to open in the web browser
<code>webbrowser.get()</code>	
<code>using</code>	the browser to use
<code>webbrowser.register()</code>	
<code>url</code>	browser name
<code>constructor</code>	path to the executable browser ( <a href="#">help</a> )
<code>instance</code>	An instance of a web browser returned from the <code>webbrowser.get()</code> method

According to Python's standard documentation, the `webbrowser` module provides a high-level interface to allow displaying Web-based documents to users. This topic explains and demonstrates proper usage of the `webbrowser` module.

## Section 61.1: Opening a URL with Default Browser

To simply open a URL, use the `webbrowser.open()` method:

```
import webbrowser  
webbrowser.open("http://stackoverflow.com")
```

If a browser window is currently open, the method will open a new tab at the specified URL. If no window is open, the method will open the operating system's default browser and navigate to the URL in the parameter. The `open` method supports the following parameters:

- `url` - the URL to open in the web browser (string) **[required]**
- `new` - 0 opens in existing tab, 1 opens new window, 2 opens new tab (integer) **[default 0]**
- `autoraise` - if set to True, the window will be moved on top of other applications' windows (Boolean) **[default False]**

Note, the `new` and `autoraise` arguments rarely work as the majority of modern browsers refuse these commands.

`Webbrowser` can also try to open URLs in new windows with the `open_new` method:

```
import webbrowser  
webbrowser.open_new("http://stackoverflow.com")
```

This method is commonly ignored by modern browsers and the URL is usually opened in a new tab. Opening a new tab can be tried by the module using the `open_new_tab` method:

```
import webbrowser  
webbrowser.open_new_tab("http://stackoverflow.com")
```

## 第61.2节：使用不同浏览器打开URL

webbrowser模块还支持使用register()和get()方法管理不同的浏览器。get方法用于通过特定可执行文件路径创建浏览器控制器，register方法用于将这些可执行文件绑定到预设的浏览器类型以供将来使用，通常在使用多种浏览器类型时使用。

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
ff.open("http://stackoverflow.com/")
```

注册浏览器类型：

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
webbrowser.register('firefox', None, ff)
# 现在要在将来使用Firefox，可以使用此方法
webbrowser.get('firefox').open("https://stackoverflow.com/")
```

## Section 61.2: Opening a URL with Different Browsers

The webbrowser module also supports different browsers using the register() and get() methods. The get method is used to create a browser controller using a specific executable's path and the register method is used to attach these executables to preset browser types for future use, commonly when multiple browser types are used.

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
ff.open("http://stackoverflow.com/")
```

Registering a browser type:

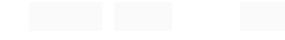
```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
webbrowser.register('firefox', None, ff)
# Now to refer to use Firefox in the future you can use this
webbrowser.get('firefox').open("https://stackoverflow.com/")
```

# 第62章：tkinter

Tkinter 是 Python 最流行的 GUI (图形用户界面) 库。本文介绍了该库的正确使用方法及其功能。

## 第 62.1 节：几何管理器

Tkinter 有三种几何管理机制：place、pack 和 grid。



place 管理器使用绝对像素坐标。

pack 管理器将控件放置在四个边中的一个。新控件会放置在已有控件的旁边。

grid 管理器将控件放置在类似动态调整大小的电子表格的网格中。

### Place

widget.place 最常用的关键字参数如下：

- x, 控件的绝对 x 坐标
- y, 控件的绝对 y 坐标
- height, 控件的绝对高度
- width, 控件的绝对宽度

使用place的代码示例：

```
class PlaceExample(Frame):  
    def __init__(self, master):  
        Frame.__init__(self, master)  
        self.grid()  
        top_text=Label(master, text="这是位于原点顶部的文本")  
        #top_text.pack()  
        top_text.place(x=0, y=0, height=50, width=200)  
        bottom_right_text=Label(master, text="这是位于位置200,400的文本")  
        #top_text.pack()  
        bottom_right_text.place(x=200, y=400, height=50, width=200)  
# Spawn Window  
if __name__=="__main__":  
    root=Tk()  
    place_frame=PlaceExample(root)  
    place_frame.mainloop()
```

### Pack

widget.pack 可以接受以下关键字参数：

- expand, 是否填充父容器剩余空间
- fill, 是否扩展以填满所有空间 (NONE (默认)、X、Y 或 BOTH)
- side, 控件靠近的边 (TOP (默认)、BOTTOM、LEFT 或 RIGHT)

### Grid

widget.grid 最常用的关键字参数如下：

- row, 控件所在的行 (默认最小未占用行)
- rowspan, 控件跨越的行数 (默认 1)
- column, 控件所在的列 (默认 0)

# Chapter 62: tkinter

Released in Tkinter is Python's most popular GUI (Graphical User Interface) library. This topic explains proper usage of this library and its features.

## Section 62.1: Geometry Managers

Tkinter has three mechanisms for geometry management: place, pack, and grid.

The place manager uses absolute pixel coordinates.

The pack manager places widgets into one of 4 sides. New widgets are placed next to existing widgets.

The grid manager places widgets into a grid similar to a dynamically resizing spreadsheet.

### Place

The most common keyword arguments for widget.place are as follows:

- x, the absolute x-coordinate of the widget
- y, the absolute y-coordinate of the widget
- height, the absolute height of the widget
- width, the absolute width of the widget

A code example using place:

```
class PlaceExample(Frame):  
    def __init__(self, master):  
        Frame.__init__(self, master)  
        self.grid()  
        top_text=Label(master, text="This is on top at the origin")  
        #top_text.pack()  
        top_text.place(x=0, y=0, height=50, width=200)  
        bottom_right_text=Label(master, text="This is at position 200, 400")  
        #top_text.pack()  
        bottom_right_text.place(x=200, y=400, height=50, width=200)  
# Spawn Window  
if __name__=="__main__":  
    root=Tk()  
    place_frame=PlaceExample(root)  
    place_frame.mainloop()
```

### Pack

widget.pack can take the following keyword arguments:

- expand, whether or not to fill space left by parent
- fill, whether to expand to fill all space (NONE (default), X, Y, or BOTH)
- side, the side to pack against (TOP (default), BOTTOM, LEFT, or RIGHT)

### Grid

The most commonly used keyword arguments of widget.grid are as follows:

- row, the row of the widget (default smallest unoccupied)
- rowspan, the number of columns a widget spans (default 1)
- column, the column of the widget (default 0)

- colspan, 控件跨越的列数（默认1）
- sticky, 如果网格单元格比控件大, 控件放置的位置 (N、NE、E、SE、S、SW、W、NW的组合)

行和列从零开始索引。行向下增加, 列向右增加。

使用grid的代码示例：

```
from tkinter import *

class GridExample(Frame):
    def __init__(self, master):
        Frame.__init__(self, master)
        self.grid()
        top_text=Label(self, text="此文本显示在左上角")
        top_text.grid() # 默认位置 0, 0
        bottom_text=Label(self, text="此文本显示在左下角")
        bottom_text.grid() # 默认位置 1, 0
        right_text=Label(self, text="此文本显示在右侧并跨越两行",
                         wraplength=100)
        right_text.grid(row=0, column=1, rowspan=2)
        # 位置是 0,1
        # rowspan 表示实际位置是 [0-1],1
        # Spawn Window
if __name__=="__main__":
    root=Tk()
    grid_frame=GridExample(root)
    grid_frame.mainloop()
```

切勿在同一框架内混合使用pack和grid ! 这样做会导致应用程序死锁！

## 第62.2节：一个最小的tkinter应用程序

tkinter是一个GUI工具包, 提供了对Tk/Tcl GUI库的封装, 并随Python一起提供。以下代码使用kinter创建一个新窗口, 并在窗口主体中放置一些文本。

注意：在Python 2中，大小写可能略有不同，详见下方备注部分。

```
import tkinter as tk

# GUI窗口是基本tkinter Frame对象的子类
class HelloWorldFrame(tk.Frame):
    def __init__(self, master):
        # 调用父类构造函数
        tk.Frame.__init__(self, master)
        # 将框架放入主窗口
        self.grid()
        # 创建带有"Hello World"文本的文本框
        hello = tk.Label(self, text="Hello World! This label can hold strings!")
        # 将文本框放入框架中
        hello.grid(row=0, column=0)

    # 生成窗口
if __name__ == "__main__":
    # 创建主窗口对象
    root = tk.Tk()
    # 设置窗口标题
    root.title("Hello World!")
```

- colspan, the number of columns a widget spans (default 1)
- sticky, where to place widget if the grid cell is larger than it (combination of N,NE,E,SE,S,SW,W,NW)

The rows and columns are zero indexed. Rows increase going down, and columns increase going right.

A code example using grid:

```
from tkinter import *

class GridExample(Frame):
    def __init__(self, master):
        Frame.__init__(self, master)
        self.grid()
        top_text=Label(self, text="This text appears on top left")
        top_text.grid() # Default position 0, 0
        bottom_text=Label(self, text="This text appears on bottom left")
        bottom_text.grid() # Default position 1, 0
        right_text=Label(self, text="This text appears on the right and spans both rows",
                         wraplength=100)
        right_text.grid(row=0, column=1, rowspan=2)
        # Position is 0,1
        # Rowspan means actual position is [0-1],1
        # Spawn Window
if __name__=="__main__":
    root=Tk()
    grid_frame=GridExample(root)
    grid_frame.mainloop()
```

Never mix pack and grid within the same frame! Doing so will lead to application deadlock!

## Section 62.2: A minimal tkinter Application

tkinter is a GUI toolkit that provides a wrapper around the Tk/Tcl GUI library and is included with Python. The following code creates a new window using tkinter and places some text in the window body.

Note: In Python 2, the capitalization may be slightly different, see Remarks section below.

```
import tkinter as tk

# GUI window is a subclass of the basic tkinter Frame object
class HelloWorldFrame(tk.Frame):
    def __init__(self, master):
        # Call superclass constructor
        tk.Frame.__init__(self, master)
        # Place frame into main window
        self.grid()
        # Create text box with "Hello World" text
        hello = tk.Label(self, text="Hello World! This label can hold strings!")
        # Place text box into frame
        hello.grid(row=0, column=0)

    # Spawn window
if __name__ == "__main__":
    # Create main window object
    root = tk.Tk()
    # Set title of window
    root.title("Hello World!")
```

```
# 实例化 HelloWorldFrame 对象  
hello_frame = HelloWorldFrame(root)  
# 启动 GUI  
hello_frame.mainloop()
```

```
# Instantiate HelloWorldFrame object  
hello_frame = HelloWorldFrame(root)  
# Start GUI  
hello_frame.mainloop()
```

# 第63章：pyautogui 模块

pyautogui 是一个用于控制鼠标和键盘的模块。该模块主要用于自动化鼠标点击和键盘按键任务。对于鼠标，屏幕坐标 (0,0) 从左上角开始。如果你失去控制，可以快速将鼠标光标移动到左上角，这将从 Python 手中接管鼠标和键盘的控制权并将其还给你。

## 第63.1节：鼠标功能

以下是一些用于控制鼠标的有用函数。

```
size()          # 返回屏幕的大小  
position()     # 返回鼠标当前的位置  
moveTo(200,0,duration=1.5)    # 将光标移动到 (200,0) 位置, 延迟1.5秒  
moveRel()       # 相对于当前位置移动光标。  
click(337,46)   # 在指定位置点击dragRel()           # 相对于当前位置拖  
动鼠标pyautogui.displayMousePosition()  # 显示当前鼠标位置, 但应在终端执行。
```

## 第63.2节：键盘功能

以下是一些用于自动按键的有用键盘功能。

```
typewrite('')    #这将在当前窗口聚焦的位置输入字符串。  
typewrite(['a','b','left','left','X','Y'])  
pyautogui.KEYBOARD_KEYS    #获取所有键盘按键的列表。  
pyautogui.hotkey('ctrl','o')  #用于组合键的输入。
```

## 第63.3节：截图与图像识别

这些功能将帮助你截图，并将图像与屏幕部分进行匹配。

```
.screenshot('c:\\path')      #截图。  
.locateOnScreen('c:\\path')   #在屏幕上搜索该图像并获取其坐标。  
.locateCenterOnScreen('c:\\path')  #获取屏幕上图像的中心坐标。
```

# Chapter 63: pyautogui module

pyautogui is a module used to control mouse and keyboard. This module is basically used to automate mouse click and keyboard press tasks. For the mouse, the coordinates of the screen (0,0) start from the top-left corner. If you are out of control, then quickly move the mouse cursor to top-left, it will take the control of mouse and keyboard from the Python and give it back to you.

## Section 63.1: Mouse Functions

These are some of useful mouse functions to control the mouse.

```
size()          #gave you the size of the screen  
position()     #return current position of mouse  
moveTo(200,0,duration=1.5)    #move the cursor to (200,0) position with 1.5 second delay  
moveRel()       #move the cursor relative to your current position.  
click(337,46)   #it will click on the position mention there  
dragRel()       #it will drag the mouse relative to position  
pyautogui.displayMousePosition()  #gave you the current mouse position but should be done on terminal.
```

## Section 63.2: Keyboard Functions

These are some of useful keyboard functions to automate the key pressing.

```
typewrite('')    #this will type the string on the screen where current window has focused.  
typewrite(['a','b','left','left','X','Y'])  
pyautogui.KEYBOARD_KEYS    #get the list of all the keyboard_keys.  
pyautogui.hotkey('ctrl','o')  #for the combination of keys to enter.
```

## Section 63.3: Screenshot And Image Recognition

These function will help you to take the screenshot and also match the image with the part of the screen.

```
.screenshot('c:\\path')      #get the screenshot.  
.locateOnScreen('c:\\path')   #search that image on screen and get the coordinates for you.  
.locateCenterOnScreen('c:\\path')  #get the coordinate for the image on screen.
```

# 第64章：索引与切片

## 参数

	描述
对象	您想要从中提取“子对象”的对象
起始	您希望子对象开始的obj索引（请记住，Python是从零开始索引的，这意味着obj的第一个元素索引为0）。如果省略，默认为0。
结束	您希望子对象结束的（不包含）obj索引。如果省略，默认为len(obj)。
步长	允许您选择每隔step个元素。如果省略，默认为1。

## 第64.1节：基本切片

对于任何可迭代对象（例如字符串、列表等），Python 允许你切片并返回其数据的子字符串或子列表。

切片格式：

```
iterable_name[start:stop:step]
```

其中，

- start 是切片的起始索引。默认为 0（第一个元素的索引）
- stop 是切片的结束索引的下一个位置。默认为 len(iterable)
- step 是步长（通过下面的示例更好理解）

示例：

```
a = "abcdef"
a           # "abcdef"
          # 与 a[:] 或 a[::] 相同, 因为它们对三个索引都使用默认值
a[-1]      # "f"
a[:]       # "abcdef"
a[::]     # "abcdef"
a[3:]      # "def" (从索引3开始, 到末尾(默认为可迭代对象的长度))
a[:4]      # "abcd" (从开头(默认0) 到位置4(不包括))
a[2:4]     # "cd" (从位置2到位置4(不包括))
```

此外，上述任何一种都可以与定义的步长一起使用：

```
a[::2]      # "ace" (每隔2个元素取一个)
a[1:4:2]    # "bd" (从索引1到索引4(不包括), 每隔2个元素取一个)
```

索引可以为负数，此时从序列末尾开始计算

```
a[:-1]     # "abcde" (从索引0(默认) 到倒数第二个元素(最后一个元素-1))
a[:-2]     # "abcd" (从索引0(默认) 到倒数第三个元素(最后一个元素-2))
a[-1:]     # "f" (从最后一个元素到末尾(默认len()))
```

步长也可以为负数，此时切片将以反向顺序遍历列表：

```
a[3:1:-1]  # "dc" (从索引3到索引1(不包括), 反向遍历)
```

该结构对于反转可迭代对象非常有用

```
a[::-1]    # "fedcba" (从最后一个元素(默认len()-1) 到第一个元素, 反向遍历(步长-1))
```

# Chapter 64: Indexing and Slicing

## Paramer

	Description
obj	The object that you want to extract a "sub-object" from
start	The index of obj that you want the sub-object to start from (keep in mind that Python is zero-indexed, meaning that the first item of obj has an index of 0). If omitted, defaults to 0.
stop	The (non-inclusive) index of obj that you want the sub-object to end at. If omitted, defaults to len(obj).
step	Allows you to select only every step item. If omitted, defaults to 1.

## Section 64.1: Basic Slicing

For any iterable (for eg. a string, list, etc), Python allows you to slice and return a substring or sublist of its data.

Format for slicing:

```
iterable_name[start:stop:step]
```

where,

- start is the first index of the slice. Defaults to 0 (the index of the first element)
- stop one past the last index of the slice. Defaults to len(iterable)
- step is the step size (better explained by the examples below)

Examples:

```
a = "abcdef"
a           # "abcdef"
          # Same as a[:] or a[::] since it uses the defaults for all three indices
a[-1]      # "f"
a[:]       # "abcdef"
a[::]     # "abcdef"
a[3:]      # "def" (from index 3, to end(defaults to size of iterable))
a[:4]      # "abcd" (from beginning(default 0) to position 4 (excluded))
a[2:4]     # "cd" (from position 2, to position 4 (excluded))
```

In addition, any of the above can be used with the step size defined:

```
a[::2]      # "ace" (every 2nd element)
a[1:4:2]    # "bd" (from index 1, to index 4 (excluded), every 2nd element)
```

Indices can be negative, in which case they're computed from the end of the sequence

```
a[:-1]     # "abcde" (from index 0 (default), to the second last element (last element - 1))
a[:-2]     # "abcd" (from index 0 (default), to the third last element (last element - 2))
a[-1:]     # "f" (from the last element to the end (default len()))
```

Step sizes can also be negative, in which case slice will iterate through the list in reverse order:

```
a[3:1:-1]  # "dc" (from index 2 to None (default), in reverse order)
```

This construct is useful for reversing an iterable

```
a[::-1]    # "fedcba" (from last element (default len()-1), to first, in reverse order(-1))
```

请注意，对于负步长，默认的end\_index是None (参见<http://stackoverflow.com/a/12521981>)

```
a[5:None:-1] # "fedcba" (这等同于a[::-1])  
a[5:0:-1]    # "fedcb" (从最后一个元素 (索引5) 到第二个元素 (索引1))
```

## 第64.2节：反转对象

你可以使用切片非常轻松地反转一个str、list或tuple（或者基本上任何实现了带步长参数切片的集合对象）。下面是一个反转字符串的例子，虽然这同样适用于上述其他类型：

```
s = 'reverse me!'  
s[::-1]    # '!em esrever'
```

我们快速看一下语法。[::-1] 表示切片应从字符串的开始到结束（因为start和end被省略了），步长为-1意味着它应该以反向遍历字符串。

## 第64.3节：切片赋值

使用切片的另一个巧妙功能是切片赋值。Python允许你通过单个操作为列表替换旧的切片赋予新的切片。

这意味着如果你有一个列表，你可以在一次赋值中替换多个成员：

```
lst = [1, 2, 3]  
lst[1:3] = [4, 5]  
print(lst) # 输出: [1, 4, 5]
```

赋值的大小也不必匹配，因此如果你想用一个大小不同的新切片替换旧切片，可以这样做：

```
lst = [1, 2, 3, 4, 5]  
lst[1:4] = [6]  
print(lst) # 输出: [1, 6, 5]
```

也可以使用已知的切片语法来做诸如替换整个列表的操作：

```
lst = [1, 2, 3]  
lst[:] = [4, 5, 6]  
print(lst) # 输出: [4, 5, 6]
```

或者仅替换最后两个元素：

```
lst = [1, 2, 3]  
lst[-2:] = [4, 5, 6]  
print(lst) # 输出: [1, 4, 5, 6]
```

## 第64.4节：制作数组的浅拷贝

快速复制数组（与将变量赋值为原数组的另一个引用不同）的方法是：

```
arr[:]
```

Notice that for negative steps the default end\_index is None (see <http://stackoverflow.com/a/12521981>)

```
a[5:None:-1] # "fedcba" (this is equivalent to a[::-1])  
a[5:0:-1]    # "fedcb" (from the last element (index 5) to second element (index 1))
```

## Section 64.2: Reversing an object

You can use slices to very easily reverse a str, list, or tuple (or basically any collection object that implements slicing with the step parameter). Here is an example of reversing a string, although this applies equally to the other types listed above:

```
s = 'reverse me!'  
s[::-1]    # '!em esrever'
```

Let's quickly look at the syntax.[::-1] means that the slice should be from the beginning until the end of the string (because start and end are omitted) and a step of -1 means that it should move through the string in reverse.

## Section 64.3: Slice assignment

Another neat feature using slices is slice assignment. Python allows you to assign new slices to replace old slices of a list in a single operation.

This means that if you have a list, you can replace multiple members in a single assignment:

```
lst = [1, 2, 3]  
lst[1:3] = [4, 5]  
print(lst) # Out: [1, 4, 5]
```

The assignment shouldn't match in size as well, so if you wanted to replace an old slice with a new slice that is different in size, you could:

```
lst = [1, 2, 3, 4, 5]  
lst[1:4] = [6]  
print(lst) # Out: [1, 6, 5]
```

It's also possible to use the known slicing syntax to do things like replacing the entire list:

```
lst = [1, 2, 3]  
lst[:] = [4, 5, 6]  
print(lst) # Out: [4, 5, 6]
```

Or just the last two members:

```
lst = [1, 2, 3]  
lst[-2:] = [4, 5, 6]  
print(lst) # Out: [1, 4, 5, 6]
```

## Section 64.4: Making a shallow copy of an array

A quick way to make a copy of an array (as opposed to assigning a variable with another reference to the original array) is:

```
arr[:]
```

让我们来看看语法。[:] 表示 start、end 和 slice 都被省略。它们默认分别为 0、len(arr) 和 1，这意味着我们请求的子数组将包含 arr 中从开始到结束的所有元素。

实际操作看起来像这样：

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr)    # ['a', 'b', 'c', 'd']
print(copy)   # ['a', 'b', 'c']
```

如你所见，arr.append('d') 向 arr 添加了 d，但 copy 保持不变！

注意，这只是一个浅拷贝，与 arr.copy() 完全相同。

## 第64.5节：自定义类的索引：\_\_getitem\_\_、\_\_setitem\_\_ 和 \_\_delitem\_\_

```
class MultiIndexingList:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return repr(self.value)

    def __getitem__(self, item):
        if isinstance(item, (int, slice)):
            return self.__class__(self.value[item])
        return [self.value[i] for i in item]

    def __setitem__(self, item, value):
        if isinstance(item, int):
            self.value[item] = value
        elif isinstance(item, slice):
            raise ValueError('Cannot interpret slice with multiindexing')
        else:
            for i in item:
                if isinstance(i, slice):
                    raise ValueError('Cannot interpret slice with multiindexing')
                self.value[i] = value

    def __delitem__(self, item):
        if isinstance(item, int):
            del self.value[item]
        elif isinstance(item, slice):
            del self.value[item]
        else:
            if any(isinstance(elem, slice) for elem in item):
                raise ValueError('Cannot interpret slice with multiindexing')
            item = sorted(item, reverse=True)
            for elem in item:
                del self.value[elem]
```

这允许通过切片和索引来访问元素：

```
a = MultiIndexingList([1, 2, 3, 4, 5, 6, 7, 8])
a
# 输出: [1, 2, 3, 4, 5, 6, 7, 8]
```

Let's examine the syntax. [:] means that start, end, and slice are all omitted. They default to 0, len(arr), and 1, respectively, meaning that subarray that we are requesting will have all of the elements of arr from the beginning until the very end.

In practice, this looks something like:

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr)    # ['a', 'b', 'c', 'd']
print(copy)   # ['a', 'b', 'c']
```

As you can see, arr.append('d') added d to arr, but copy remained unchanged!

Note that this makes a *shallow* copy, and is identical to arr.copy().

## Section 64.5: Indexing custom classes: \_\_getitem\_\_, \_\_setitem\_\_ and \_\_delitem\_\_

```
class MultiIndexingList:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return repr(self.value)

    def __getitem__(self, item):
        if isinstance(item, (int, slice)):
            return self.__class__(self.value[item])
        return [self.value[i] for i in item]

    def __setitem__(self, item, value):
        if isinstance(item, int):
            self.value[item] = value
        elif isinstance(item, slice):
            raise ValueError('Cannot interpret slice with multiindexing')
        else:
            for i in item:
                if isinstance(i, slice):
                    raise ValueError('Cannot interpret slice with multiindexing')
                self.value[i] = value

    def __delitem__(self, item):
        if isinstance(item, int):
            del self.value[item]
        elif isinstance(item, slice):
            del self.value[item]
        else:
            if any(isinstance(elem, slice) for elem in item):
                raise ValueError('Cannot interpret slice with multiindexing')
            item = sorted(item, reverse=True)
            for elem in item:
                del self.value[elem]
```

This allows slicing and indexing for element access:

```
a = MultiIndexingList([1, 2, 3, 4, 5, 6, 7, 8])
a
# Out: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
a[1,5,2,6,1]
# 输出: [2, 6, 3, 7, 2]
a[4, 1, 5:, 2, ::2]
# 输出: [5, 2, [6, 7, 8], 3, [1, 3, 5, 7]]
# 4/1----50---|2/-----:2----- <-- 指示哪个元素来自哪个索引
```

设置和删除元素时只允许逗号分隔的整数索引（不支持切片）：

```
a[4] = 1000
a
# 输出: [1, 2, 3, 4, 1000, 6, 7, 8]
a[2,6,1] = 100
a
# 输出: [1, 100, 100, 4, 1000, 6, 100, 8]
del a[5]
a
# 输出: [1, 100, 100, 4, 1000, 100, 8]
del a[4,2,5]
a
# 输出: [1, 100, 4, 8]
```

## 第64.6节：基本索引

Python列表是基于0的即列表中的第一个元素可以通过索引0访问

```
arr = ['a', 'b', 'c', 'd']
print(arr[0])
>> 'a'
```

您可以通过索引1访问列表中的第二个元素，通过索引2访问第三个元素，依此类推：

```
print(arr[1])
>> 'b'
print(arr[2])
>> 'c'
```

你也可以使用负索引从列表末尾访问元素。例如，索引-1将返回列表的最后一个元素，索引-2将返回倒数第二个元素：

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

如果你尝试访问列表中不存在的索引，将会引发`IndexError`错误：

```
print arr[6]
回溯 (最近一次调用最后) :
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
a[1, 5, 2, 6, 1]
# Out: [2, 6, 3, 7, 2]
a[4, 1, 5:, 2, ::2]
# Out: [5, 2, [6, 7, 8], 3, [1, 3, 5, 7]]
# 4/1----50---|2/-----:2----- <-- indicated which element came from which index
```

While setting and deleting elements only allows for *comma separated* integer indexing (no slicing):

```
a[4] = 1000
a
# Out: [1, 2, 3, 4, 1000, 6, 7, 8]
a[2,6,1] = 100
a
# Out: [1, 100, 100, 4, 1000, 6, 100, 8]
del a[5]
a
# Out: [1, 100, 100, 4, 1000, 100, 8]
del a[4,2,5]
a
# Out: [1, 100, 4, 8]
```

## Section 64.6: Basic Indexing

Python lists are 0-based i.e. the first element in the list can be accessed by the index 0

```
arr = ['a', 'b', 'c', 'd']
print(arr[0])
>> 'a'
```

You can access the second element in the list by index 1, third element by index 2 and so on:

```
print(arr[1])
>> 'b'
print(arr[2])
>> 'c'
```

You can also use negative indices to access elements from the end of the list. eg. index -1 will give you the last element of the list and index -2 will give you the second-to-last element of the list:

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

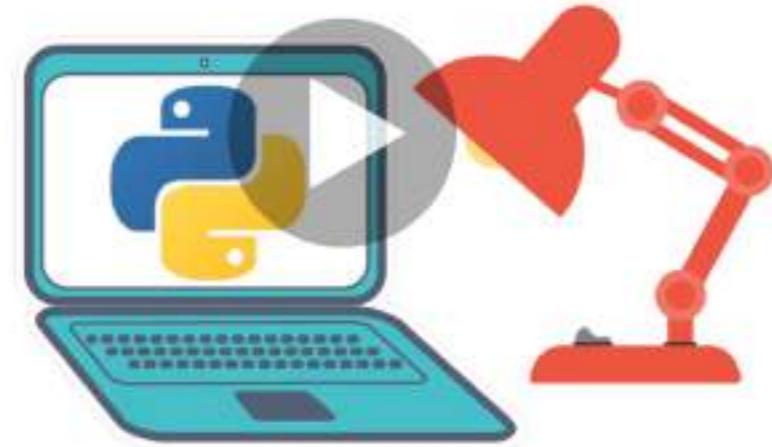
If you try to access an index which is not present in the list, an `IndexError` will be raised:

```
print arr[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

# 视频：完整的Python 训练营：从零开始 成为Python 3高手

像专业人士一样学习Python！从基础开始，直到  
创建你自己的应用程序和游戏！

## COMPLETE PYTHON 3 BOOTCAMP



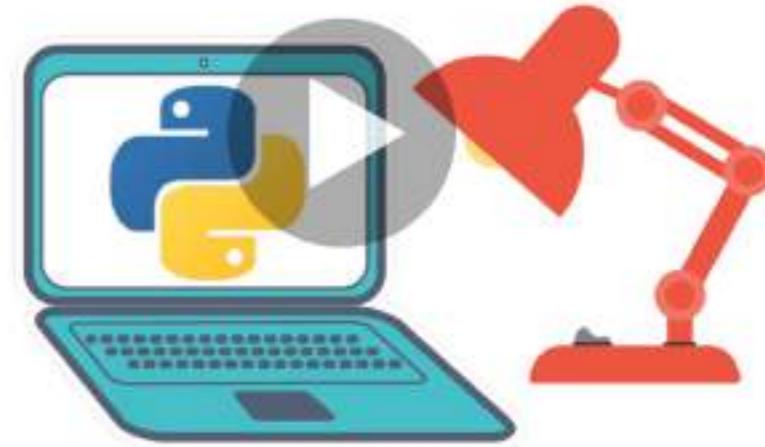
- ✓ 学习专业使用Python，掌握Python 2和Python 3！
- ✓ 用Python创建游戏，比如井字棋和二十一点！
- ✓ 学习高级Python功能，如collections模块和时间戳的使用！
- ✓ 学习使用面向对象编程和类！
- ✓ 理解复杂主题，如装饰器。
- ✓ 理解如何使用Jupyter Notebook并创建.py文件
- ✓ 了解如何在Jupyter Notebook系统中创建图形用户界面（GUI）！
- ✓ 从零开始全面理解Python！

今天观看 →

# VIDEO: Complete Python Bootcamp: Go from zero to hero in Python 3

Learn Python like a Professional! Start from the basics and go all the way to creating your own applications and games!

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ Learn to use Python professionally, learning both Python 2 and Python 3!
- ✓ Create games with Python, like Tic Tac Toe and Blackjack!
- ✓ Learn advanced Python features, like the collections module and how to work with timestamps!
- ✓ Learn to use Object Oriented Programming with classes!
- ✓ Understand complex topics, like decorators.
- ✓ Understand how to use both the Jupyter Notebook and create .py files
- ✓ Get an understanding of how to create GUIs in the Jupyter Notebook system!
- ✓ Build a complete understanding of Python from the ground up!

Watch Today →

# 第65章：使用Matplotlib绘图

Matplotlib (<https://matplotlib.org/>) 是一个基于NumPy的二维绘图库。以下是一些基本示例。  
更多示例可以在官方文档中找到 (<https://matplotlib.org/2.0.2/gallery.html> 和 <https://matplotlib.org/2.0.2/examples/index.html>)

## 第65.1节：具有公共X轴但不同Y轴的图表： 使用 `twinx()`

在本例中，我们将在同一图表中绘制正弦曲线和双曲正弦曲线，具有公共的x轴但不同的y轴。这是通过使用`twinx()`命令实现的。

```
# Python中的绘图教程  
# 通过双x轴添加多个图形  
# 适用于具有不同y轴范围的图形  
# 分离坐标轴和图形对象  
# 复制坐标轴对象并绘制曲线  
# 使用坐标轴设置属性  
  
# 注意：  
# 第二条曲线的网格未成功：如果你发现了，请告诉我！:(  
  
import numpy as np  
import matplotlib.pyplot as plt  
  
x = np.linspace(0, 2.0*np.pi, 101)  
y = np.sin(x)  
z = np.sinh(x)  
  
# 将图形对象和坐标轴对象与绘图对象分离  
fig, ax1 = plt.subplots()  
  
# 复制坐标轴，使用不同的y轴  
# 但相同的x轴  
ax2 = ax1.twinx() # ax2和ax1将共享x轴但y轴不同  
  
# 在坐标轴1和2上绘制曲线，并获取曲线句柄  
curve1, = ax1.plot(x, y, label="sin", color='r')  
curve2, = ax2.plot(x, z, label="sinh", color='b')  
  
# 创建曲线列表以访问曲线参数  
curves = [curve1, curve2]  
  
# 通过坐标轴1或坐标轴2对象添加图例。  
# 通常一个命令就足够了  
# ax1.legend() # 不会显示ax2的图例  
# ax2.legend() # 不会显示ax1的图例  
ax1.legend(curves, [curve.get_label() for curve in curves])  
# ax2.legend(curves, [curve.get_label() for curve in curves]) # 也有效  
  
# 全局图形属性  
plt.title("正弦和双曲正弦函数图")  
plt.show()
```

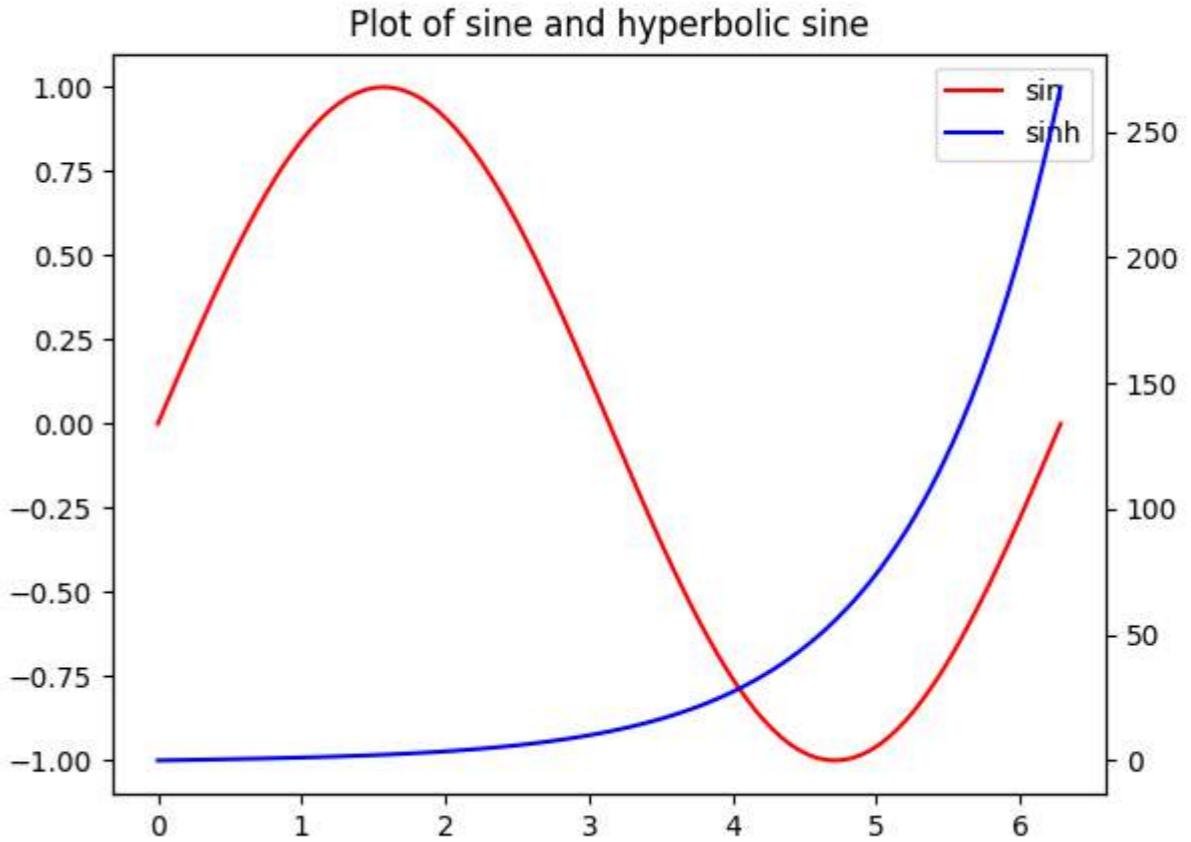
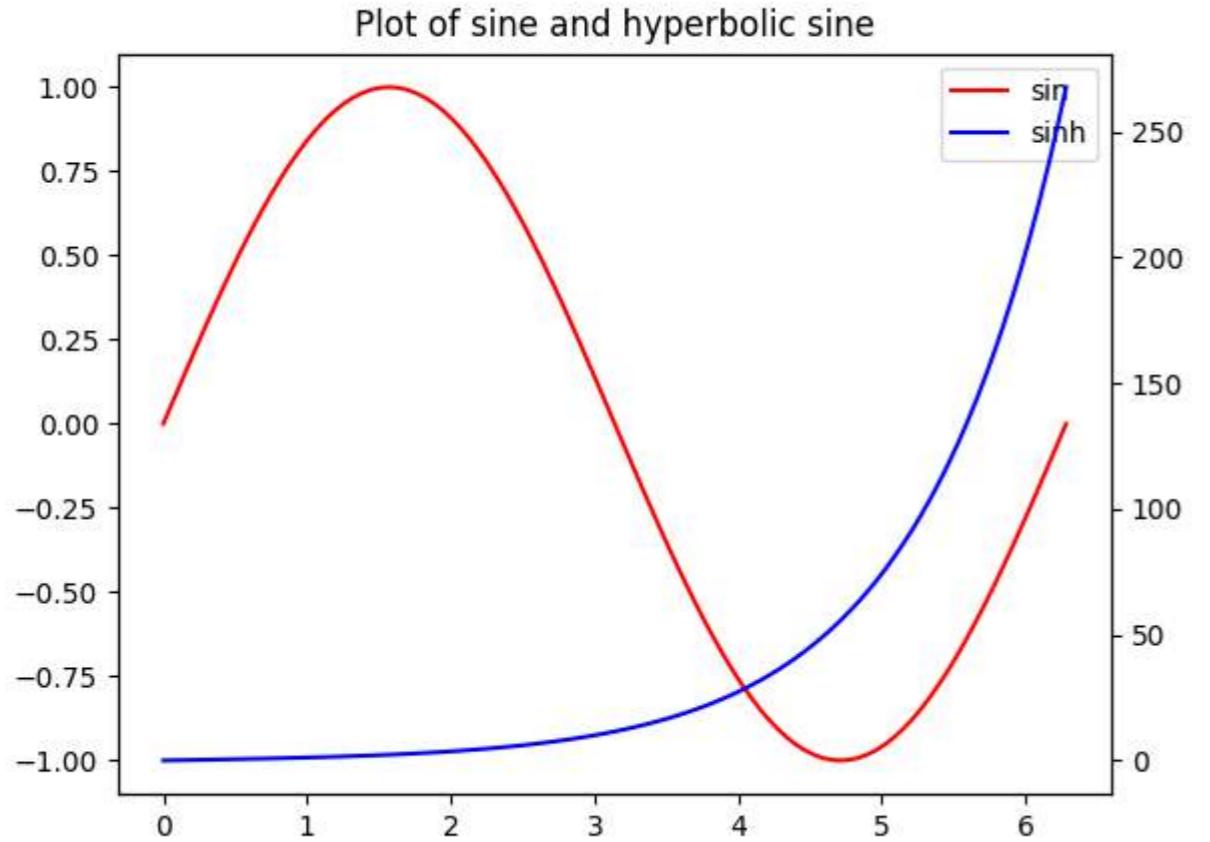
# Chapter 65: Plotting with Matplotlib

Matplotlib (<https://matplotlib.org/>) is a library for 2D plotting based on NumPy. Here are some basic examples.  
More examples can be found in the official documentation (<https://matplotlib.org/2.0.2/gallery.html> and <https://matplotlib.org/2.0.2/examples/index.html>)

## Section 65.1: Plots with Common X-axis but different Y-axis : Using `twinx()`

In this example, we will plot a sine curve and a hyperbolic sine curve in the same plot with a common x-axis having different y-axis. This is accomplished by the use of `twinx()` command.

```
# Plotting tutorials in Python  
# Adding Multiple plots by twin x axis  
# Good for plots having different y axis range  
# Separate axes and figure objects  
# replicate axes object and plot curves  
# use axes to set attributes  
  
# Note:  
# Grid for second curve unsuccessful : let me know if you find it! :(  
  
import numpy as np  
import matplotlib.pyplot as plt  
  
x = np.linspace(0, 2.0*np.pi, 101)  
y = np.sin(x)  
z = np.sinh(x)  
  
# separate the figure object and axes object  
# from the plotting object  
fig, ax1 = plt.subplots()  
  
# Duplicate the axes with a different y axis  
# and the same x axis  
ax2 = ax1.twinx() # ax2 and ax1 will have common x axis and different y axis  
  
# plot the curves on axes 1, and 2, and get the curve handles  
curve1, = ax1.plot(x, y, label="sin", color='r')  
curve2, = ax2.plot(x, z, label="sinh", color='b')  
  
# Make a curves list to access the parameters in the curves  
curves = [curve1, curve2]  
  
# add legend via axes 1 or axes 2 object.  
# one command is usually sufficient  
# ax1.legend() # will not display the legend of ax2  
# ax2.legend() # will not display the legend of ax1  
ax1.legend(curves, [curve.get_label() for curve in curves])  
# ax2.legend(curves, [curve.get_label() for curve in curves]) # also valid  
  
# Global figure properties  
plt.title("Plot of sine and hyperbolic sine")  
plt.show()
```



## 第65.2节：使用twiny()绘制具有相同Y轴但不同X轴的图

本例演示了使用 `twiny()`方法绘制具有相同y轴但不同x轴的曲线图。同时，图中还添加了标题、图例、标签、网格、坐标轴刻度和颜色等附加功能。

```
# Python绘图教程
# 通过双y轴添加多个图形
# 适用于具有不同x轴范围的图形
# 分离坐标轴和图形对象
# 复制坐标轴对象并绘制曲线
# 使用坐标轴设置属性

import numpy as np
import matplotlib.pyplot as plt

y = np.linspace(0, 2.0*np.pi, 101)
x1 = np.sin(y)
x2 = np.sinh(y)

# 用于设置x轴和y轴刻度的数值
ynumbers = np.linspace(0, 7, 15)
xnumbers1 = np.linspace(-1, 1, 11)
xnumbers2 = np.linspace(0, 300, 7)

# 将图形对象和坐标轴对象与绘图对象分离
fig, ax1 = plt.subplots()
```

## Section 65.2: Plots with common Y-axis and different X-axis using `twiny()`

In this example, a plot with curves having common y-axis but different x axis is demonstrated using `twiny()` method. Also, some additional features such as the title, legend, labels, grids, axis ticks and colours are added to the plot.

```
# Plotting tutorials in Python
# Adding Multiple plots by twin y axis
# Good for plots having different x axis range
# Separate axes and figure objects
# replicate axes object and plot curves
# use axes to set attributes

import numpy as np
import matplotlib.pyplot as plt

y = np.linspace(0, 2.0*np.pi, 101)
x1 = np.sin(y)
x2 = np.sinh(y)

# values for making ticks in x and y axis
ynumbers = np.linspace(0, 7, 15)
xnumbers1 = np.linspace(-1, 1, 11)
xnumbers2 = np.linspace(0, 300, 7)

# separate the figure object and axes object
# from the plotting object
fig, ax1 = plt.subplots()
```

```

# 复制坐标轴，使用不同的x轴
# 但相同的y轴
ax2 = ax1.twiny() # ax2 和 ax1 将共享相同的 y 轴但拥有不同的 x 轴

# 在坐标轴 1 和 2 上绘制曲线，并获取坐标轴句柄
curve1, = ax1.plot(x1, y, label="sin", color='r')
curve2, = ax2.plot(x2, y, label="sinh", color='b')

# 创建曲线列表以访问曲线参数
curves = [curve1, curve2]

# 通过坐标轴1或坐标轴2对象添加图例。
# 一条命令通常就足够了
# ax1.legend() # 不会显示 ax2 的图例
# ax2.legend() # 不会显示 ax1 的图例
# ax1.legend(curves, [curve.get_label() for curve in curves])
ax2.legend(curves, [curve.get_label() for curve in curves]) # 也有效

# 通过坐标轴设置 x 轴标签
ax1.set_xlabel("幅度", color=curve1.get_color())
ax2.set_xlabel("幅度", color=curve2.get_color())

# 通过坐标轴设置 y 轴标签
ax1.set_ylabel("角度/数值", color=curve1.get_color())
# ax2.set_ylabel("Magnitude", color=curve2.get_color()) # 不起作用
# ax2 没有对 y 轴的属性控制

# y 轴刻度 - 也使其带颜色
ax1.tick_params(axis='y', colors=curve1.get_color())
# ax2.tick_params(axis='y', colors=curve2.get_color()) # 不起作用
# ax2 没有对 y 轴的属性控制

# 通过坐标轴设置x轴刻度
ax1.tick_params(axis='x', colors=curve1.get_color())
ax2.tick_params(axis='x', colors=curve2.get_color())

# 设置x轴刻度
ax1.set_xticks(xnumbers1)
ax2.set_xticks(xnumbers2)

# 设置y轴刻度
ax1.set_yticks(ynumbers)
# ax2.set_yticks(ynumbers) # 也适用

# 通过坐标轴1设置网格 # 如果使用坐标轴1来
# 定义公共x轴的属性，则使用此方法
# ax1.grid(color=curve1.get_color())

# 使用坐标轴2绘制网格
ax1.grid(color=curve2.get_color())
ax2.grid(color=curve2.get_color())
ax1.xaxis.grid(False)

# 全局图形属性
plt.title("正弦和双曲正弦函数图")
plt.show()

```

```

# Duplicate the axes with a different x axis
# and the same y axis
ax2 = ax1.twiny() # ax2 and ax1 will have common y axis and different x axis

# plot the curves on axes 1, and 2, and get the axes handles
curve1, = ax1.plot(x1, y, label="sin", color='r')
curve2, = ax2.plot(x2, y, label="sinh", color='b')

# Make a curves list to access the parameters in the curves
curves = [curve1, curve2]

# add legend via axes 1 or axes 2 object.
# one command is usually sufficient
# ax1.legend() # will not display the legend of ax2
# ax2.legend() # will not display the legend of ax1
# ax1.legend(curves, [curve.get_label() for curve in curves])
ax2.legend(curves, [curve.get_label() for curve in curves]) # also valid

# x axis labels via the axes
ax1.set_xlabel("Magnitude", color=curve1.get_color())
ax2.set_xlabel("Magnitude", color=curve2.get_color())

# y axis label via the axes
ax1.set_ylabel("Angle/Value", color=curve1.get_color())
# ax2.set_ylabel("Magnitude", color=curve2.get_color()) # does not work
# ax2 has no property control over y axis

# y ticks - make them coloured as well
ax1.tick_params(axis='y', colors=curve1.get_color())
# ax2.tick_params(axis='y', colors=curve2.get_color()) # does not work
# ax2 has no property control over y axis

# x axis ticks via the axes
ax1.tick_params(axis='x', colors=curve1.get_color())
ax2.tick_params(axis='x', colors=curve2.get_color())

# set x ticks
ax1.set_xticks(xnumbers1)
ax2.set_xticks(xnumbers2)

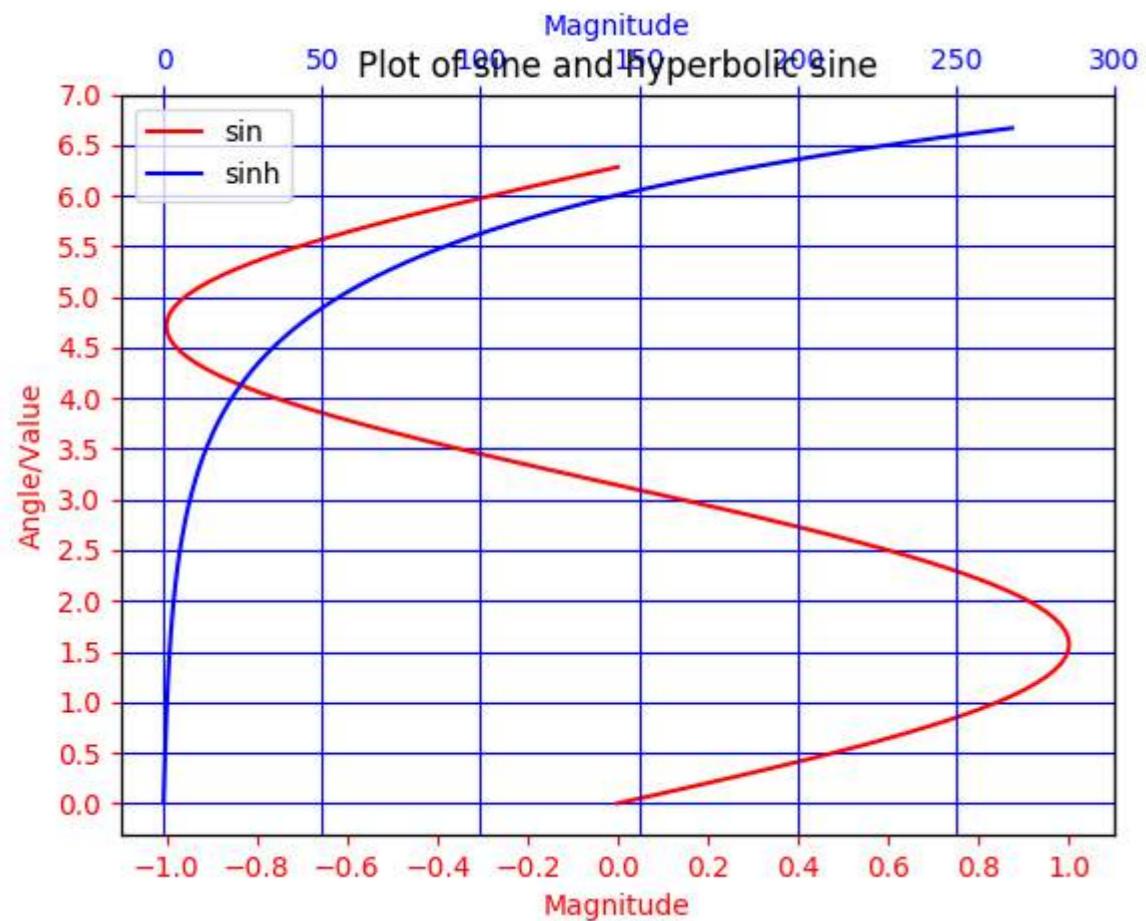
# set y ticks
ax1.set_yticks(ynumbers)
# ax2.set_yticks(ynumbers) # also works

# Grids via axes 1 # use this if axes 1 is used to
# define the properties of common x axis
# ax1.grid(color=curve1.get_color())

# To make grids using axes 2
ax1.grid(color=curve2.get_color())
ax2.grid(color=curve2.get_color())
ax1.xaxis.grid(False)

# Global figure properties
plt.title("Plot of sine and hyperbolic sine")
plt.show()

```



## 第65.3节：Matplotlib中的简单绘图

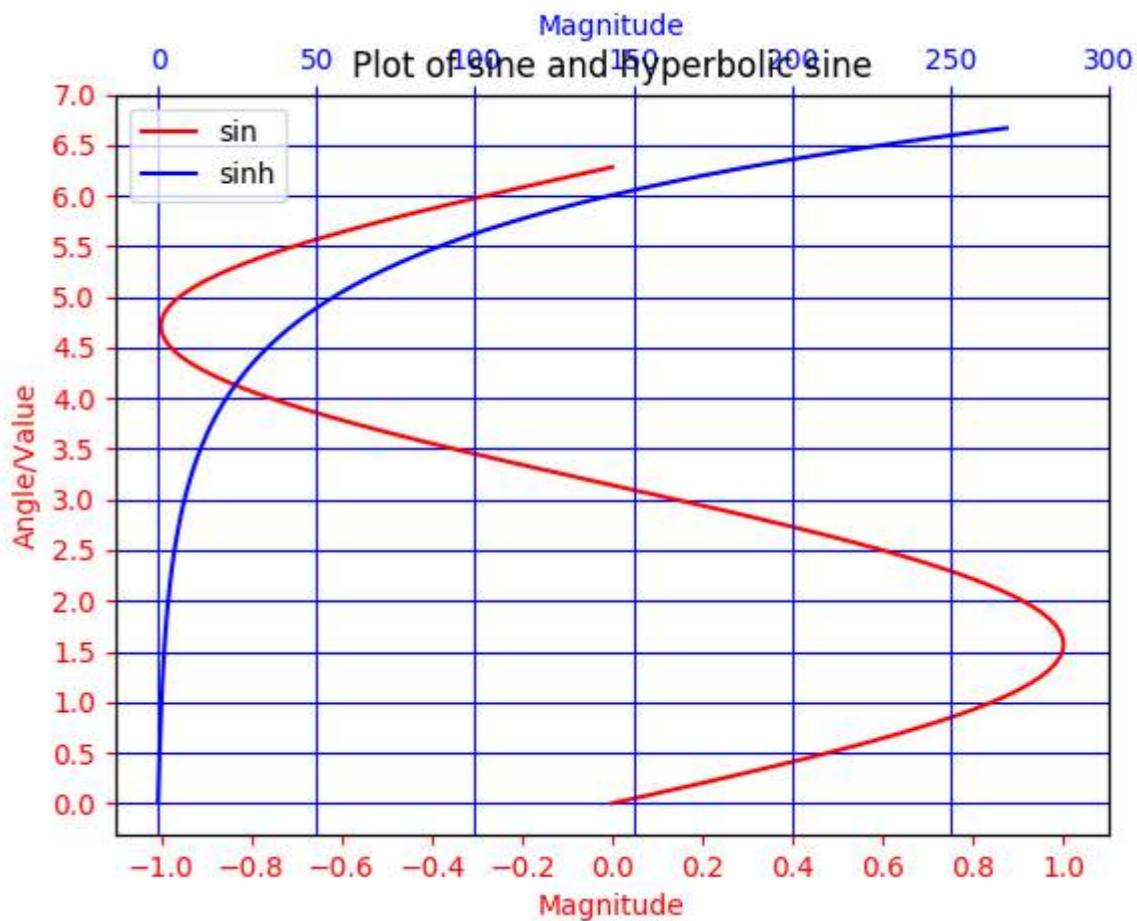
此示例演示如何使用Matplotlib创建一个简单的正弦曲线

```
# Python绘图教程
# 启动一个简单的绘图

import numpy 作为 np
import matplotlib.pyplot 作为 plt

# 角度在0到2*pi之间变化
x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)      # 正弦函数

plt.plot(x, y)
plt.show()
```



## Section 65.3: A Simple Plot in Matplotlib

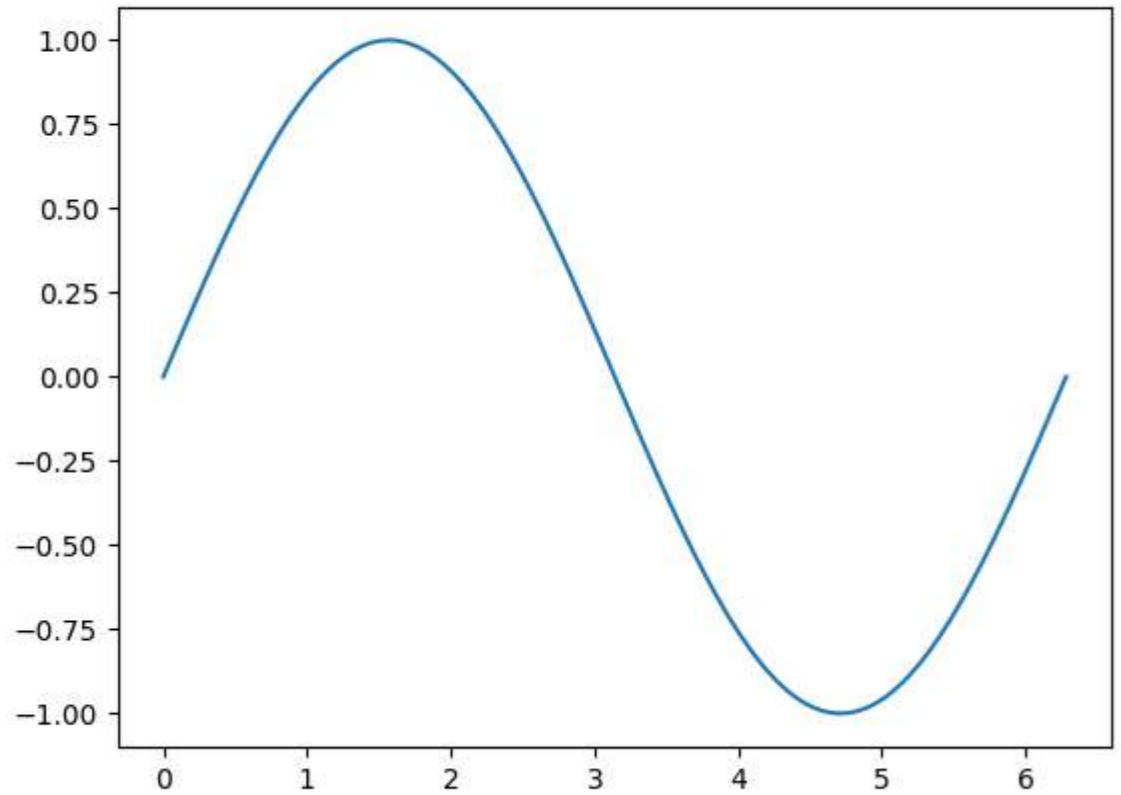
This example illustrates how to create a simple sine curve using **Matplotlib**

```
# Plotting tutorials in Python
# Launching a simple plot

import numpy as np
import matplotlib.pyplot as plt

# angle varying between 0 and 2*pi
x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)      # sine function

plt.plot(x, y)
plt.show()
```



## 第65.4节：为简单绘图添加更多功能：坐标轴标签、标题、坐标轴刻度、网格和图例

在此示例中，我们以正弦曲线图为例，添加更多功能；即标题、坐标轴标签、标题、坐标轴刻度、网格和图例。

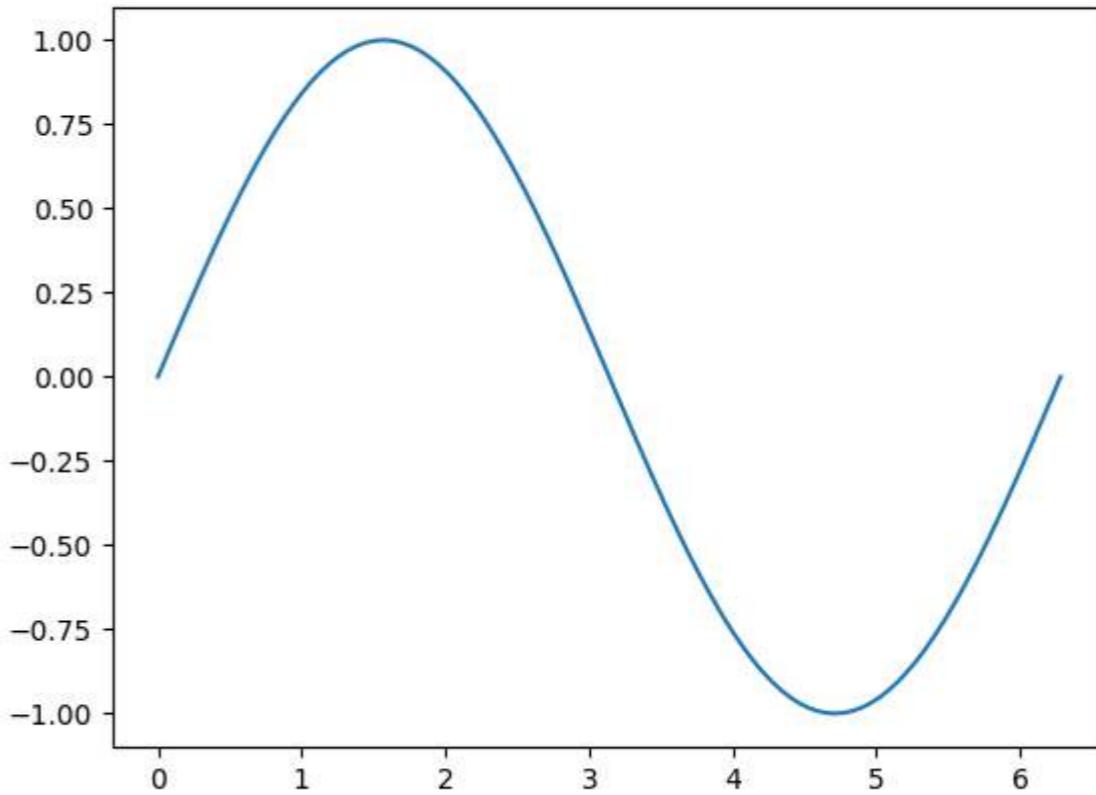
```
# Python绘图教程
# 增强绘图

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)

# 用于设置x轴和y轴刻度的数值
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - 红色
plt.xlabel("弧度角")
plt.ylabel("幅值")
plt.title("一些三角函数的图形")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [x起点, x终点, y起点, y终点]
plt.show()
```



## Section 65.4: Adding more features to a simple plot : axis labels, title, axis ticks, grid, and legend

In this example, we take a sine curve plot and add more features to it; namely the title, axis labels, title, axis ticks, grid and legend.

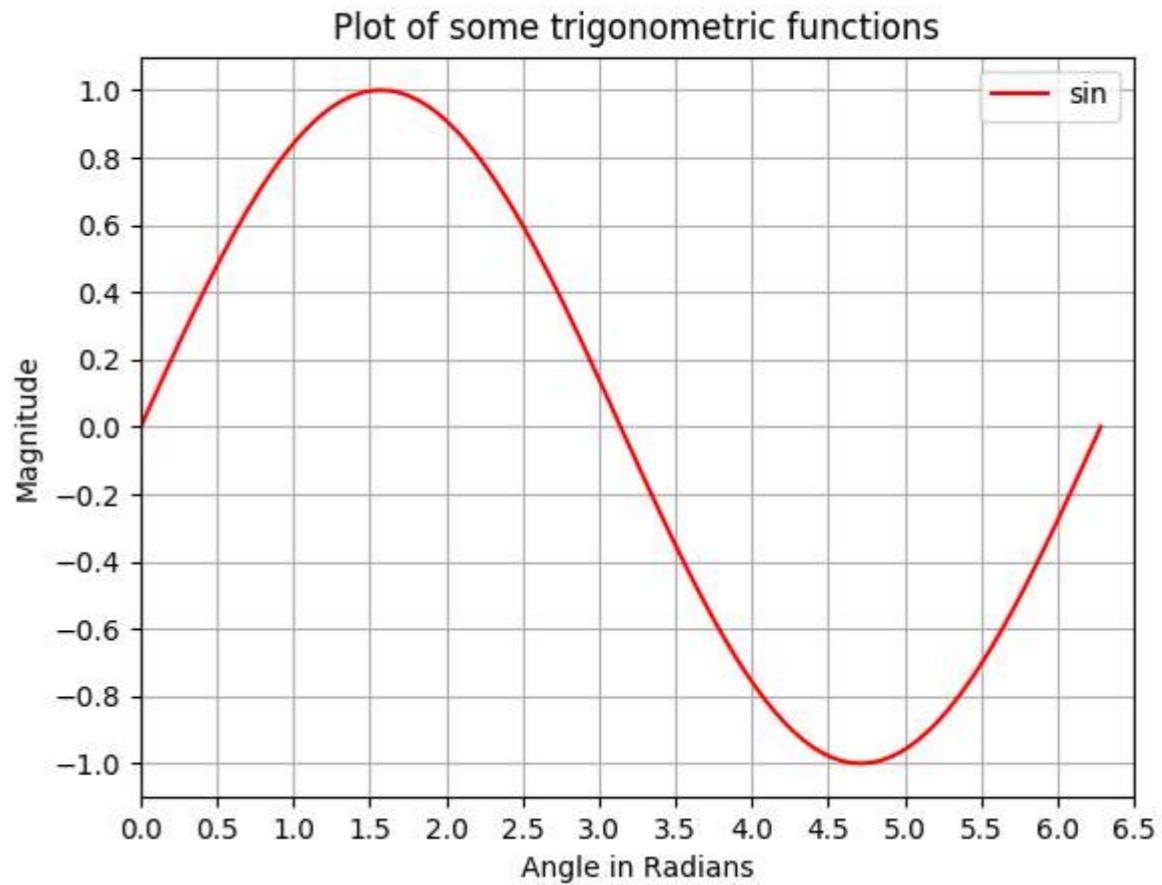
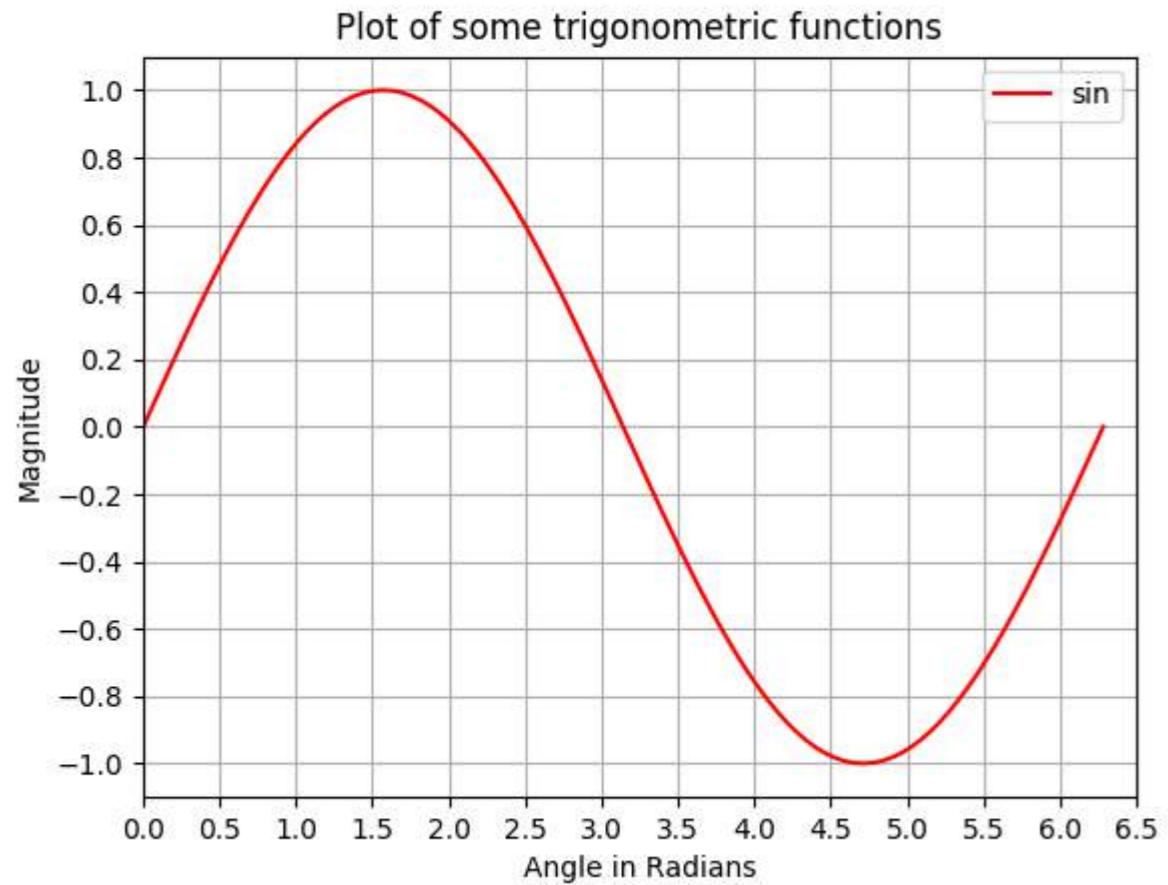
```
# Plotting tutorials in Python
# Enhancing a plot

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - red colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```



## 第65.5节：通过叠加在同一图形中绘制多个图形，类似MATLAB

在此示例中，通过将正弦曲线和余弦曲线叠加绘制在同一图形中。

```
# Python绘图教程
# 通过叠加添加多个图形
# 适用于共享相似x、y范围的图形
# 使用单个绘图命令和图例

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)

# 用于设置x轴和y轴刻度的数值
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, 'r', x, z, 'g') # r, g - 红色, 绿色
plt.xlabel("弧度角")
plt.ylabel("幅值")
plt.title("一些三角函数的图形")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend(['正弦', '余弦'])
plt.grid()
```

## Section 65.5: Making multiple plots in the same figure by superimposition similar to MATLAB

In this example, a sine curve and a cosine curve are plotted in the same figure by superimposing the plots on top of each other.

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using single plot command and legend

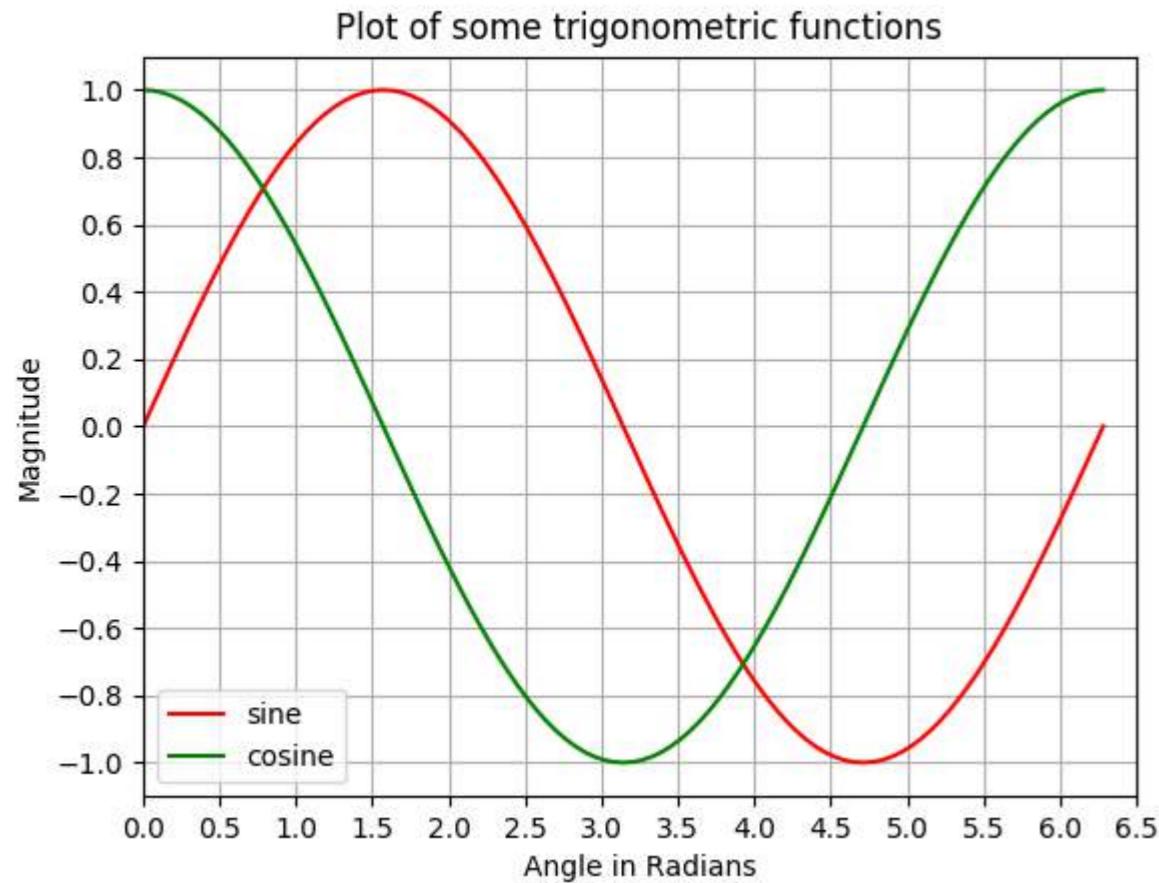
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)

# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, 'r', x, z, 'g') # r, g - red, green colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend(['sine', 'cosine'])
plt.grid()
```

```
plt.axis([0, 6.5, -1.1, 1.1]) # [x起点, x终点, y起点, y终点]
plt.show()
```



## 第65.6节：使用多个绘图命令在同一图形中通过叠加绘图制作多个图

类似于前面的例子，这里使用单独的绘图命令在同一图形上绘制了正弦曲线和余弦曲线。这种方法更符合Python风格，并且可以为每个绘图获取单独的句柄。

```
# Python中的绘图教程
# 通过叠加添加多个绘图
# 适用于共享相似x、y范围的绘图
# 使用多个绘图命令
# 比之前的方法好得多且更推荐

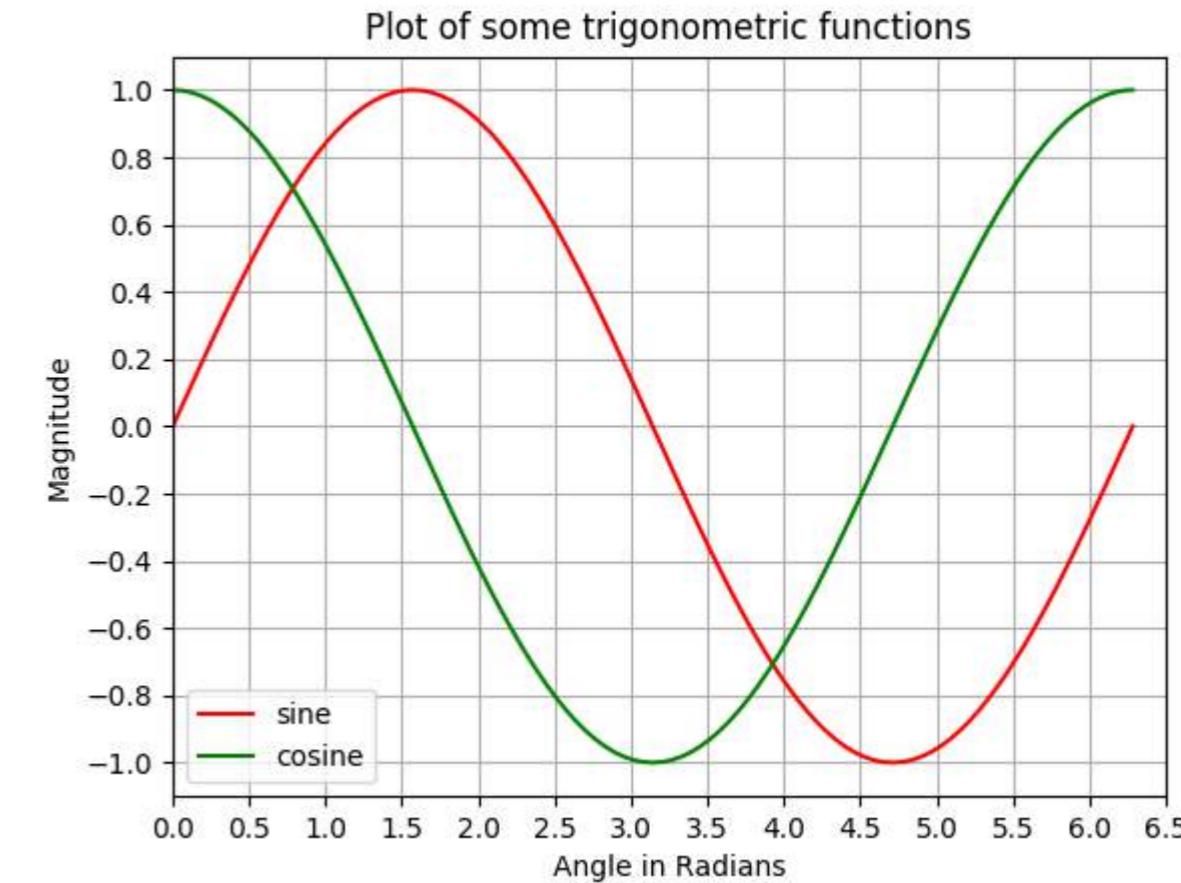
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)

# 用于设置x轴和y轴刻度的数值
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - 红色
plt.plot(x, z, color='g', label='cos') # g - 绿色
plt.xlabel("弧度角")
plt.ylabel("幅值")
```

```
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```



## Section 65.6: Making multiple Plots in the same figure using plot superimposition with separate plot commands

Similar to the previous example, here, a sine and a cosine curve are plotted on the same figure using separate plot commands. This is more Pythonic and can be used to get separate handles for each plot.

```
# Plotting tutorials in Python
# Adding Multiple plots by superimposition
# Good for plots sharing similar x, y limits
# Using multiple plot commands
# Much better and preferred than previous

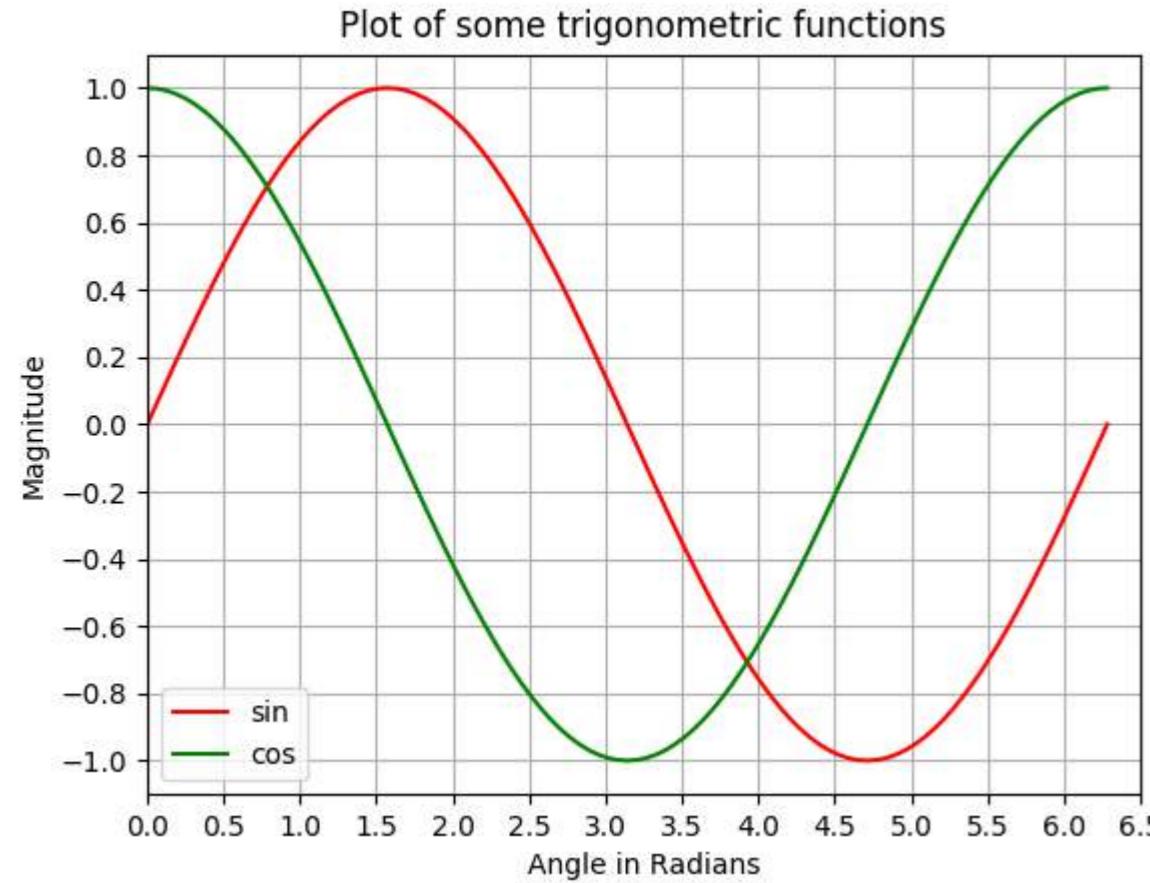
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)

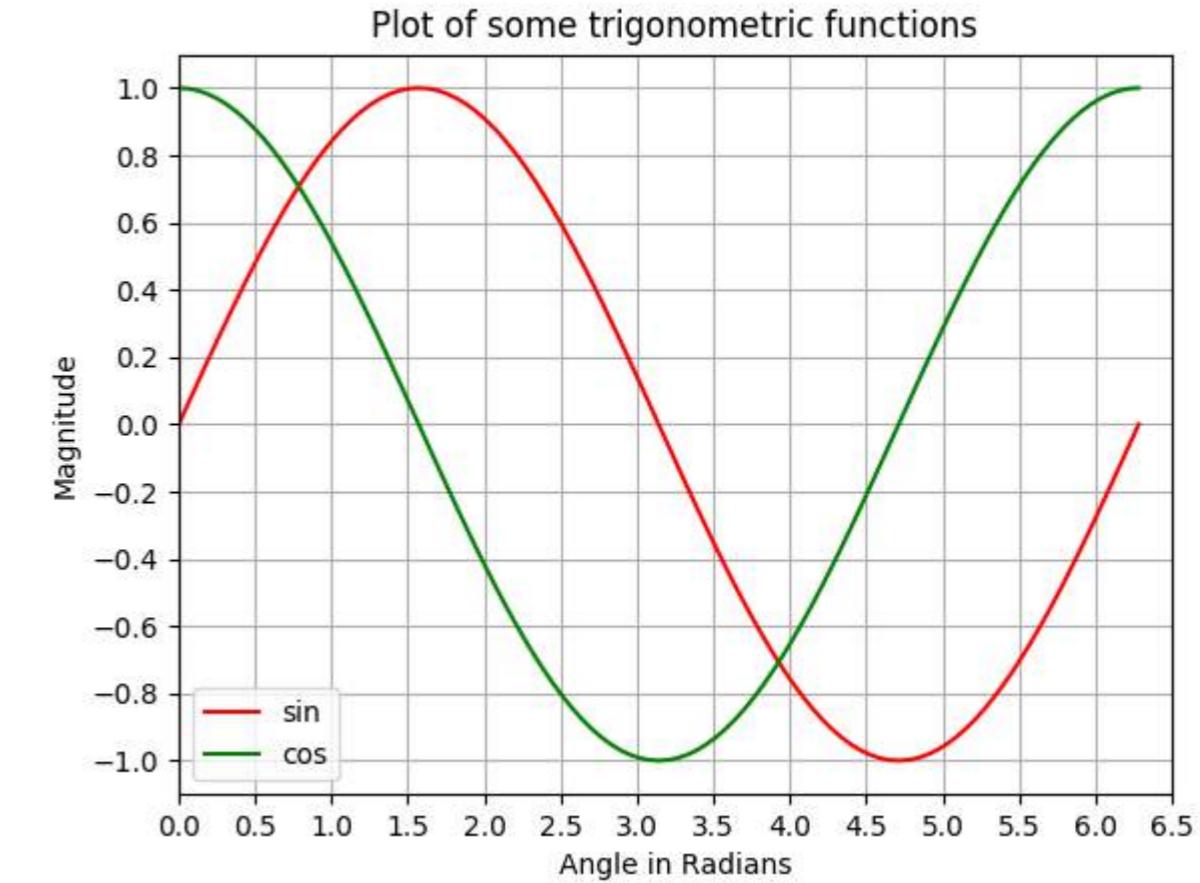
# values for making ticks in x and y axis
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - red colour
plt.plot(x, z, color='g', label='cos') # g - green colour
plt.xlabel("Angle in Radians")
plt.ylabel("Magnitude")
```

```
plt.title("一些三角函数的图形")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [x起点, x终点, y起点, y终点]
plt.show()
```



```
plt.title("Plot of some trigonometric functions")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```



# 第66章：graph-tool

Python工具可用于生成图形

## 第66.1节：PyDotPlus

PyDotPlus是旧pydot项目的改进版本，提供了Graphviz的Dot语言的Python接口。

### 安装

对于最新的稳定版本：

```
pip install pydotplus
```

对于开发版本：

```
pip install https://github.com/carlos-jenkins/pydotplus/archive/master.zip
```

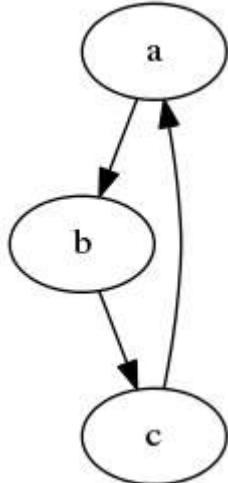
加载由DOT文件定义的图形

- 该文件被假定为DOT格式。它将被加载、解析并返回一个Dot类，表示该图形。例如，一个简单的demo.dot：

```
digraph demo1{ a -> b -> c; c ->a; }
```

```
import pydotplus
graph_a = pydotplus.graph_from_dot_file('demo.dot')
graph_a.write_svg('test.svg') # 生成svg格式的图形。
```

你将得到如下的svg（可缩放矢量图形）：



## 第66.2节：PyGraphviz

从Python软件包索引获取PyGraphviz，网址为<http://pypi.python.org/pypi/pygraphviz>

或者使用以下命令安装：

```
pip install pygraphviz
```

# Chapter 66: graph-tool

The python tools can be used to generate graph

## Section 66.1: PyDotPlus

PyDotPlus is an improved version of the old pydot project that provides a Python Interface to Graphviz's Dot language.

### Installation

For the latest stable version:

```
pip install pydotplus
```

For the development version:

```
pip install https://github.com/carlos-jenkins/pydotplus/archive/master.zip
```

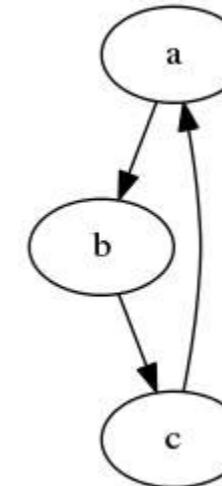
Load graph as defined by a DOT file

- The file is assumed to be in DOT format. It will be loaded, parsed and a Dot class will be returned, representing the graph. For example, a simple demo.dot:

```
digraph demo1{ a -> b -> c; c ->a; }
```

```
import pydotplus
graph_a = pydotplus.graph_from_dot_file('demo.dot')
graph_a.write_svg('test.svg') # generate graph in svg.
```

You will get a svg(Scalable Vector Graphics) like this:



## Section 66.2: PyGraphviz

Get PyGraphviz from the Python Package Index at <http://pypi.python.org/pypi/pygraphviz>

or install it with:

```
pip install pygraphviz
```

系统将尝试查找并安装与您的操作系统和  
Python版本匹配的合适版本。

您也可以通过以下命令安装开发版本（位于github.com）：

```
pip install git://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

从Python软件包索引获取PyGraphviz，网址为<http://pypi.python.org/pypi/pygraphviz>

或者使用以下命令安装：

```
easy_install pygraphviz
```

系统将尝试查找并安装与您的操作系统和

Python版本匹配的合适版本。

加载由DOT文件定义的图形

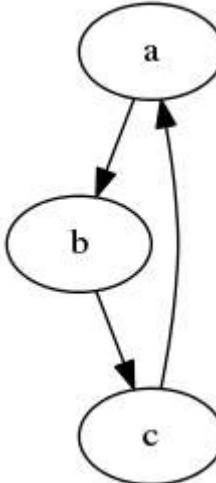
- 该文件假定为DOT格式。它将被加载、解析，并返回一个Dot类，表示该图。例如，一个简单的demo.dot：

```
digraph demo1{ a -> b -> c; c ->a; }
```

- 加载它并绘制。

```
import pygraphviz as pgv
G = pgv.AGraph("demo.dot")
G.draw('test', format='svg', prog='dot')
```

你将得到如下的svg（可缩放矢量图形）：



and an attempt will be made to find and install an appropriate version that matches your operating system and Python version.

You can install the development version (at github.com) with:

```
pip install git://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

Get PyGraphviz from the Python Package Index at <http://pypi.python.org/pypi/pygraphviz>

or install it with:

```
easy_install pygraphviz
```

and an attempt will be made to find and install an appropriate version that matches your operating system and Python version.

Load graph as defined by a DOT file

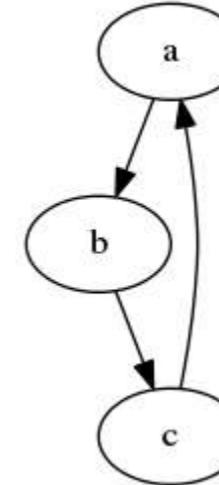
- The file is assumed to be in DOT format. It will be loaded, parsed and a Dot class will be returned, representing the graph. For example,a simple demo.dot:

```
digraph demo1{ a -> b -> c; c ->a; }
```

- Load it and draw it.

```
import pygraphviz as pgv
G = pgv.AGraph("demo.dot")
G.draw('test', format='svg', prog='dot')
```

You will get a svg(Scalable Vector Graphics) like this:



# 第67章：生成器

生成器是由生成器函数（使用`yield`）或生成器表达式（使用`(an_expression for x in an_iterator)`）创建的惰性迭代器。

## 第67.1节：介绍

生成器表达式类似于列表、字典和集合推导式，但用圆括号括起来。

当它们作为函数调用的唯一参数时，圆括号可以省略。

```
expression = (x**2 for x in range(10))
```

此示例生成前10个完全平方数，包括0（其中`x = 0`）。

生成器函数类似于普通函数，但其主体中包含一个或多个`yield`语句。此类函数不能`return`任何值（不过如果想提前停止生成器，可以使用空的`return`）。

```
def function():
    for x in range(10):
        yield x**2
```

这个生成器函数等价于之前的生成器表达式，输出相同。

注意：所有生成器表达式都有各自的等价函数，但反之不然。

如果两个括号会重复出现，生成器表达式可以不用括号：

```
sum(i for i in range(10) if i % 2 == 0)      #输出: 20
any(x = 0 for x in foo)                      #输出: True 或 False, 取决于 foo
type(a > b for a in foo if a % 2 == 1)       #输出: <class 'generator'>
```

代替：

```
sum((i for i in range(10) if i % 2 == 0))
any((x = 0 for x in foo))
type((a > b for a in foo if a % 2 == 1))
```

但不能：

```
fooFunction(i for i in range(10) if i % 2 == 0, foo, bar)
return x = 0 for x in foo
barFunction(baz, a > b for a in foo if a % 2 == 1)
```

调用生成器函数会产生一个**生成器对象**，之后可以对其进行迭代。与其他类型的迭代器不同，生成器对象只能被遍历一次。

```
g1 = function()
print(g1) # 输出: <generator object function at 0x1012e1888>
```

注意生成器的主体不会立即执行：当你调用上例中的`function()`时，它会立即返回一个生成器对象，而不会执行哪怕第一条`print`语句。这使得生成器比返回列表的函数消耗更少的内存，并且允许创建生成无限长序列的生成器。

# Chapter 67: Generators

Generators are lazy iterators created by generator functions (using `yield`) or generator expressions (using `(an_expression for x in an_iterator)`).

## Section 67.1: Introduction

**Generator expressions** are similar to list, dictionary and set comprehensions, but are enclosed with parentheses. The parentheses do not have to be present when they are used as the sole argument for a function call.

```
expression = (x**2 for x in range(10))
```

This example generates the 10 first perfect squares, including 0 (in which `x = 0`).

**Generator functions** are similar to regular functions, except that they have one or more `yield` statements in their body. Such functions cannot `return` any values (however empty `returns` are allowed if you want to stop the generator early).

```
def function():
    for x in range(10):
        yield x**2
```

This generator function is equivalent to the previous generator expression, it outputs the same.

**Note:** all generator expressions have their own *equivalent* functions, but not vice versa.

A generator expression can be used without parentheses if both parentheses would be repeated otherwise:

```
sum(i for i in range(10) if i % 2 == 0)      #Output: 20
any(x = 0 for x in foo)                      #Output: True or False depending on foo
type(a > b for a in foo if a % 2 == 1)       #Output: <class 'generator'>
```

Instead of:

```
sum((i for i in range(10) if i % 2 == 0))
any((x = 0 for x in foo))
type((a > b for a in foo if a % 2 == 1))
```

But not:

```
fooFunction(i for i in range(10) if i % 2 == 0, foo, bar)
return x = 0 for x in foo
barFunction(baz, a > b for a in foo if a % 2 == 1)
```

Calling a generator function produces a **generator object**, which can later be iterated over. Unlike other types of iterators, generator objects may only be traversed once.

```
g1 = function()
print(g1) # Out: <generator object function at 0x1012e1888>
```

Notice that a generator's body is **not** immediately executed: when you call `function()` in the example above, it immediately returns a generator object, without executing even the first `print` statement. This allows generators to consume less memory than functions that return a list, and it allows creating generators that produce infinitely long sequences.

因此，生成器常用于数据科学以及涉及大量数据的其他场景。另一个优点是其他代码可以立即使用生成器产生的值，而无需等待完整序列的生成。

但是，如果你需要多次使用生成器产生的值，且生成这些值的成本高于存储成本，那么将生成的值存储为列表可能比重新生成序列更好。详见下文“重置生成器”。

通常生成器对象用于循环中，或用于任何需要可迭代对象的函数：

```
对于 x 在 g1 中：  
print("Received", x)  
  
# 输出：  
# 接收到 0  
# 接收到 1  
# 接收到 4  
# 接收到 9  
# 接收到 16  
# 接收到 25  
# 接收到 36  
# 接收到 49  
# 接收到 64  
# 接收到 81  
  
arr1 = list(g1)  
# arr1 = [], 因为上面的循环已经消耗了所有的值。  
g2 = function()  
arr2 = list(g2) # arr2 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

由于生成器对象是迭代器，可以使用 `next()` 函数手动迭代它们。这样做会在每次调用时依次返回生成的值。

在底层，每次调用生成器的 `next()` 时，Python 会执行生成器函数体内的语句，直到遇到下一个 `yield` 语句。此时它返回 `yield` 命令的参数，并记住执行到的位置。再次调用 `next()` 会从该位置继续执行，直到遇到下一个 `yield` 语句。

如果 Python 在生成器函数末尾没有遇到更多的 `yield`，则会引发 `StopIteration` 异常（这是正常现象，所有迭代器的行为都是如此）。

```
g3 = function()  
a = next(g3) # a 变为 0  
b = next(g3) # b 变为 1  
c = next(g3) # c 变为 2  
...  
j = next(g3) # 引发 StopIteration, j 保持未定义
```

请注意，在 Python 2 中，生成器对象有 `.next()` 方法，可以用来手动迭代生成的值。在 Python 3 中，该方法被所有迭代器的标准方法 `__next__()` 所取代。

## 重置生成器

请记住，生成器生成的对象只能迭代一次。如果你已经在脚本中迭代过这些对象，任何进一步的尝试都会返回 `None`。

如果你需要多次使用生成器生成的对象，可以重新定义生成器

For this reason, generators are often used in data science, and other contexts involving large amounts of data. Another advantage is that other code can immediately use the values yielded by a generator, without waiting for the complete sequence to be produced.

However, if you need to use the values produced by a generator more than once, and if generating them costs more than storing, it may be better to store the yielded values as a `list` than to re-generate the sequence. See 'Resetting a generator' below for more details.

Typically a generator object is used in a loop, or in any function that requires an iterable:

```
for x in g1:  
    print("Received", x)  
  
# Output:  
# Received 0  
# Received 1  
# Received 4  
# Received 9  
# Received 16  
# Received 25  
# Received 36  
# Received 49  
# Received 64  
# Received 81  
  
arr1 = list(g1)  
# arr1 = [], because the loop above already consumed all the values.  
g2 = function()  
arr2 = list(g2) # arr2 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Since generator objects are iterators, one can iterate over them manually using the `next()` function. Doing so will return the yielded values one by one on each subsequent invocation.

Under the hood, each time you call `next()` on a generator, Python executes statements in the body of the generator function until it hits the next `yield` statement. At this point it returns the argument of the `yield` command, and remembers the point where that happened. Calling `next()` once again will resume execution from that point and continue until the next `yield` statement.

If Python reaches the end of the generator function without encountering any more `yields`, a `StopIteration` exception is raised (this is normal, all iterators behave in the same way).

```
g3 = function()  
a = next(g3) # a becomes 0  
b = next(g3) # b becomes 1  
c = next(g3) # c becomes 2  
...  
j = next(g3) # Raises StopIteration, j remains undefined
```

Note that in Python 2 generator objects had `.next()` methods that could be used to iterate through the yielded values manually. In Python 3 this method was replaced with the `__next__()` standard for all iterators.

## Resetting a generator

Remember that you can only iterate through the objects generated by a generator *once*. If you have already iterated through the objects in a script, any further attempt do so will yield `None`.

If you need to use the objects generated by a generator more than once, you can either define the generator

函数并第二次使用，或者，你也可以在第一次使用时将生成器函数的输出存储到列表中。重新定义生成器函数是处理大量数据时的好选择，因为存储所有数据项的列表会占用大量磁盘空间。相反，如果生成这些项的成本较高，你可能更愿意将生成的项存储在列表中以便重复使用。

## 第67.2节：无限序列

生成器可以用来表示无限序列：

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1

natural_numbers = integers_starting_from(1)
```

如上所示的无限数列也可以借助`itertools.count`生成。上述代码可以写成如下形式

```
natural_numbers = itertools.count(1)
```

你可以对无限生成器使用生成器推导式来生成新的生成器：

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

请注意，无限生成器没有终点，因此将其传递给任何试图完全消费该生成器的函数将会有严重后果：

```
list(multiples_of_two) # 将永远不会终止，或者引发特定操作系统的错误
```

相反，使用带有`range`（或`python 3.0`以下版本的`xrange`）的列表/集合推导式：

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
# [3, 6, 9, 12, 15]
```

或者使用`itertools.islice()`来切片迭代器以获取子集：

```
from itertools import islice
multiples_of_four = (x * 4 for x in integers_starting_from(1))
first_five_multiples_of_four = list(islice(multiples_of_four, 5))
# [4, 8, 12, 16, 20]
```

注意，原始生成器也会被更新，就像所有来自同一个“根”的其他生成器一样：

```
next(natural_numbers) # 生成 16
next(multiples_of_two) # 生成 34
next(multiples_of_four) # 生成 24
```

无限序列也可以用`for`循环迭代。请确保包含条件`break`语句，以便循环最终终止：

```
for idx, number in enumerate(multiples_of_two):
    print(number)
    if idx == 9:
```

function again and use it a second time, or, alternatively, you can store the output of the generator function in a list on first use. Re-defining the generator function will be a good option if you are dealing with large volumes of data, and storing a list of all data items would take up a lot of disc space. Conversely, if it is costly to generate the items initially, you may prefer to store the generated items in a list so that you can re-use them.

## Section 67.2: Infinite sequences

Generators can be used to represent infinite sequences:

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1

natural_numbers = integers_starting_from(1)
```

Infinite sequence of numbers as above can also be generated with the help of `itertools.count`. The above code could be written as below

```
natural_numbers = itertools.count(1)
```

You can use generator comprehensions on infinite generators to produce new generators:

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

Be aware that an infinite generator does not have an end, so passing it to any function that will attempt to consume the generator entirely will have **dire consequences**:

```
list(multiples_of_two) # will never terminate, or raise an OS-specific error
```

Instead, use list/set comprehensions with `range` (or `xrange` for python < 3.0):

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
# [3, 6, 9, 12, 15]
```

or use `itertools.islice()` to slice the iterator to a subset:

```
from itertools import islice
multiples_of_four = (x * 4 for x in integers_starting_from(1))
first_five_multiples_of_four = list(islice(multiples_of_four, 5))
# [4, 8, 12, 16, 20]
```

Note that the original generator is updated too, just like all other generators coming from the same "root":

```
next(natural_numbers) # yields 16
next(multiples_of_two) # yields 34
next(multiples_of_four) # yields 24
```

An infinite sequence can also be iterated with a `for`-loop. Make sure to include a conditional `break` statement so that the loop would terminate eventually:

```
for idx, number in enumerate(multiples_of_two):
    print(number)
    if idx == 9:
```

```
break # 在获取前10个2的倍数后停止
```

### 经典示例 - 斐波那契数列

```
import itertools

def fibonacci():
    a, b = 1, 1
    while True:
        yield a
    a, b = b, a + b

first_ten_fibs = list(itertools.islice(fibonacci(), 10))
# [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

def nth_fib(n):
    return next(itertools.islice(fibonacci(), n - 1, n))

ninety_nineth_fib = nth_fib(99) # 354224848179261915075
```

## 第67.3节：向生成器发送对象

除了从生成器接收值之外，还可以使用 `send()`

方法向生成器发送对象。

```
def accumulator():
    total = 0
    value = None
    while True:
        # 接收发送的值
        value = yield total
        if value is None: break
        # 汇总值
        total += value

generator = accumulator()

# 运行直到第一个"yield"
next(generator) # 0

# 从此处开始，生成器汇总值
generator.send(1) # 1
generator.send(10) # 11
generator.send(100) # 111
# ...

# 调用 next(generator) 等同于调用 generator.send(None)
next(generator) # StopIteration
```

这里发生的是以下情况：

- 当你第一次调用 `next(generator)` 时，程序会执行到第一个 `yield` 语句，并返回此时 `total` 的值，即 0。生成器的执行在此处暂停。
- 当你随后调用 `generator.send(x)` 时，解释器会将参数 `x` 作为上一个 `yield` 语句的返回值，该返回值被赋给 `value`。生成器随后照常继续执行，直到它产生下一个值。
- 当你最终调用 `next(generator)` 时，程序会将此视为向生成器发送 `None`。  
`None` 本身没有特殊含义，但此示例中使用 `None` 作为请求生成器停止的特殊值。

```
break # stop after taking the first 10 multiplies of two
```

### Classic example - Fibonacci numbers

```
import itertools

def fibonacci():
    a, b = 1, 1
    while True:
        yield a
    a, b = b, a + b

first_ten_fibs = list(itertools.islice(fibonacci(), 10))
# [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

def nth_fib(n):
    return next(itertools.islice(fibonacci(), n - 1, n))

ninety_nineth_fib = nth_fib(99) # 354224848179261915075
```

## Section 67.3: Sending objects to a generator

In addition to receiving values from a generator, it is possible to *send* an object to a generator using the `send()` method.

```
def accumulator():
    total = 0
    value = None
    while True:
        # receive sent value
        value = yield total
        if value is None: break
        # aggregate values
        total += value

generator = accumulator()

# advance until the first "yield"
next(generator) # 0

# from this point on, the generator aggregates values
generator.send(1) # 1
generator.send(10) # 11
generator.send(100) # 111
# ...

# Calling next(generator) is equivalent to calling generator.send(None)
next(generator) # StopIteration
```

What happens here is the following:

- When you first call `next(generator)`, the program advances to the first `yield` statement, and returns the value of `total` at that point, which is 0. The execution of the generator suspends at this point.
- When you then call `generator.send(x)`, the interpreter takes the argument `x` and makes it the return value of the last `yield` statement, which gets assigned to `value`. The generator then proceeds as usual, until it yields the next value.
- When you finally call `next(generator)`, the program treats this as if you're sending `None` to the generator. There is nothing special about `None`, however, this example uses `None` as a special value to ask the generator to stop.

## 第67.4节：从另一个可迭代对象中生成所有值

Python 3.x 版本 ≥ 3.3

如果你想从另一个可迭代对象中生成所有值，可以使用 `yield from`：

```
def foob(x):
    yield from range(x * 2)
    yield from range(2)

list(foob(5)) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1]
```

这对生成器同样适用。

```
def fibto(n):
a, b = 1, 1
    while True:
        if a >= n: break
        yield a
a, b = b, a + b

def usefib():
    yield from fibto(10)
    yield from fibto(20)

list(usefib()) # [1, 1, 2, 3, 5, 8, 1, 1, 2, 3, 5, 8, 13]
```

## 第67.5节：迭代

生成器对象支持迭代器协议。也就是说，它提供了一个`next()`方法（Python 3.x中为`__next__()`），用于逐步执行，并且其`__iter__`方法返回其自身。这意味着生成器可以用于任何支持通用可迭代对象的语言结构中。

```
# Python 2.x xrange() 的简单部分实现
def xrange(n):
i = 0
    当 i < n 时:
        yield i
        i += 1

# 循环
for i in xrange(10):
    print(i) # 打印值 0, 1, ..., 9

# 解包
a, b, c = xrange(3) # 0, 1, 2

# 构建列表
l = list(xrange(10)) # [0, 1, ..., 9]
```

## 第67.6节：next()函数

内置函数`next()`是一个方便的包装器，可用于从任何迭代器（包括生成器迭代器）接收值，并在迭代器耗尽时提供默认值。

```
def nums():
    yield 1
```

## Section 67.4: Yielding all values from another iterable

Python 3.x Version ≥ 3.3

Use `yield from` if you want to yield all values from another iterable:

```
def foob(x):
    yield from range(x * 2)
    yield from range(2)

list(foob(5)) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1]
```

This works with generators as well.

```
def fibto(n):
a, b = 1, 1
    while True:
        if a >= n: break
        yield a
a, b = b, a + b

def usefib():
    yield from fibto(10)
    yield from fibto(20)

list(usefib()) # [1, 1, 2, 3, 5, 8, 1, 1, 2, 3, 5, 8, 13]
```

## Section 67.5: Iteration

A generator object supports the *iterator protocol*. That is, it provides a `next()` method (`__next__()` in Python 3.x), which is used to step through its execution, and its `__iter__` method returns itself. This means that a generator can be used in any language construct which supports generic iterable objects.

```
# naive partial implementation of the Python 2.x xrange()
def xrange(n):
i = 0
    while i < n:
        yield i
        i += 1

# looping
for i in xrange(10):
    print(i) # prints the values 0, 1, ..., 9

# unpacking
a, b, c = xrange(3) # 0, 1, 2

# building a list
l = list(xrange(10)) # [0, 1, ..., 9]
```

## Section 67.6: The next() function

The `next()` built-in is a convenient wrapper which can be used to receive a value from any iterator (including a generator iterator) and to provide a default value in case the iterator is exhausted.

```
def nums():
    yield 1
```

```

yield 2
yield 3
generator = nums()

next(generator, None) # 1
next(generator, None) # 2
next(generator, None) # 3
next(generator, None) # None
next(generator, None) # None
# ...

```

语法是 `next(iterator[, default])`。如果迭代器结束且传入了默认值，则返回该默认值。如果未提供默认值，则抛出 `StopIteration` 异常。

## 第67.7节：协程

生成器可以用来实现协程：

```

# 创建生成器并推进到第一个yield
def 协程(func):
    def 启动(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr)
        return cr
    return 启动

# 示例协程
@coroutine
def adder(sum = 0):
    while True:
        x = yield sum
        sum += x

# 示例用法
s = adder()
s.send(1) # 1
s.send(2) # 3

```

协程通常用于实现状态机，因为它们主要适用于创建需要状态才能正常运行的单方法过程。它们在现有状态下操作，并返回操作完成时获得的值。

## 第67.8节：重构列表构建代码

假设你有一段复杂代码，通过从空列表开始并反复追加元素来创建并返回一个列表：

```

def create():
    result = []
    # 逻辑代码...
    result.append(value) # 可能在多个地方
    # 更多逻辑...
    return result # 可能在多个地方

values = create()

```

当不方便用列表推导式替换内部逻辑时，可以将整个函数原地转换为生成器，然后收集结果：

```

yield 2
yield 3
generator = nums()

next(generator, None) # 1
next(generator, None) # 2
next(generator, None) # 3
next(generator, None) # None
next(generator, None) # None
# ...

```

The syntax is `next(iterator[, default])`. If iterator ends and a default value was passed, it is returned. If no default was provided, `StopIteration` is raised.

## Section 67.7: Coroutines

Generators can be used to implement coroutines:

```

# create and advance generator to the first yield
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr)
        return cr
    return start

# example coroutine
@coroutine
def adder(sum = 0):
    while True:
        x = yield sum
        sum += x

# example use
s = adder()
s.send(1) # 1
s.send(2) # 3

```

Coroutines are commonly used to implement state machines, as they are primarily useful for creating single-method procedures that require a state to function properly. They operate on an existing state and return the value obtained on completion of the operation.

## Section 67.8: Refactoring list-building code

Suppose you have complex code that creates and returns a list by starting with a blank list and repeatedly appending to it:

```

def create():
    result = []
    # logic here...
    result.append(value) # possibly in several places
    # more logic...
    return result # possibly in several places

values = create()

```

When it's not practical to replace the inner logic with a list comprehension, you can turn the entire function into a generator in-place, and then collect the results:

```

def create_gen():
    # 逻辑代码...
    yield value
    # 更多逻辑
    return # 如果在函数末尾, 当然不需要

values = list(create_gen())

```

如果逻辑是递归的, 使用 `yield from` 来包含递归调用中所有的值, 形成“扁平化”的结果:

```

def preorder_traversal(node):
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)

```

## 第67.9节 : 使用递归的yield : 递归列出目录中的所有文件

首先, 导入处理文件的库:

```

from os import listdir
from os.path import isfile, join, exists

```

一个用于仅从目录中读取文件的辅助函数:

```

def get_files(path):
    对于路径中的文件进行遍历:
    完整路径= 连接路径和文件名
    如果 是文件(完整路径):
        如果 存在(完整路径):
            生成 完整路径

```

另一个辅助函数, 用于仅获取子目录:

```

定义 获取目录(路径):
    对于 路径中的目录进行遍历:
    完整路径= 连接路径和目录名
    如果不是 文件(完整路径):
        如果 存在(完整路径):
            生成 完整路径

```

现在使用这些函数递归获取目录及其所有子目录中的所有文件 (使用生成器) :

```

def get_files_recursive(directory):
    对于文件在get_files(目录)中:
        生成文件
    对于子目录在get_directories(目录)中:
        对于文件在get_files_recursive(子目录)中:# 这里是递归调用
            生成文件

```

此函数可以使用`yield from`简化:

```

def get_files_recursive(directory):
    yield from get_files(目录)
    对于子目录在get_directories(目录)中:
        yield from get_files_recursive(子目录)

```

```

def create_gen():
    # logic...
    yield value
    # more logic
    return # not needed if at the end of the function, of course

values = list(create_gen())

```

If the logic is recursive, use `yield from` to include all the values from the recursive call in a "flattened" result:

```

def preorder_traversal(node):
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)

```

## Section 67.9: Yield with recursion: recursively listing all files in a directory

First, import the libraries that work with files:

```

from os import listdir
from os.path import isfile, join, exists

```

A helper function to read only files from a directory:

```

def get_files(path):
    for file in listdir(path):
        full_path = join(path, file)
        if isfile(full_path):
            if exists(full_path):
                yield full_path

```

Another helper function to get only the subdirectories:

```

def get_directories(path):
    for directory in listdir(path):
        full_path = join(path, directory)
        if not isfile(full_path):
            if exists(full_path):
                yield full_path

```

Now use these functions to recursively get all files within a directory and all its subdirectories (using generators):

```

def get_files_recursive(directory):
    for file in get_files(directory):
        yield file
    for subdirectory in get_directories(directory):
        for file in get_files_recursive(subdirectory): # here the recursive call
            yield file

```

This function can be simplified using `yield from`:

```

def get_files_recursive(directory):
    yield from get_files(directory)
    for subdirectory in get_directories(directory):
        yield from get_files_recursive(subdirectory)

```

## 第67.10节：生成器表达式

可以使用类似推导式的语法创建生成器迭代器。

```
生成器= (i * 2 对于 i 在 range(3))  
  
next(生成器) # 0  
next(generator) # 2  
next(generator) # 4  
next(generator) # 引发 StopIteration
```

如果一个函数不一定需要传入列表，你可以通过在函数调用中放置生成器表达式来节省字符数（并提高可读性）。函数调用的括号会隐式地将你的表达式变成生成器表达式。

```
sum(i ** 2 for i in range(4)) # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

此外，你还会节省内存，因为生成器允许 Python 按需使用值，而不是加载整个列表（如上例中的[0, 1, 2, 3]），你正在迭代的列表。

## 第67.11节：使用生成器查找斐波那契数列

生成器的一个实际用例是迭代无限序列的值。下面是一个查找斐波那契数列前十项的示例。

```
def fib(a=0, b=1):  
    """生成器，生成斐波那契数。`a` 和 `b` 是种子值"""  
    while True:  
        yield a  
        a, b = b, a + b  
  
f = fib()  
print(''.join(str(next(f)) for _ in range(10)))
```

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

## 第67.12节：搜索

即使不进行迭代，`next` 函数也很有用。将生成器表达式传递给 `next` 是快速搜索第一个匹配某个谓词的元素的简便方法。像下面这样的过程式代码

```
def find_and_transform(序列, 谓词, 函数):  
    for 元素 in 序列:  
        if 谓词(元素):  
            return 函数(元素)  
    raise ValueError  
  
项 = find_and_transform(我的序列, 我的谓词, 我的函数)
```

可以替换为：

```
项 = next(我的函数(x) for x in 我的序列 if 我的谓词(x))  
# 如果没有匹配项，将引发 StopIteration；此异常可以  
# 被捕获并转换（如果需要）。
```

## Section 67.10: Generator expressions

It's possible to create generator iterators using a comprehension-like syntax.

```
generator = (i * 2 for i in range(3))  
  
next(generator) # 0  
next(generator) # 2  
next(generator) # 4  
next(generator) # raises StopIteration
```

If a function doesn't necessarily need to be passed a list, you can save on characters (and improve readability) by placing a generator expression inside a function call. The parenthesis from the function call implicitly make your expression a generator expression.

```
sum(i ** 2 for i in range(4)) # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

Additionally, you will save on memory because instead of loading the entire list you are iterating over ([0, 1, 2, 3] in the above example), the generator allows Python to use values as needed.

## Section 67.11: Using a generator to find Fibonacci Numbers

A practical use case of a generator is to iterate through values of an infinite series. Here's an example of finding the first ten terms of the [Fibonacci Sequence](#).

```
def fib(a=0, b=1):  
    """Generator that yields Fibonacci numbers. `a` and `b` are the seed values"""  
    while True:  
        yield a  
        a, b = b, a + b  
  
f = fib()  
print(''.join(str(next(f)) for _ in range(10)))
```

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

## Section 67.12: Searching

The `next` function is useful even without iterating. Passing a generator expression to `next` is a quick way to search for the first occurrence of an element matching some predicate. Procedural code like

```
def find_and_transform(序列, 谓词, 函数):  
    for element in sequence:  
        if predicate(element):  
            return func(element)  
    raise ValueError  
  
item = find_and_transform(my_sequence, my_predicate, my_func)
```

can be replaced with:

```
item = next(my_func(x) for x in my_sequence if my_predicate(x))  
# StopIteration will be raised if there are no matches; this exception can  
# be caught and transformed, if desired.
```

为此，可能需要创建一个别名，例如first=next，或者一个包装函数来转换异常：

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

## 第67.13节：并行迭代生成器

要并行迭代多个生成器，使用zip内置函数：

```
for x, y in zip(a,b):
    print(x,y)
```

结果为：

```
1 x
2 y
3 z
```

在 Python 2 中，你应该使用itertools.izip。这里我们也可以看到所有的zip函数都会生成元组。

注意，zip 会在任一可迭代对象耗尽元素时停止迭代。如果你想要迭代直到最长的可迭代对象结束，请使用itertools.zip\_longest()。

For this purpose, it may be desirable to create an alias, such as `first = next`, or a wrapper function to convert the exception:

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

## Section 67.13: Iterating over generators in parallel

To iterate over several generators in parallel, use the `zip` builtin:

```
for x, y in zip(a, b):
    print(x, y)
```

Results in:

```
1 x
2 y
3 z
```

In python 2 you should use `itertools.izip` instead. Here we can also see that the all the `zip` functions yield tuples.

Note that zip will stop iterating as soon as one of the iterables runs out of items. If you'd like to iterate for as long as the longest iterable, use `itertools.zip_longest()`.

# 第68章：Reduce

## 参数

	详情
函数	用于归约可迭代对象的函数（必须接受两个参数）。(仅限位置参数)
可迭代对象	将被归约的可迭代对象。(仅限位置参数)
初始化值	归约的起始值。(可选，仅限位置参数)

## 第68.1节：概述

```
# 无需导入
```

```
# 无需导入...
from functools import reduce # ... 但它可以从 functools 模块加载
```

```
from functools import reduce # 必须的
```

reduce 通过对可迭代对象的下一个元素和迄今为止的累计结果重复应用函数来减少可迭代对象。

```
def add(s1, s2):
    return s1 + s2

asequence = [1, 2, 3]

reduce(add, asequence) # 等同于: add(add(1,2),3)
# 输出: 6
```

在此示例中，我们定义了自己的add函数。然而，Python 在operator模块中提供了一个标准的等效函数：

```
import operator
reduce(operator.add, asequence)
# 输出: 6
```

reduce 也可以传入一个起始值：

```
reduce(add, asequence, 10)
# Out: 16
```

## 第68.2节：使用reduce

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                            arg2=s2,
                                            res=s1*s2))

    return s1 * s2

asequence = [1, 2, 3]
```

给定一个initializer，函数通过将其应用于初始化器和第一个可迭代元素开始：

```
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
```

# Chapter 68: Reduce

## Parameter

	Details
function	function that is used for reducing the iterable (must take two arguments). (positional-only)
iterable	iterable that's going to be reduced. (positional-only)

initializer start-value of the reduction. (optional, positional-only)

## Section 68.1: Overview

```
# No import needed
```

```
# No import required...
from functools import reduce # ... but it can be loaded from the functools module
```

```
from functools import reduce # mandatory
```

reduce reduces an iterable by applying a function repeatedly on the next element of an iterable and the cumulative result so far.

```
def add(s1, s2):
    return s1 + s2

asequence = [1, 2, 3]

reduce(add, asequence) # equivalent to: add(add(1,2),3)
# Out: 6
```

In this example, we defined our own add function. However, Python comes with a standard equivalent function in the `operator` module:

```
import operator
reduce(operator.add, asequence)
# Out: 6
```

reduce can also be passed a starting value:

```
reduce(add, asequence, 10)
# Out: 16
```

## Section 68.2: Using reduce

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                            arg2=s2,
                                            res=s1*s2))

    return s1 * s2

asequence = [1, 2, 3]
```

Given an initializer the function is started by applying it to the initializer and the first iterable element:

```
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
```

```
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# 输出: 30
```

如果没有initializer参数，reduce会先将函数应用于列表的前两个元素：

```
cumprod = reduce(multiply, asequence)
# 输出: 1 * 2 = 2
#      2 * 3 = 6
print(cumprod)
# 输出: 6
```

## 第68.3节：累积乘积

```
import operator
reduce(operator.mul, [10, 5, -3])
# 输出: -150
```

## 第68.4节：any/all的非短路变体

reduce不会在 iterable 完全迭代完之前终止迭代，因此它可以用来创建非短路的any()或all()函数：

```
import operator
# 非短路的"all"
reduce(operator.and_, [False, True, True, True]) # = False

# 非短路"any"
reduce(operator.or_, [True, False, False, False]) # = True
```

```
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# Out: 30
```

Without initializer parameter the reduce starts by applying the function to the first two list elements:

```
cumprod = reduce(multiply, asequence)
# Out: 1 * 2 = 2
#      2 * 3 = 6
print(cumprod)
# Out: 6
```

## Section 68.3: Cumulative product

```
import operator
reduce(operator.mul, [10, 5, -3])
# Out: -150
```

## Section 68.4: Non short-circuit variant of any/all

reduce will not terminate the iteration before the iterable has been completely iterated over so it can be used to create a non short-circuit any() or all() function:

```
import operator
# non short-circuit "all"
reduce(operator.and_, [False, True, True, True]) # = False

# non short-circuit "any"
reduce(operator.or_, [True, False, False, False]) # = True
```

# 第69章：映射函数

## 参数

	详情
函数	映射函数（必须接受与可迭代对象数量相同的参数）（仅限位置参数）
可迭代对象	该函数应用于可迭代对象的每个元素（仅限位置参数）
*additional_iterables	参见可迭代对象，但数量不限（可选，且仅限位置参数）

## 第69.1节：map、itertools imap 和 future\_builtins.map 的基本用法

map 函数是 Python 内置函数中用于函数式编程的最简单函数。 map() 将指定函数应用于可迭代对象的每个元素：

```
names = ['Fred', 'Wilma', 'Barney']
Python 3.x 版本 ≥ 3.0
map(len, names) # Python 3.x 中的 map 是一个类；其实例是可迭代的
# 输出: <map object at 0x00000198B32E2CF8>
```

Python 3 兼容的 map 包含在 future\_builtins 模块中：

```
Python 2.x 版本 ≥ 2.6
from future_builtins import map # 包含兼容 Python 3.x 的 map()
map(len, names) # 见下文
# 输出: <itertools.imap 实例 位于 0x3eb0a20>
```

或者，在 Python 2 中可以使用来自 itertools 的 map 来获取生成器

```
Python 2.x 版本 ≥ 2.3
map(len, names) # map() 返回一个列表
# 输出: [4, 5, 6]

from itertools import imap
imap(len, names) # itertools imap() 返回一个生成器
# 输出: <itertools.imap at 0x405ea20>
```

结果可以显式转换为列表，以消除 Python 2 和 3 之间的差异：

```
list(map(len, names))
# 输出: [4, 5, 6]
```

map() 可以被等效的 列表推导式 或 生成器表达式 替代：

```
[len(项) for 项 in 名称] # 等同于 Python 2.x 的 map()
# 输出: [4, 5, 6]

(len(item) for item in names) # 等同于 Python 3.x 的 map()
# 输出: <generator object <genexpr> at 0x00000195888D5FC0>
```

## 第69.2节：映射可迭代对象中的每个值

例如，你可以取每个元素的绝对值：

```
list(map(abs, (1, -1, 2, -2, 3, -3))) # 在2.x中调用`list`不是必须的
```

# Chapter 69: Map Function

## Parameter

	Details
function	function for mapping (must take as many parameters as there are iterables) ( <i>positional-only</i> )
iterable	the function is applied to each element of the iterable ( <i>positional-only</i> )

\*additional\_iterables see iterable, but as many as you like (*optional, positional-only*)

## Section 69.1: Basic use of map, itertools imap and future\_builtins.map

The map function is the simplest one among Python built-ins used for functional programming. map() applies a specified function to each element in an iterable:

```
names = ['Fred', 'Wilma', 'Barney']
Python 3.x Version ≥ 3.0
map(len, names) # map in Python 3.x is a class; its instances are iterable
# Out: <map object at 0x00000198B32E2CF8>
```

A Python 3-compatible map is included in the future\_builtins module:

```
Python 2.x Version ≥ 2.6
from future_builtins import map # contains a Python 3.x compatible map()
map(len, names) # see below
# Out: <itertools.imap instance at 0x3eb0a20>
```

Alternatively, in Python 2 one can use imap from itertools to get a generator

```
Python 2.x Version ≥ 2.3
map(len, names) # map() returns a list
# Out: [4, 5, 6]

from itertools import imap
imap(len, names) # itertools imap() returns a generator
# Out: <itertools.imap at 0x405ea20>
```

The result can be explicitly converted to a list to remove the differences between Python 2 and 3:

```
list(map(len, names))
# Out: [4, 5, 6]
```

map() can be replaced by an equivalent list comprehension or generator expression:

```
[len(item) for item in names] # equivalent to Python 2.x map()
# Out: [4, 5, 6]

(len(item) for item in names) # equivalent to Python 3.x map()
# Out: <generator object <genexpr> at 0x00000195888D5FC0>
```

## Section 69.2: Mapping each value in an iterable

For example, you can take the absolute value of each element:

```
list(map(abs, (1, -1, 2, -2, 3, -3))) # the call to `list` is unnecessary in 2.x
```

```
# 输出: [1, 1, 2, 2, 3, 3]
```

匿名函数也支持映射列表：

```
map(lambda x:x*2, [1, 2, 3, 4, 5])  
# 输出: [2, 4, 6, 8, 10]
```

或者将小数值转换为百分比：

```
def to_percent(num):  
    return num * 100  
  
list(map(to_percent, [0.95, 0.75, 1.01, 0.1]))  
# 输出: [95.0, 75.0, 101.0, 10.0]
```

或者将美元转换为欧元（给定汇率）：

```
from functools import partial  
from operator import mul  
  
rate = 0.9 # 虚构汇率, 1美元 = 0.9欧元  
dollars = {'under_my_bed': 1000,  
          'jeans': 45,  
          'bank': 5000}  
  
sum(map(partial(mul, rate), dollars.values()))  
# 输出: 5440.5
```

`functools.partial` 是一种方便的方式，用于固定函数的参数，使其可以与 `map` 一起使用，而无需使用 `lambda` 或创建自定义函数。

## 第69.3节：映射不同可迭代对象的值

例如计算多个可迭代对象中每个第  $i$  个元素的平均值：

```
def average(*args):  
    return float(sum(args)) / len(args) # 强制转换为浮点数 - 仅对 Python 2.x 必需  
  
measurement1 = [100, 111, 99, 97]  
measurement2 = [102, 117, 91, 102]  
measurement3 = [104, 102, 95, 101]  
  
list(map(average, measurement1, measurement2, measurement3))  
# 输出: [102.0, 110.0, 95.0, 100.0]
```

如果传递给 `map` 多个可迭代对象，根据 Python 版本的不同，有不同的要求：

- 函数必须接受与可迭代对象数量相同的参数：

```
def median_of_three(a, b, c):  
    return sorted((a, b, c))[1]  
  
list(map(median_of_three, measurement1, measurement2))
```

TypeError: median\_of\_three() 缺少 1 个必需的位置参数: 'c'

```
# Out: [1, 1, 2, 2, 3, 3]
```

Anonymous function also support for mapping a list:

```
map(lambda x:x*2, [1, 2, 3, 4, 5])  
# Out: [2, 4, 6, 8, 10]
```

or converting decimal values to percentages:

```
def to_percent(num):  
    return num * 100  
  
list(map(to_percent, [0.95, 0.75, 1.01, 0.1]))  
# Out: [95.0, 75.0, 101.0, 10.0]
```

or converting dollars to euros (given an exchange rate):

```
from functools import partial  
from operator import mul  
  
rate = 0.9 # fictitious exchange rate, 1 dollar = 0.9 euros  
dollars = {'under_my_bed': 1000,  
          'jeans': 45,  
          'bank': 5000}  
  
sum(map(partial(mul, rate), dollars.values()))  
# Out: 5440.5
```

`functools.partial` 是一种方便的方式，用于固定函数的参数，使其可以与 `map` 一起使用，而无需使用 `lambda` 或创建自定义函数。

## Section 69.3: Mapping values of different iterables

For example calculating the average of each  $i$ -th element of multiple iterables:

```
def average(*args):  
    return float(sum(args)) / len(args) # cast to float - only mandatory for python 2.x  
  
measurement1 = [100, 111, 99, 97]  
measurement2 = [102, 117, 91, 102]  
measurement3 = [104, 102, 95, 101]  
  
list(map(average, measurement1, measurement2, measurement3))  
# Out: [102.0, 110.0, 95.0, 100.0]
```

There are different requirements if more than one iterable is passed to `map` depending on the version of python:

- The function must take as many parameters as there are iterables:

```
def median_of_three(a, b, c):  
    return sorted((a, b, c))[1]  
  
list(map(median_of_three, measurement1, measurement2))
```

TypeError: median\_of\_three() missing 1 required positional argument: 'c'

```
list(map(三数中值, 测量1, 测量2, 测量3, 测量4))
```

TypeError: median\_of\_three() 接受3个位置参数, 但给了4个

Python 2.x 版本 ≥ 2.0.1

- map: 映射会持续迭代, 直到所有可迭代对象都被完全消费, 但会假设来自已完全消费的可迭代对象的值为None  
:

```
import operator

测量1 = [100, 111, 99, 97]
测量2 = [102, 117]

# 计算元素之间的差异
list(map(operator.sub, 测量1, 测量2))
```

TypeError: 不支持的操作数类型 : 'int' 和 'NoneType' 进行减法

- itertools imap 和 future\_builtins.map: 映射会在任一可迭代对象停止时停止 :

```
import operator
from itertools import imap

测量1 = [100, 111, 99, 97]
测量2 = [102, 117]

# 计算元素之间的差异
list(imap(operator.sub, measurement1, measurement2))
# 输出: [-2, -6]
list(imap(operator.sub, measurement2, measurement1))
# 输出: [2, 6]
```

Python 3.x 版本 ≥ 3.0.0

- 映射会在任一可迭代对象停止时终止 :

```
import operator

测量1 = [100, 111, 99, 97]
测量2 = [102, 117]

# 计算元素之间的差异
list(map(operator.sub, 测量1, 测量2))
# 输出: [-2, -6]
list(map(operator.sub, 测量2, 测量1))
# 输出: [2, 6]
```

```
list(map(median_of_three, measurement1, measurement2, measurement3, measurement4))
```

TypeError: median\_of\_three() takes 3 positional arguments but 4 were given

Python 2.x Version ≥ 2.0.1

- map: The mapping iterates as long as one iterable is still not fully consumed but assumes None from the fully consumed iterables:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
```

TypeError: unsupported operand type(s) for -: 'int' and 'NoneType'

- itertools imap and future\_builtins.map: The mapping stops as soon as one iterable stops:

```
import operator
from itertools import imap

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(imap(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(imap(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

Python 3.x Version ≥ 3.0.0

- The mapping stops as soon as one iterable stops:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(map(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

## 第69.4节：使用Map转置：将"None"作为函数参数（仅限python 2.x）

```
from itertools import imap
from future_builtins import map as fmap # 不同名称以突出差异

image = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]

list(map(None, *image))
# 输出: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(fmap(None, *image))
# 输出: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(imap(None, *image))
# 输出: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]

image2 = [[1, 2, 3],
          [4, 5],
          [7, 8, 9]]
list(map(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8), (3, None, 9)] # 用 None 填充缺失值
list(fmap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)] # 忽略含缺失值的列
list(imap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)] # 同上

Python 3.x 版本 ≥ 3.0.0
list(map(None, *image))
```

TypeError: 'NoneType' 对象不可调用

但是有一种变通方法可以获得类似的结果：

```
def conv_to_list(*args):
    return list(args)

list(map(conv_to_list, *image))
# 输出: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

## 第69.5节：串行映射与并行映射

map() 是一个内置函数，这意味着它在任何地方都可用，无需使用'import'语句。它像print()一样随处可用。如果你看示例5，你会看到我必须先使用import语句才能使用漂亮打印 (import pprint)。因此pprint不是内置函数。

### 串行映射

在这种情况下，可迭代对象的每个参数按升序依次作为映射函数的参数传入。当我们只有一个可迭代对象需要映射且映射函数只需要一个参数时，就会出现这种情况。

#### 示例1

```
insects = ['fly', 'ant', 'beetle', 'cankerworm']
f = lambda x: x + ' is an insect'
print(list(map(f, insects))) # 定义的函数 f 在可迭代对象的每个元素上执行
```

## Section 69.4: Transposing with Map: Using "None" as function argument (python 2.x only)

```
from itertools import imap
from future_builtins import map as fmap # Different name to highlight differences

image = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]

list(map(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(fmap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(imap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]

image2 = [[1, 2, 3],
          [4, 5],
          [7, 8, 9]]
list(map(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8), (3, None, 9)] # Fill missing values with None
list(fmap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)] # ignore columns with missing values
list(imap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)] # ditto

Python 3.x Version ≥ 3.0.0
list(map(None, *image))
```

TypeError: 'NoneType' object is not callable

But there is a workaround to have similar results:

```
def conv_to_list(*args):
    return list(args)

list(map(conv_to_list, *image))
# Out: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

## Section 69.5: Series and Parallel Mapping

map() is a built-in function, which means that it is available everywhere without the need to use an 'import' statement. It is available everywhere just like print(). If you look at Example 5 you will see that I had to use an import statement before I could use pretty print (import pprint). Thus pprint is not a built-in function

### Series mapping

In this case each argument of the iterable is supplied as argument to the mapping function in ascending order. This arises when we have just one iterable to map and the mapping function requires a single argument.

#### Example 1

```
insects = ['fly', 'ant', 'beetle', 'cankerworm']
f = lambda x: x + ' is an insect'
print(list(map(f, insects))) # the function defined by f is executed on each item of the iterable
```

insects

结果为

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

示例 2

```
print(list(map(len, insects))) # len 函数对 insects 列表中的每个元素执行
```

结果为

```
[3, 3, 6, 10]
```

并行映射

在这种情况下，映射函数的每个参数都是从所有可迭代对象中并行提取的（每个可迭代对象取一个）。因此，提供的可迭代对象数量必须与函数所需的参数数量相匹配。

```
食肉动物 = ['狮子', '老虎', '豹', '北极狐']
```

```
食草动物 = ['非洲水牛', '驼鹿', '长颈鹿', '鹦鹉']
```

```
杂食动物 = ['鸡', '鸽子', '老鼠', '猪']
```

```
def animals(w, x, y, z):
    return '{0}, {1}, {2}, 和 {3} 都是动物'.format(w.title(), x, y, z)
```

示例 3

```
# 参数过多
# 注意这里 map 试图将四个可迭代对象中的每个各取一个元素传给 len。这
导致 len 报错
# 它接收了过多的参数
print(list(map(len, insects, carnivores, herbivores, omnivores)))
```

结果为

```
TypeError: len() 需要恰好一个参数 (给定了4个)
```

示例 4

```
# 参数太少
# 注意这里 map 应该对 insects 的每个元素逐个执行 animals。
但是 animals 抱怨说
# 它只得到一个参数，而它本来期望四个。
print(list(map(animals, insects)))
```

结果为

```
TypeError: animals() 缺少 3 个必需的位置参数 : 'x', 'y', 和 'z'
```

示例 5

```
# 这里 map 为 w, x, y, z 分别提供来自列表的一个值
import pprint
pprint pprint(list(map(animals, insects, carnivores, herbivores, omnivores)))
```

insects

results in

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

Example 2

```
print(list(map(len, insects))) # the len function is executed each item in the insect list
```

results in

```
[3, 3, 6, 10]
```

Parallel mapping

In this case each argument of the mapping function is pulled from across all iterables (one from each iterable) in parallel. Thus the number of iterables supplied must match the number of arguments required by the function.

```
carnivores = ['lion', 'tiger', 'leopard', 'arctic fox']
```

```
herbivores = ['african buffalo', 'moose', 'okapi', 'parakeet']
```

```
omnivores = ['chicken', 'dove', 'mouse', 'pig']
```

```
def animals(w, x, y, z):
    return '{0}, {1}, {2}, and {3} ARE ALL ANIMALS'.format(w.title(), x, y, z)
```

Example 3

```
# Too many arguments
# observe here that map is trying to pass one item each from each of the four iterables to len. This
leads len to complain that
# it is being fed too many arguments
print(list(map(len, insects, carnivores, herbivores, omnivores)))
```

results in

```
TypeError: len() takes exactly one argument (4 given)
```

Example 4

```
# Too few arguments
# observe here that map is supposed to execute animal on individual elements of insects one-by-one.
But animals complain when
# it only gets one argument, whereas it was expecting four.
print(list(map(animals, insects)))
```

results in

```
TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Example 5

```
# here map supplies w, x, y, z with one value from across the list
import pprint
pprint pprint(list(map(animals, insects, carnivores, herbivores, omnivores)))
```

结果为

```
[“飞，狮子，非洲水牛和鸡都是动物”，  
“蚂蚁，老虎，驼鹿和鸽子都是动物”，  
“甲虫，豹，长颈鹿和老鼠都是动物”，  
“尺蠖，北极狐，鹦鹉和猪都是动物”]
```

results in

```
[‘Fly, lion, african buffalo, and chicken ARE ALL ANIMALS’,  
‘Ant, tiger, moose, and dove ARE ALL ANIMALS’,  
‘Beetle, leopard, okapi, and mouse ARE ALL ANIMALS’,  
‘Cankerworm, arctic fox, parakeet, and pig ARE ALL ANIMALS’]
```

# 视频：Python 数据 科学与机器 学习训练营

学习如何使用 NumPy、Pandas、Seaborn、  
Matplotlib、Plotly、Scikit-Learn、机器学习、  
Tensorflow 等！



- ✓ 使用 Python 进行数据科学和机器学习
- ✓ 使用 Spark 进行大数据分析
- ✓ 实现机器学习算法
- ✓ 学习使用 NumPy 处理数值数据
- ✓ 学习使用 Pandas 进行数据分析
- ✓ 学习使用 Matplotlib 进行 Python 绘图
- ✓ 学习使用 Seaborn 进行统计图表绘制
- ✓ 使用 Plotly 进行交互式动态可视化
- ✓ 使用 SciKit-Learn 进行机器学习任务
- ✓ K-均值聚类
- ✓ 逻辑回归
- ✓ 线性回归
- ✓ 随机森林和决策树
- ✓ 神经网络
- ✓ 支持向量机今天观看→

# VIDEO: Python for Data Science and Machine Learning Bootcamp

Learn how to use NumPy, Pandas, Seaborn,  
Matplotlib , Plotly, Scikit-Learn , Machine Learning,  
Tensorflow, and more!



- ✓ Use Python for Data Science and Machine Learning
- ✓ Use Spark for Big Data Analysis
- ✓ Implement Machine Learning Algorithms
- ✓ Learn to use NumPy for Numerical Data
- ✓ Learn to use Pandas for Data Analysis
- ✓ Learn to use Matplotlib for Python Plotting
- ✓ Learn to use Seaborn for statistical plots
- ✓ Use Plotly for interactive dynamic visualizations
- ✓ Use SciKit-Learn for Machine Learning Tasks
- ✓ K-Means Clustering
- ✓ Logistic Regression
- ✓ Linear Regression
- ✓ Random Forest and Decision Trees
- ✓ Neural Networks
- ✓ Support Vector Machines

**Watch Today →**

# 第70章：指数运算

## 第70.1节：使用内置函数进行指数运算：\*\* 和 pow()

指数运算可以使用内置的pow函数或\*\*运算符：

```
2** 3    # 8  
pow(2, 3) # 8
```

对于大多数（Python 2.x 中全部）算术运算，结果的类型将是较宽操作数的类型。但对于\*\*运算符，这条规则不适用；以下情况是该规则的例外：

- 底数：int，指数：int < 0：

```
2** -3  
# 输出: 0.125 (结果是浮点数)
```

- 这对于 Python 3.x 也同样适用。
- 在 Python 2.2.0 之前，这会引发一个ValueError。
- 底数：int<0或float<0，指数：float!=int

```
(-2) ** (0.5) # 也包括 (-2.) ** (0.5)  
# 输出: (8.659560562354934e-17+1.4142135623730951j) (结果是复数)
```

- 在 Python 3.0.0 之前，这会引发一个ValueError。

operator模块包含两个函数，它们等价于\*\*运算符：

```
import operator  
operator.pow(4, 2)      # 16  
operator.__pow__(4, 3)  # 64
```

或者可以直接调用\_\_pow\_\_方法：

```
val1, val2 = 4, 2  
val1.__pow__(val2)      # 16  
val2.__rpow__(val1)      # 16  
# 不可变类如 int, float, complex 不支持原地幂运算：  
# val1.__ipow__(val2)
```

## 第70.2节：平方根：math.sqrt() 和 cmath.sqrt

math模块包含math.sqrt()函数，可以计算任何数字（可转换为float）的平方根，结果总是float类型：

```
import math  
  
math.sqrt(9)            # 3.0  
math.sqrt(11.11)        # 3.3331666624997918  
math.sqrt(Decimal('6.25')) # 2.5
```

如果结果为complex，math.sqrt()函数会抛出ValueError异常：

# Chapter 70: Exponentiation

## Section 70.1: Exponentiation using builtins: \*\* and pow()

Exponentiation can be used by using the builtin `pow`-function or the `**` operator:

```
2 ** 3    # 8  
pow(2, 3) # 8
```

For most (all in Python 2.x) arithmetic operations the result's type will be that of the wider operand. This is not true for `**`; the following cases are exceptions from this rule:

- Base: `int`, exponent: `int < 0`:

```
2 ** -3  
# Out: 0.125 (result is a float)
```

- This is also valid for Python 3.x.
- Before Python 2.2.0, this raised a `ValueError`.
- Base: `int < 0` or `float < 0`, exponent: `float != int`

```
(-2) ** (0.5) # also (-2.) ** (0.5)  
# Out: (8.659560562354934e-17+1.4142135623730951j) (result is complex)
```

- Before python 3.0.0, this raised a `ValueError`.

The `operator` module contains two functions that are equivalent to the `**`-operator:

```
import operator  
operator.pow(4, 2)      # 16  
operator.__pow__(4, 3)  # 64
```

or one could directly call the `__pow__` method:

```
val1, val2 = 4, 2  
val1.__pow__(val2)      # 16  
val2.__rpow__(val1)      # 16  
# in-place power operation isn't supported by immutable classes like int, float, complex:  
# val1.__ipow__(val2)
```

## Section 70.2: Square root: math.sqrt() and cmath.sqrt

The `math` module contains the `math.sqrt()`-function that can compute the square root of any number (that can be converted to a `float`) and the result will always be a `float`:

```
import math  
  
math.sqrt(9)            # 3.0  
math.sqrt(11.11)        # 3.3331666624997918  
math.sqrt(Decimal('6.25')) # 2.5
```

The `math.sqrt()` function raises a `ValueError` if the result would be `complex`:

```
math.sqrt(-10)
```

ValueError: math domain error

math.sqrt(x)比math.pow(x, 0.5)或x \*\* 0.5更快，但结果的精度相同。cmath模块与math模块非常相似，区别在于它可以计算复数，且所有结果均为 a + bi 形式。它也可以使用.sqrt()函数：

```
import cmath
```

```
cmath.sqrt(4) # 2+0j  
cmath.sqrt(-4) # 2j
```

j 是什么？j 是 -1 的平方根的等价物。所有数字都可以表示为 a + bi 的形式，或者在这种情况下，a + bj。a 是数字的实部，比如 2+0j 中的 2。由于它没有虚部，b 是 0。b 表示数字的虚部，比如 2j 中的 2。由于这里没有实部，2j 也可以写成 0 + 2j。

## 第 70.3 节：模幂运算：带三个参数的 pow()

为 pow() 提供三个参数 pow(a, b, c) 会计算模幂运算 a mod c：

```
pow(3, 4, 17) # 13
```

# 等价的未优化表达式：  
3\*\* 4 % 17 # 13

# 步骤：  
3\*\* 4 # 81  
81 % 17 # 13

对于内置类型，只有在满足以下条件时才能使用模幂运算：

- 第一个参数是一个整数
- 第二个参数是一个整数 $\geq 0$
- 第三个参数是一个整数 $\neq 0$

这些限制在Python 3.x中也存在

例如，可以使用3参数形式的pow来定义一个模逆函数：

```
def modular_inverse(x, p):  
    """找到a，使得 a*x ≡ 1 (mod p)，假设p是素数。"""  
    return pow(x, p-2, p)
```

```
[modular_inverse(x, 13) for x in range(1,13)]  
# 输出: [1, 7, 9, 10, 8, 11, 2, 5, 3, 4, 6, 12]
```

## 第70.4节：计算大整数根

尽管Python原生支持大整数，但在Python中对非常大的数字取n次根可能会失败。

```
x = 2 ** 100  
cube = x ** 3
```

```
math.sqrt(-10)
```

ValueError: math domain error

math.sqrt(x) is faster than math.pow(x, 0.5) or x \*\* 0.5 but the precision of the results is the same. The cmath module is extremely similar to the math module, except for the fact it can compute complex numbers and all of its results are in the form of a + bi. It can also use .sqrt():

```
import cmath
```

```
cmath.sqrt(4) # 2+0j  
cmath.sqrt(-4) # 2j
```

What's with the j? j is the equivalent to the square root of -1. All numbers can be put into the form a + bi, or in this case, a + bj. a is the real part of the number like the 2 in 2+0j. Since it has no imaginary part, b is 0. b represents part of the imaginary part of the number like the 2 in 2j. Since there is no real part in this, 2j can also be written as 0 + 2j.

## Section 70.3: Modular exponentiation: pow() with 3 arguments

Supplying pow() with 3 arguments pow(a, b, c) evaluates the [modular exponentiation](#)  $ab \bmod c$ :

```
pow(3, 4, 17) # 13
```

# equivalent unoptimized expression:  
3 \*\* 4 % 17 # 13

# steps:  
3 \*\* 4 # 81  
81 % 17 # 13

For built-in types using modular exponentiation is only possible if:

- First argument is an int
- Second argument is an int  $\geq 0$
- Third argument is an int  $\neq 0$

These restrictions are also present in python 3.x

For example one can use the 3-argument form of pow to define a [modular inverse](#) function:

```
def modular_inverse(x, p):  
    """Find a such as a*x ≡ 1 (mod p), assuming p is prime."""  
    return pow(x, p-2, p)
```

```
[modular_inverse(x, 13) for x in range(1,13)]  
# Out: [1, 7, 9, 10, 8, 11, 2, 5, 3, 4, 6, 12]
```

## Section 70.4: Computing large integer roots

Even though Python natively supports big integers, taking the nth root of very large numbers can fail in Python.

```
x = 2 ** 100  
cube = x ** 3
```

```
root = cube ** (1.0 / 3)
```

溢出错误：长整型数值过大，无法转换为浮点数

处理如此大的整数时，您需要使用自定义函数来计算数字的第n次方根。

```
def nth_root(x, n):
    # 从第n次方根的合理范围开始。
    upper_bound = 1
    while upper_bound ** n <= x:
        upper_bound *= 2
    lower_bound = upper_bound // 2
    # 只要范围合理，就继续寻找更好的结果。
    while lower_bound < upper_bound:
        mid = (lower_bound + upper_bound) // 2
        mid_nth = mid ** n
        if lower_bound < mid and mid_nth < x:
            lower_bound = mid
        elif upper_bound > mid and mid_nth > x:
            upper_bound = mid
        else:
            # 找到完美的第n次方根。
            return mid
    return mid + 1

x = 2 ** 100
cube = x ** 3
root = nth_root(cube, 3)
x == root
# 真
```

```
root = cube ** (1.0 / 3)
```

OverflowError: long int too large to convert to float

When dealing with such large integers, you will need to use a custom function to compute the nth root of a number.

```
def nth_root(x, n):
    # Start with some reasonable bounds around the nth root.
    upper_bound = 1
    while upper_bound ** n <= x:
        upper_bound *= 2
    lower_bound = upper_bound // 2
    # Keep searching for a better result as long as the bounds make sense.
    while lower_bound < upper_bound:
        mid = (lower_bound + upper_bound) // 2
        mid_nth = mid ** n
        if lower_bound < mid and mid_nth < x:
            lower_bound = mid
        elif upper_bound > mid and mid_nth > x:
            upper_bound = mid
        else:
            # Found perfect nth root.
            return mid
    return mid + 1

x = 2 ** 100
cube = x ** 3
root = nth_root(cube, 3)
x == root
# True
```

## 第70.5节：使用math模块进行指数运算： math.pow()

math模块包含另一个math.pow()函数。与内置的pow()函数或\*\*运算符的区别在于结果总是一个float类型：

```
import math
math.pow(2, 2)      # 4.0
math.pow(-2, 2)     # 4.0
```

这排除了带有复数输入的计算：

```
math.pow(2, 2+0j)
```

TypeError: 无法将复数转换为浮点数

以及会导致复数结果的计算：

```
math.pow(-2, 0.5)
```

ValueError: math domain error

## Section 70.5: Exponentiation using the math module: math.pow()

The `math`-module contains another `math.pow()` function. The difference to the builtin `pow()`-function or `**` operator is that the result is always a `float`:

```
import math
math.pow(2, 2)      # 4.0
math.pow(-2, 2)     # 4.0
```

Which excludes computations with complex inputs:

```
math.pow(2, 2+0j)
```

TypeError: can't convert complex to float

and computations that would lead to complex results:

```
math.pow(-2, 0.5)
```

ValueError: math domain error

## 第70.6节：指数函数：math.exp() 和 cmath.exp()

math 和 cmath模块都包含欧拉数：e，使用内置的pow()函数或\*\*运算符时，效果大致等同于math.exp()：

```
import math

math.e ** 2 # 7.3890560989306495
math.exp(2) # 7.38905609893065

import cmath
cmath.e ** 2 # 7.3890560989306495
cmath.exp(2) # (7.38905609893065+0j)
```

不过结果有所不同，直接使用指数函数比用内置的以math.e为底的幂运算更可靠：

```
print(math.e ** 10)      # 22026.465794806703
print(math.exp(10))      # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
# 差异从这里开始 -----^
```

## 第70.7节：指数函数减1：math.expm1()

math模块包含expm1()函数，可以计算表达式math.e \*\* x - 1，对于非常小的x，其精度比math.exp(x)或cmath.exp(x)更高：

```
import math

print(math.e ** 1e-3 - 1) # 0.0010005001667083846
print(math.exp(1e-3) - 1) # 0.0010005001667083846
print(math.expm1(1e-3)) # 0.0010005001667083417
# -----^
```

对于非常小的x，差异变得更大：

```
print(math.e ** 1e-15 - 1) # 1.1102230246251565e-15
print(math.exp(1e-15) - 1) # 1.1102230246251565e-15
print(math.expm1(1e-15)) # 1.0000000000000007e-15
# -----^
```

这一改进在科学计算中非常重要。例如，普朗克定律（Planck's law）包含一个指数函数减1：

```
def planks_law(lambda_, T):
    from scipy.constants import h, k, c # 如果未安装scipy，请硬编码这些常数！
    return 2 * h * c ** 2 / (lambda_ ** 5 * math.expm1(h * c / (lambda_ * k * T)))

def planks_law_naive(lambda_, T):
    from scipy.constants import h, k, c # 如果未安装scipy，请硬编码这些常数！
    return 2 * h * c ** 2 / (lambda_ ** 5 * (math.e ** (h * c / (lambda_ * k * T)) - 1))

planks_law(100, 5000)      # 4.139080074896474e-19
planks_law_naive(100, 5000) # 4.139080073488451e-19
# -----^
```

## Section 70.6: Exponential function: math.exp() and cmath.exp()

Both the `math` and `cmath`-module contain the [Euler number: e](#) and using it with the builtin `pow()`-function or `**`-operator works mostly like `math.exp()`:

```
import math

math.e ** 2 # 7.3890560989306495
math.exp(2) # 7.38905609893065

import cmath
cmath.e ** 2 # 7.3890560989306495
cmath.exp(2) # (7.38905609893065+0j)
```

However the result is different and using the exponential function directly is more reliable than builtin exponentiation with base `math.e`:

```
print(math.e ** 10)      # 22026.465794806703
print(math.exp(10))      # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
# difference starts here -----^
```

## Section 70.7: Exponential function minus 1: math.expm1()

The `math` module contains the `expm1()`-function that can compute the expression `math.e ** x - 1` for very small x with higher precision than `math.exp(x)` or `cmath.exp(x)` would allow:

```
import math

print(math.e ** 1e-3 - 1) # 0.0010005001667083846
print(math.exp(1e-3) - 1) # 0.0010005001667083846
print(math.expm1(1e-3)) # 0.0010005001667083417
# -----^
```

For very small x the difference gets bigger:

```
print(math.e ** 1e-15 - 1) # 1.1102230246251565e-15
print(math.exp(1e-15) - 1) # 1.1102230246251565e-15
print(math.expm1(1e-15)) # 1.0000000000000007e-15
# -----^
```

The improvement is significant in scientific computing. For example the [Planck's law](#) contains an exponential function minus 1:

```
def planks_law(lambda_, T):
    from scipy.constants import h, k, c # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * math.expm1(h * c / (lambda_ * k * T)))

def planks_law_naive(lambda_, T):
    from scipy.constants import h, k, c # If no scipy installed hardcode these!
    return 2 * h * c ** 2 / (lambda_ ** 5 * (math.e ** (h * c / (lambda_ * k * T)) - 1))

planks_law(100, 5000)      # 4.139080074896474e-19
planks_law_naive(100, 5000) # 4.139080073488451e-19
# -----^
```

```
planks_law(1000, 5000)      # 4.139080128493406e-23
planks_law_naive(1000, 5000) # 4.139080233183142e-23
#-----
```

## 第70.8节：魔法方法和指数运算：内置、math和cmath

假设你有一个只存储整数值的类：

```
class Integer(object):
    def __init__(self, value):
        self.value = int(value) # 转换为整数

    def __repr__(self):
        return '{cls}({val})'.format(cls=self.__class__.__name__,
                                     val=self.value)

    def __pow__(self, other, modulo=None):
        if modulo is None:
            print('使用 __pow__')
            return self.__class__(self.value ** other)
        else:
            print('使用 __pow__ 幂运算')
            return self.__class__(pow(self.value, other, modulo))

    def __float__(self):
        print('使用 __float__ 转换')
        return float(self.value)

    def __complex__(self):
        print('使用 __complex__ 转换')
        return complex(self.value, 0)
```

使用内置的 pow 函数或 \*\* 运算符总是调用 \_\_pow\_\_ 方法：

```
Integer(2) ** 2             # Integer(4)
# 输出：使用 __pow__
Integer(2) ** 2.5           # Integer(5)
# 输出：使用 __pow__
pow(Integer(2), 0.5)        # Integer(1)
# 输出：使用 __pow__
operator.pow(Integer(2), 3)  # Integer(8)
# 输出：使用 __pow__
operator.__pow__(Integer(3), 3) # Integer(27)
# 输出：使用 __pow__
```

\_\_pow\_\_() 方法的第二个参数只能通过内置的 pow() 方法提供，或者直接调用该方法：

```
pow(Integer(2), 3, 4)       # Integer(0)
# 输出：使用带模数的 __pow__
Integer(2).__pow__(3, 4)     # Integer(0)
# 输出：使用带模数的 __pow__
```

而 math 函数总是将其转换为 float 并使用浮点计算：

```
import math
```

```
planks_law(1000, 5000)      # 4.139080128493406e-23
planks_law_naive(1000, 5000) # 4.139080233183142e-23
#-----
```

## Section 70.8: Magic methods and exponentiation: builtin, math and cmath

Supposing you have a class that stores purely integer values:

```
class Integer(object):
    def __init__(self, value):
        self.value = int(value) # Cast to an integer

    def __repr__(self):
        return '{cls}({val})'.format(cls=self.__class__.__name__,
                                     val=self.value)

    def __pow__(self, other, modulo=None):
        if modulo is None:
            print('Using __pow__')
            return self.__class__(self.value ** other)
        else:
            print('Using __pow__ with modulo')
            return self.__class__(pow(self.value, other, modulo))

    def __float__(self):
        print('Using __float__')
        return float(self.value)

    def __complex__(self):
        print('Using __complex__')
        return complex(self.value, 0)
```

Using the builtin pow function or \*\* operator always calls \_\_pow\_\_:

```
Integer(2) ** 2             # Integer(4)
# Prints: Using __pow__
Integer(2) ** 2.5           # Integer(5)
# Prints: Using __pow__
pow(Integer(2), 0.5)        # Integer(1)
# Prints: Using __pow__
operator.pow(Integer(2), 3)  # Integer(8)
# Prints: Using __pow__
operator.__pow__(Integer(3), 3) # Integer(27)
# Prints: Using __pow__
```

The second argument of the \_\_pow\_\_() method can only be supplied by using the builtin-pow() or by directly calling the method:

```
pow(Integer(2), 3, 4)       # Integer(0)
# Prints: Using __pow__ with modulo
Integer(2).__pow__(3, 4)     # Integer(0)
# Prints: Using __pow__ with modulo
```

While the math-functions always convert it to a float and use the float-computation:

```
import math
```

```
math.pow(Integer(2), 0.5) # 1.4142135623730951  
# 输出：使用 __float__
```

cmath函数尝试将其转换为complex，但如果 没有显式转换为

```
import cmath  
  
cmath.exp(Integer(2)) # (7.38905609893065+0j)  
# 输出：使用 __complex__
```

```
del Integer.__complex__ # 删除 __complex__ 方法 - 实例无法转换为 complex
```

```
cmath.exp(Integer(2)) # (7.38905609893065+0j)  
# 打印：使用 __float__
```

如果缺少 \_\_float\_\_()方法，math和cmath都无法工作：

```
del Integer.__float__ # 删除 __complex__ 方法  
  
math.sqrt(Integer(2)) # 以及 cmath.exp(Integer(2))
```

TypeError: 需要一个浮点数

## 第70.9节：根：带分数指数的n次根

虽然math.sqrt函数专门用于平方根，但通常使用带分数指数的幂运算符 (\*\*\*) 来执行n次根运算（如立方根）更为方便。

幂运算的逆运算是以指数的倒数为指数的幂运算。因此，如果你可以通过将数字提升到3次方来求立方，那么你也可以通过将数字提升到1/3次方来求立方根。

```
>>> x = 3  
>>> y = x ** 3  
>>> y  
27  
>>> z = y ** (1.0 / 3)  
>>> z  
3.0  
>>> z == x  
True
```

```
math.pow(Integer(2), 0.5) # 1.4142135623730951  
# Prints: Using __float__
```

cmath-functions try to convert it to complex but can also fallback to float if there is no explicit conversion to complex. 也可以通过float：

```
import cmath  
  
cmath.exp(Integer(2)) # (7.38905609893065+0j)  
# Prints: Using __complex__  
  
del Integer.__complex__ # Deleting __complex__ method - instances cannot be cast to complex  
  
cmath.exp(Integer(2)) # (7.38905609893065+0j)  
# Prints: Using __float__
```

Neither math nor cmath will work if also the \_\_float\_\_()-method is missing:

```
del Integer.__float__ # Deleting __complex__ method  
  
math.sqrt(Integer(2)) # also cmath.exp(Integer(2))
```

TypeError: a float is required

## Section 70.9: Roots: nth-root with fractional exponents

While the math.sqrt function is provided for the specific case of square roots, it's often convenient to use the exponentiation operator (\*\*) with fractional exponents to perform nth-root operations, like cube roots.

The inverse of an exponentiation is exponentiation by the exponent's reciprocal. So, if you can cube a number by putting it to the exponent of 3, you can find the cube root of a number by putting it to the exponent of 1/3.

```
>>> x = 3  
>>> y = x ** 3  
>>> y  
27  
>>> z = y ** (1.0 / 3)  
>>> z  
3.0  
>>> z == x  
True
```

# 第71章：搜索

## 第71.1节：元素搜索

Python中所有内置集合都实现了使用in检查元素是否存在的方法。

### 列表

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist # True
10 in alist # False
```

### 元组

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple # False
'4' in atuple # True
```

### 字符串

```
astring = 'i am a string'
'a' in astring # True
'am' in astring # True
'I' in astring # False
```

### 集合

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset # True
10 in aset # False
```

### 字典

字典 有点特殊：普通的 in 只检查 键。如果你想在 值 中搜索，需要明确指定。如果你想搜索 键值对 也是一样。

```
adict = {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
1 in adict # True - 隐式搜索键
'a' in adict # False
2 in adict.keys() # True - 显式搜索键
'a' in adict.values() # True - 显式搜索值
(0, 'a') in adict.items() # True - 显式搜索键值对
```

## 第71.2节：在自定义类中搜索：`__contains__` 和 `__iter__`

为了允许在自定义类中使用 in，类必须提供魔法方法 `__contains__`，或者如果没有，则提供 `__iter__` 方法。

假设你有一个包含 list 的 list 的类：

```
class ListList:
    def __init__(self, value):
        self.value = value
        # 创建一个包含所有值的集合以便快速访问
        self.setofvalues = set(item for sublist in self.value for item in sublist)

    def __iter__(self):
        print('使用 __iter__.')
        # 遍历所有子列表元素的生成器
        return (item for sublist in self.value for item in sublist)
```

# Chapter 71: Searching

## Section 71.1: Searching for an element

All built-in collections in Python implement a way to check element membership using in.

### List

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist # True
10 in alist # False
```

### Tuple

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple # False
'4' in atuple # True
```

### String

```
astring = 'i am a string'
'a' in astring # True
'am' in astring # True
'I' in astring # False
```

### Set

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset # True
10 in aset # False
```

### Dict

`dict` 是一个有点特殊的：正常的 in 只检查键。如果你想搜索值，你需要指定它。如果你想搜索键值对，也是一样的。

```
adict = {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
1 in adict # True - implicitly searches in keys
'a' in adict # False
2 in adict.keys() # True - explicitly searches in keys
'a' in adict.values() # True - explicitly searches in values
(0, 'a') in adict.items() # True - explicitly searches key/value pairs
```

## Section 71.2: Searching in custom classes: `__contains__` and `__iter__`

To allow the use of in for custom classes the class must either provide the magic method `__contains__` or, failing that, an `__iter__`-method.

假设你有一个类包含一个 list of lists：

```
class ListList:
    def __init__(self, value):
        self.value = value
        # Create a set of all values for fast access
        self.setofvalues = set(item for sublist in self.value for item in sublist)

    def __iter__(self):
        print('Using __iter__.')
        # A generator over all sublist elements
        return (item for sublist in self.value for item in sublist)
```

```

def __contains__(self, value):
    print('使用 __contains__')
    # 仅查找值是否在集合中
    return value in self.setofvalues

# 即使没有集合，你也可以使用 iter 方法进行包含检查：
# 返回任何(item == value for item in iter(self))

```

使用成员测试可以通过in实现：

```

a = ListList([[1,1,1],[0,1,1],[1,5,1]])
10 in a    # False
# 输出：使用 __contains__。
5 in a    # True
# 输出：使用 __contains__。

```

即使删除了\_\_contains\_\_方法：

```

del ListList.__contains__
5 in a    # True
# 输出：使用 __iter__。

```

**注意：**循环中的in（如`for i in a`）总是使用`__iter__`，即使类实现了`__contains__`方法。

## 第71.3节：获取字符串索引：str.index()、str.rindex() 和 str.find()、str.rfind()

字符串也有一个`index`方法，还有更高级的选项以及额外的`str.find`。对于这两者都有一个对应的`reversed`方法。

```

astring = 'Hello on StackOverflow'
astring.index('o')    # 4
astring.rindex('o')   # 20

astring.find('o')    # 4
astring.rfind('o')   # 20

```

`index/rindex` 和 `find/rfind` 之间的区别在于如果子字符串未在字符串中找到时的处理方式：

```

astring.index('q') # ValueError: substring not found
astring.find('q')  # -1

```

所有这些方法都允许指定起始和结束索引：

```

astring.index('o', 5)    # 6
astring.index('o', 6)    # 6 - 起始索引包含在内
astring.index('o', 5, 7) # 6
astring.index('o', 5, 6) # - 结束索引不包含在内

```

ValueError: substring not found

```

astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - 仍然是从左到右

```

```

def __contains__(self, value):
    print('Using __contains__')
    # Just lookup if the value is in the set
    return value in self.setofvalues

# Even without the set you could use the iter method for the contains-check:
# return any(item == value for item in iter(self))

```

Using membership testing is possible using in:

```

a = ListList([[1,1,1],[0,1,1],[1,5,1]])
10 in a    # False
# Prints: Using __contains__.
5 in a    # True
# Prints: Using __contains__.

```

even after deleting the `__contains__` method:

```

del ListList.__contains__
5 in a    # True
# Prints: Using __iter__.

```

**Note:** The looping in (as in `for i in a`) will always use `__iter__` even if the class implements a `__contains__` method.

## Section 71.3: Getting the index for strings: str.index(), str.rindex() and str.find(), str.rfind()

String also have an `index` method but also more advanced options and the additional `str.find`. For both of these there is a complementary `reversed` method.

```

astring = 'Hello on StackOverflow'
astring.index('o')    # 4
astring.rindex('o')   # 20

astring.find('o')    # 4
astring.rfind('o')   # 20

```

The difference between `index/rindex` and `find/rfind` is what happens if the substring is not found in the string:

```

astring.index('q') # ValueError: substring not found
astring.find('q')  # -1

```

All of these methods allow a start and end index:

```

astring.index('o', 5)    # 6
astring.index('o', 6)    # 6 - start is inclusive
astring.index('o', 5, 7) # 6
astring.index('o', 5, 6) # - end is not inclusive

```

ValueError: substring not found

```

astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - still from left to right

```

```
astring.rindex('o', 4, 7) # 6
```

## 第71.4节：获取索引列表和元组：list.index(), tuple.index()

列表和元组都有一个index方法来获取元素的位置：

```
alist = [10, 16, 26, 5, 2, 19, 105, 26]
# 在列表中搜索16
alist.index(16) # 1
alist[1] # 16

alist.index(15)
```

ValueError: 15 is not in list

但只返回第一个找到元素的位置：

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26) # 2
atuple[2] # 26
atuple[7] # 26 - 也是26！
```

## 第71.5节：在字典中搜索值对应的键

字典（dict）没有内置的方法来搜索值或键，因为字典是无序的。你可以创建一个函数来获取指定值对应的键（或键列表）：

```
def getKeysForValue(dictionary, value):
foundkeys = []
for key in dictionary:
    if dictionary[key] == value:
        foundkeys.append(key)
return foundkeys
```

这也可以写成等效的列表推导式：

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

如果您只关心找到的一个键：

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

前两个函数将返回一个包含所有具有指定值的键的列表：

```
adict = {'a': 10, 'b': 20, 'c': 10}
getKeysForValue(adict, 10) # ['c', 'a'] - 顺序是随机的，也可能是 ['a', 'c']
getKeysForValueComp(adict, 10) # ['c', 'a'] - 同上
getKeysForValueComp(adict, 20) # ['b']
getKeysForValueComp(adict, 25) # []
```

另一个函数只会返回一个键：

```
astring.rindex('o', 4, 7) # 6
```

## Section 71.4: Getting the index list and tuples: list.index(), tuple.index()

`list` and `tuple` have an `index`-method to get the position of the element:

```
alist = [10, 16, 26, 5, 2, 19, 105, 26]
# search for 16 in the list
alist.index(16) # 1
alist[1] # 16

alist.index(15)
```

ValueError: 15 is not in list

But only returns the position of the first found element:

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26) # 2
atuple[2] # 26
atuple[7] # 26 - is also 26!
```

## Section 71.5: Searching key(s) for a value in dict

`dict` have no builtin method for searching a value or key because `dictionaries` are unordered. You can create a function that gets the key (or keys) for a specified value:

```
def getKeysForValue(dictionary, value):
    foundkeys = []
    for keys in dictionary:
        if dictionary[keys] == value:
            foundkeys.append(keys)
    return foundkeys
```

This could also be written as an equivalent list comprehension:

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

If you only care about one found key:

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

The first two functions will return a `list` of all keys that have the specified value:

```
adict = {'a': 10, 'b': 20, 'c': 10}
getKeysForValue(adict, 10) # ['c', 'a'] - order is random could as well be ['a', 'c']
getKeysForValueComp(adict, 10) # ['c', 'a'] - ditto
getKeysForValueComp(adict, 20) # ['b']
getKeysForValueComp(adict, 25) # []
```

The other one will only return one key:

```
getOneKeyForValue(adict, 10) # 'c' - 根据情况也可能是 'a'  
getOneKeyForValue(adict, 20) # 'b'
```

如果值不在字典中，则会抛出一个StopIteration异常：

```
getOneKeyForValue(adict, 25)
```

StopIteration

## 第71.6节：获取排序序列的索引： `bisect.bisect_left()`

排序序列允许使用更快的搜索算法：`bisect.bisect_left()`1：

```
import bisect  
  
def index_sorted(sorted_seq, value):  
    """定位最左侧与x完全相等的值，否则抛出ValueError"""\n    i = bisect.bisect_left(sorted_seq, value)  
    if i != len(sorted_seq) and sorted_seq[i] == value:  
        return i  
    raise ValueError  
  
alist = [i for i in range(1, 100000, 3)] # 从1到100000，步长为3的排序列表  
index_sorted(alist, 97285) # 32428  
index_sorted(alist, 4) # 1  
index_sorted(alist, 97286)
```

值错误

对于非常大的已排序序列，速度提升可以非常显著。对于第一次搜索，大约快500倍：

```
%timeit index_sorted(alist, 97285)  
# 100000次循环，3次中最好：每次循环3 微秒  
%timeit alist.index(97285)  
# 1000次循环，3次中最好：每次循环1.58 毫秒
```

如果元素是非常靠前的一个，速度会稍慢一些：

```
%timeit index_sorted(alist, 4)  
# 100000次循环，3次中最好：每次循环2.98 微秒  
%timeit alist.index(4)  
# 1000000 次循环，3 次中最佳：每次循环 580 纳秒
```

## 第 71.7 节：搜索嵌套序列

在嵌套序列中搜索，比如一个列表的元组，需要类似于在字典中搜索键对应值的方法  
但需要自定义函数。

如果值在序列中找到，返回最外层序列的索引：

```
getOneKeyForValue(adict, 10) # 'c' - depending on the circumstances this could also be 'a'  
getOneKeyForValue(adict, 20) # 'b'
```

and raise a `StopIteration-Exception` if the value is not in the `dict`:

```
getOneKeyForValue(adict, 25)
```

StopIteration

## Section 71.6: Getting the index for sorted sequences: `bisect.bisect_left()`

Sorted sequences allow the use of faster searching algorithms: `bisect.bisect_left()`1:

```
import bisect  
  
def index_sorted(sorted_seq, value):  
    """Locate the leftmost value exactly equal to x or raise a ValueError"""\n    i = bisect.bisect_left(sorted_seq, value)  
    if i != len(sorted_seq) and sorted_seq[i] == value:  
        return i  
    raise ValueError  
  
alist = [i for i in range(1, 100000, 3)] # Sorted list from 1 to 100000 with step 3  
index_sorted(alist, 97285) # 32428  
index_sorted(alist, 4) # 1  
index_sorted(alist, 97286)
```

ValueError

For very large **sorted sequences** the speed gain can be quite high. In case for the first search approximately 500 times as fast:

```
%timeit index_sorted(alist, 97285)  
# 100000 loops, best of 3: 3 μs per loop  
%timeit alist.index(97285)  
# 1000 loops, best of 3: 1.58 ms per loop
```

While it's a bit slower if the element is one of the very first:

```
%timeit index_sorted(alist, 4)  
# 100000 loops, best of 3: 2.98 μs per loop  
%timeit alist.index(4)  
# 1000000 loops, best of 3: 580 ns per loop
```

## Section 71.7: Searching nested sequences

Searching in nested sequences like a `list` of `tuple` requires an approach like searching the keys for values in `dict` but needs customized functions.

The index of the outermost sequence if the value was found in the sequence:

```
def outer_index(nested_sequence, value):
    return next(index for index, inner in enumerate(nested_sequence)
               for item in inner
               if item == value)

alist_of_tuples = [(4, 5, 6), (3, 1, 'a'), (7, 0, 4.3)]
outer_index(alist_of_tuples, 'a') # 1
outer_index(alist_of_tuples, 4.3) # 2
```

外部和内部序列的索引：

```
def outer_inner_index(nested_sequence, value):
    return next((oindex, iindex) for oindex, inner in enumerate(nested_sequence)
               for iindex, item in enumerate(inner)
               if item == value)

outer_inner_index(alist_of_tuples, 'a') # (1, 2)
alist_of_tuples[1][2] # 'a'

outer_inner_index(alist_of_tuples, 7) # (2, 0)
alist_of_tuples[2][0] # 7
```

一般来说（不总是）使用 `next` 和带条件的生成器表达式来查找被搜索值的第一个出现位置是最有效的方法。

```
def outer_index(nested_sequence, value):
    return next(index for index, inner in enumerate(nested_sequence)
               for item in inner
               if item == value)

alist_of_tuples = [(4, 5, 6), (3, 1, 'a'), (7, 0, 4.3)]
outer_index(alist_of_tuples, 'a') # 1
outer_index(alist_of_tuples, 4.3) # 2
```

or the index of the outer and inner sequence:

```
def outer_inner_index(nested_sequence, value):
    return next((oindex, iindex) for oindex, inner in enumerate(nested_sequence)
               for iindex, item in enumerate(inner)
               if item == value)

outer_inner_index(alist_of_tuples, 'a') # (1, 2)
alist_of_tuples[1][2] # 'a'

outer_inner_index(alist_of_tuples, 7) # (2, 0)
alist_of_tuples[2][0] # 7
```

In general (*not always*) using `next` and a **generator expression** with conditions to find the first occurrence of the searched value is the most efficient approach.

# 第72章：排序、最小值和最大值

## 第72.1节：使自定义类可排序

`min`、`max` 和 `sorted` 都需要对象是可排序的。为了正确排序，类需要定义全部6个方法 `__lt__`、`__gt__`、`__ge__`、`__le__`、`__ne__` 和 `__eq__`：

```
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - 测试小于 {!r}'.format(self, other))
        return self.value < other.value

    def __le__(self, other):
        print('{!r} - 测试小于或等于 {!r}'.format(self, other))
        return self.value <= other.value

    def __gt__(self, other):
        print('{!r} - 测试大于 {!r}'.format(self, other))
        return self.value > other.value

    def __ge__(self, other):
        print('{!r} - 测试大于或等于 {!r}'.format(self, other))
        return self.value >= other.value

    def __eq__(self, other):
        print('{!r} - 测试是否等于 {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - 测试是否不等于 {!r}'.format(self, other))
        return self.value != other.value
```

虽然实现所有这些方法似乎没有必要，但省略其中一些会使你的代码容易出现错误。

示例：

```
alist = [IntegerContainer(5), IntegerContainer(3),
         IntegerContainer(10), IntegerContainer(7)
     ]

res = max(alist)
# 输出: IntegerContainer(3) - 测试是否大于 IntegerContainer(5)
#       IntegerContainer(10) - 测试是否大于 IntegerContainer(5)
#       IntegerContainer(7) - 测试是否大于 IntegerContainer(10)
print(res)
# 输出: IntegerContainer(10)

res = min(alist)
# 输出: IntegerContainer(3) - 测试小于 IntegerContainer(5)
#       IntegerContainer(10) - 测试小于 IntegerContainer(3)
```

# Chapter 72: Sorting, Minimum and Maximum

## Section 72.1: Make custom classes orderable

`min`, `max`, 和 `sorted` 所有的对象都需要可排序。为了正确排序，类需要定义所有6个方法 `__lt__`, `__gt__`, `__ge__`, `__le__`, `__ne__` 和 `__eq__`:

```
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __le__(self, other):
        print('{!r} - Test less than or equal to {!r}'.format(self, other))
        return self.value <= other.value

    def __gt__(self, other):
        print('{!r} - Test greater than {!r}'.format(self, other))
        return self.value > other.value

    def __ge__(self, other):
        print('{!r} - Test greater than or equal to {!r}'.format(self, other))
        return self.value >= other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value
```

虽然实现所有这些方法似乎没有必要，但省略其中一些会使你的代码容易出现错误。

Examples:

```
alist = [IntegerContainer(5), IntegerContainer(3),
         IntegerContainer(10), IntegerContainer(7)
     ]

res = max(alist)
# Out: IntegerContainer(3) - Test greater than IntegerContainer(5)
#       IntegerContainer(10) - Test greater than IntegerContainer(5)
#       IntegerContainer(7) - Test greater than IntegerContainer(10)
print(res)
# Out: IntegerContainer(10)

res = min(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#       IntegerContainer(10) - Test less than IntegerContainer(3)
```

```

#     IntegerContainer(7) - 测试小于 IntegerContainer(3)
print(res)
# 输出: IntegerContainer(3)

res = sorted(alist)
# 输出: IntegerContainer(3) - 测试小于 IntegerContainer(5)
#     IntegerContainer(10) - 测试小于 IntegerContainer(3)
#     IntegerContainer(10) - 测试小于 IntegerContainer(5)
#     IntegerContainer(7) - 测试小于 IntegerContainer(5)
#     IntegerContainer(7) - 测试小于 IntegerContainer(10)
print(res)
# 输出: [IntegerContainer(3), IntegerContainer(5), IntegerContainer(7), IntegerContainer(10)]

```

sorted 使用 reverse=True 也使用 \_\_lt\_\_:

```

res = sorted(alist, reverse=True)
# 输出: IntegerContainer(10) - 测试小于 IntegerContainer(7)
#     IntegerContainer(3) - 测试小于 IntegerContainer(10)
#     IntegerContainer(3) - 测试小于 IntegerContainer(10)
#     IntegerContainer(3) - 测试小于 IntegerContainer(7)
#     IntegerContainer(5) - 测试小于 IntegerContainer(7)
#     IntegerContainer(5) - 测试小于 IntegerContainer(3)
print(res)
# 输出: [IntegerContainer(10), IntegerContainer(7), IntegerContainer(5), IntegerContainer(3)]

```

但是sorted可以使用\_\_gt\_\_作为替代，如果默认的没有实现：

```

删除IntegerContainer.__lt__ # IntegerContainer 不再实现“小于”

res = min(alist)
# 输出: IntegerContainer(5) - 测试大于 IntegerContainer(3)
#     IntegerContainer(3) - 测试大于 IntegerContainer(10)
#     IntegerContainer(3) - 测试大于 IntegerContainer(7)
print(res)
# 输出: IntegerContainer(3)

```

如果既没有实现\_\_lt\_\_也没有实现\_\_gt\_\_，排序方法将抛出TypeError异常：

```

删除IntegerContainer.__gt__ # IntegerContainer 不再实现“大于”

res = min(alist)

```

TypeError: 无法比较的类型 : IntegerContainer() < IntegerContainer()

functools.total\_ordering装饰器可以简化编写这些丰富比较方法的工作。  
如果你用 total\_ordering装饰你的类，你需要实现\_\_eq\_\_、\_\_ne\_\_以及\_\_lt\_\_、  
\_\_le\_\_、\_\_ge\_\_或\_\_gt\_\_中的一个，装饰器会补充剩下的部分：

```

import functools

@functools.total_ordering
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

```

```

#     IntegerContainer(7) - Test less than IntegerContainer(3)
print(res)
# Out: IntegerContainer(3)

res = sorted(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#     IntegerContainer(10) - Test less than IntegerContainer(3)
#     IntegerContainer(10) - Test less than IntegerContainer(5)
#     IntegerContainer(7) - Test less than IntegerContainer(5)
#     IntegerContainer(7) - Test less than IntegerContainer(10)
print(res)
# Out: [IntegerContainer(3), IntegerContainer(5), IntegerContainer(7), IntegerContainer(10)]

```

sorted with reverse=True also uses \_\_lt\_\_:

```

res = sorted(alist, reverse=True)
# Out: IntegerContainer(10) - Test less than IntegerContainer(7)
#     IntegerContainer(3) - Test less than IntegerContainer(10)
#     IntegerContainer(3) - Test less than IntegerContainer(10)
#     IntegerContainer(3) - Test less than IntegerContainer(7)
#     IntegerContainer(5) - Test less than IntegerContainer(7)
#     IntegerContainer(5) - Test less than IntegerContainer(3)
print(res)
# Out: [IntegerContainer(10), IntegerContainer(7), IntegerContainer(5), IntegerContainer(3)]

```

But sorted can use \_\_gt\_\_ instead if the default is not implemented:

```

del IntegerContainer.__lt__ # The IntegerContainer no longer implements "less than"

res = min(alist)
# Out: IntegerContainer(5) - Test greater than IntegerContainer(3)
#     IntegerContainer(3) - Test greater than IntegerContainer(10)
#     IntegerContainer(3) - Test greater than IntegerContainer(7)
print(res)
# Out: IntegerContainer(3)

```

Sorting methods will raise a TypeError if neither \_\_lt\_\_ nor \_\_gt\_\_ are implemented:

```

del IntegerContainer.__gt__ # The IntegerContainer no longer implements "greater than"

res = min(alist)

```

TypeError: unorderable types: IntegerContainer() < IntegerContainer()

functools.total\_ordering装饰器可以简化编写这些丰富比较方法的工作。  
If you decorate your class with total\_ordering, you need to implement \_\_eq\_\_, \_\_ne\_\_ and only one of the \_\_lt\_\_,  
\_\_le\_\_, \_\_ge\_\_ or \_\_gt\_\_, and the decorator will fill in the rest:

```

import functools

@functools.total_ordering
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

```

```

def __lt__(self, other):
    print('{!r} - 测试小于 {!r}'.format(self, other))
    return self.value < other.value

def __eq__(self, other):
    print('{!r} - 测试是否等于 {!r}'.format(self, other))
    return self.value == other.value

def __ne__(self, other):
    print('{!r} - 测试是否不等于 {!r}'.format(self, other))
    return self.value != other.value

```

```

IntegerContainer(5) > IntegerContainer(6)
# 输出: IntegerContainer(5) - 测试是否小于 IntegerContainer(6)
# 返回: False

IntegerContainer(6) > IntegerContainer(5)
# 输出: IntegerContainer(6) - 测试是否小于 IntegerContainer(5)
# 输出: IntegerContainer(6) - 测试是否等于 IntegerContainer(5)
# 返回 True

```

注意 > (大于) 现在最终调用了 小于 方法，有时甚至调用了 \_\_eq\_\_方法。这也意味着如果速度非常重要，你应该自己实现每个丰富的比较方法。

## 第72.2节：特殊情况：字典

获取最小值或最大值或使用 `sorted` 取决于对对象的迭代。在 `dict` 的情况下，迭代仅针对键进行：

```

adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# 输出: 'a'
max(adict)
# 输出: 'c'
sorted(adict)
# 输出: ['a', 'b', 'c']

```

要保持字典结构，必须遍历`.items()`：

```

min(adict.items())
# 输出: ('a', 3)
max(adict.items())
# 输出: ('c', 1)
sorted(adict.items())
# 输出: [('a', 3), ('b', 5), ('c', 1)]

```

对于`sorted`，可以创建一个`OrderedDict`来保持排序，同时拥有类似字典的结构：

```

from collections import OrderedDict
OrderedDict(sorted(adict.items()))
# Output: OrderedDict([('a', 3), ('b', 5), ('c', 1)])
res = OrderedDict(sorted(adict.items()))
res['a']
# Output: 3

```

**按值排序**

```

def __lt__(self, other):
    print('{!r} - Test less than {!r}'.format(self, other))
    return self.value < other.value

def __eq__(self, other):
    print('{!r} - Test equal to {!r}'.format(self, other))
    return self.value == other.value

def __ne__(self, other):
    print('{!r} - Test not equal to {!r}'.format(self, other))
    return self.value != other.value

```

```

IntegerContainer(5) > IntegerContainer(6)
# Output: IntegerContainer(5) - Test less than IntegerContainer(6)
# Returns: False

IntegerContainer(6) > IntegerContainer(5)
# Output: IntegerContainer(6) - Test less than IntegerContainer(5)
# Output: IntegerContainer(6) - Test equal to IntegerContainer(5)
# Returns True

```

Notice how the `>` (*greater than*) now ends up calling the `less than` method, and in some cases even the `__eq__` method. This also means that if speed is of great importance, you should implement each rich comparison method yourself.

## Section 72.2: Special case: dictionaries

Getting the minimum or maximum or using `sorted` depends on iterations over the object. In the case of `dict`, the iteration is only over the keys:

```

adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# Output: 'a'
max(adict)
# Output: 'c'
sorted(adict)
# Output: ['a', 'b', 'c']

```

To keep the dictionary structure, you have to iterate over the `.items()`:

```

min(adict.items())
# Output: ('a', 3)
max(adict.items())
# Output: ('c', 1)
sorted(adict.items())
# Output: [('a', 3), ('b', 5), ('c', 1)]

```

For `sorted`, you could create an `OrderedDict` to keep the sorting while having a `dict`-like structure:

```

from collections import OrderedDict
OrderedDict(sorted(adict.items()))
# Output: OrderedDict([('a', 3), ('b', 5), ('c', 1)])
res = OrderedDict(sorted(adict.items()))
res['a']
# Output: 3

```

**By value**

这同样可以通过key参数实现：

```
min(adict.items(), key=lambda x: x[1])
# Output: ('c', 1)
max(adict.items(), key=operator.itemgetter(1))
# Output: ('b', 5)
sorted(adict.items(), key=operator.itemgetter(1), reverse=True)
# Output: [('b', 5), ('a', 3), ('c', 1)]
```

## 第72.3节：使用关键参数

可以找到序列中序列的最小值/最大值：

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# 输出: (0, 10)
```

但如果你想按每个序列中的特定元素排序，可以使用key参数：

```
min(list_of_tuples, key=lambda x: x[0])      # 按第一个元素排序
# 输出: (0, 10)

min(list_of_tuples, key=lambda x: x[1])      # 按第二个元素排序
# 输出: (2, 8)

sorted(list_of_tuples, key=lambda x: x[0])    # 按第一个元素排序 (升序)
# 输出: [(0, 10), (1, 15), (2, 8)]

sorted(list_of_tuples, key=lambda x: x[1])    # 按第二个元素排序
# 输出: [(2, 8), (0, 10), (1, 15)]

import operator
# operator模块包含比lambda函数更高效的替代方案
max(list_of_tuples, key=operator.itemgetter(0)) # 按第一个元素排序
# 输出: (2, 8)

max(list_of_tuples, key=operator.itemgetter(1)) # 按第二个元素排序
# 输出: (1, 15)

sorted(list_of_tuples, key=operator.itemgetter(0), reverse=True) # 反向 (降序) 排序
# 输出: [(2, 8), (1, 15), (0, 10)]

sorted(list_of_tuples, key=operator.itemgetter(1), reverse=True) # 反向 (降序) 排序
# 输出: [(1, 15), (0, 10), (2, 8)]
```

## 第72.4节：max、min的默认参数

你不能将空序列传递给max或min：

```
min([])
```

ValueError: min() 参数是空序列

但是，在Python 3中，你可以传入关键字参数default，若序列为空则返回该值，而不是抛出异常：

Again this is possible using the key argument:

```
min(adict.items(), key=lambda x: x[1])
# Output: ('c', 1)
max(adict.items(), key=operator.itemgetter(1))
# Output: ('b', 5)
sorted(adict.items(), key=operator.itemgetter(1), reverse=True)
# Output: [('b', 5), ('a', 3), ('c', 1)]
```

## Section 72.3: Using the key argument

Finding the minimum/maximum of a sequence of sequences is possible:

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# Output: (0, 10)
```

but if you want to sort by a specific element in each sequence use the key-argument:

```
min(list_of_tuples, key=lambda x: x[0])      # Sorting by first element
# Output: (0, 10)

min(list_of_tuples, key=lambda x: x[1])      # Sorting by second element
# Output: (2, 8)

sorted(list_of_tuples, key=lambda x: x[0])    # Sorting by first element (increasing)
# Output: [(0, 10), (1, 15), (2, 8)]

sorted(list_of_tuples, key=lambda x: x[1])    # Sorting by first element
# Output: [(2, 8), (0, 10), (1, 15)]

import operator
# The operator module contains efficient alternatives to the lambda function
max(list_of_tuples, key=operator.itemgetter(0)) # Sorting by first element
# Output: (2, 8)

max(list_of_tuples, key=operator.itemgetter(1)) # Sorting by second element
# Output: (1, 15)

sorted(list_of_tuples, key=operator.itemgetter(0), reverse=True) # Reversed (decreasing)
# Output: [(2, 8), (1, 15), (0, 10)]

sorted(list_of_tuples, key=operator.itemgetter(1), reverse=True) # Reversed(decreasing)
# Output: [(1, 15), (0, 10), (2, 8)]
```

## Section 72.4: Default Argument to max, min

You can't pass an empty sequence into max or min:

```
min([])
```

ValueError: min() arg is an empty sequence

However, with Python 3, you can pass in the keyword argument default with a value that will be returned if the sequence is empty, instead of raising an exception:

```
max([], default=42)
# 输出: 42
max([], default=0)
# 输出: 0
```

## 第72.5节：获取排序序列

使用one序列：

```
sorted((7, 2, 1, 5))          # 元组
# 输出: [1, 2, 5, 7]

sorted(['c', 'A', 'b'])        # 列表
# 输出: ['A', 'b', 'c']

sorted({11, 8, 1})            # 集合
# 输出: [1, 8, 11]

sorted({'11': 5, '3': 2, '10': 15}) # 字典
# 输出: ['10', '11', '3']      # 仅迭代键

sorted('bdca')                # 字符串
# 输出: ['a', 'b', 'c', 'd']
```

结果总是一个新的列表；原始数据保持不变。

## 第72.6节：从可迭代对象中提取N个最大或N个最小的元素

要找到可迭代对象中若干个最大或最小的值，可以使用heapq模块的nlargest和nsmallest函数：

```
import heapq

# 从范围内获取5个最大元素

heapq.nlargest(5, range(10))
# 输出: [9, 8, 7, 6, 5]

heapq.nsmallest(5, range(10))
# 输出: [0, 1, 2, 3, 4]
```

这比对整个可迭代对象进行排序然后从末尾或开头切片要高效得多。内部这些函数使用了[二叉堆优先队列](#)数据结构，这对于此用例非常高效。

像min、max和sorted一样，这些函数接受可选的key关键字参数，该参数必须是一个函数，给定一个元素，返回其排序键。

下面是一个从文件中提取1000行最长行的程序：

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

这里我们打开文件，并将文件句柄f传递给nlargest。遍历文件会将文件的每一行作为一个独立的字符串；然后nlargest将每个元素（或行）传递给函数len以确定其排序键。

```
max([], default=42)
# Output: 42
max([], default=0)
# Output: 0
```

## Section 72.5: Getting a sorted sequence

Using one sequence:

```
sorted((7, 2, 1, 5))          # tuple
# Output: [1, 2, 5, 7]

sorted(['c', 'A', 'b'])        # list
# Output: ['A', 'b', 'c']

sorted({11, 8, 1})            # set
# Output: [1, 8, 11]

sorted({'11': 5, '3': 2, '10': 15}) # dict
# Output: ['10', '11', '3']      # only iterates over the keys

sorted('bdca')                # string
# Output: ['a', 'b', 'c', 'd']
```

The result is always a new [list](#); the original data remains unchanged.

## Section 72.6: Extracting N largest or N smallest items from an iterable

To find some number (more than one) of largest or smallest values of an iterable, you can use the [nlargest](#) and [nsmallest](#) of the [heapq](#) module:

```
import heapq

# get 5 largest items from the range

heapq.nlargest(5, range(10))
# Output: [9, 8, 7, 6, 5]

heapq.nsmallest(5, range(10))
# Output: [0, 1, 2, 3, 4]
```

This is much more efficient than sorting the whole iterable and then slicing from the end or beginning. Internally these functions use the [binary heap priority queue](#) data structure, which is very efficient for this use case.

Like min, max and sorted, these functions accept the optional key keyword argument, which must be a function that, given an element, returns its sort key.

Here is a program that extracts 1000 longest lines from a file:

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

Here we open the file, and pass the file handle f to nlargest. Iterating the file yields each line of the file as a separate string; nlargest then passes each element (or line) is passed to the function len to determine its sort key.

`len`, 给定一个字符串, 返回该行的字符长度。

这只需要存储一个目前为止最大的1000行的列表, 这可以与以下情况形成对比

```
longest_lines = sorted(f, key=len)[1000:]
```

这将不得不将整个文件保存在内存中。

## 第72.7节：获取多个

值中的最小值或最大值

```
min(7,2,1,5)  
# 输出: 1
```

```
max(7,2,1,5)  
# 输出: 7
```

## 第72.8节：序列的最小值和最大值

获取序列（可迭代对象）的最小值等同于访问一个已排序序列的第一个元素：

```
min([2, 7, 5])  
# 输出: 2  
sorted([2, 7, 5])[0]  
# 输出: 2
```

最大值稍微复杂一些, 因为 `sorted` 保持顺序, 而 `max` 返回第一个遇到的值。

如果没有重复元素, 最大值与排序结果的最后一个元素相同：

```
max([2, 7, 5])  
# 输出: 7  
sorted([2, 7, 5])[-1]  
# 输出: 7
```

但如果多个元素被评估为最大值, 则不一定如此：

```
class MyClass(object):  
    def __init__(self, value, name):  
        self.value = value  
        self.name = name  
  
    def __lt__(self, other):  
        return self.value < other.value  
  
    def __repr__(self):  
        return str(self.name)  
  
sorted([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])  
# 输出: [second, first, third]  
max([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])  
# 输出: first
```

允许任何包含支持<或>操作元素的可迭代对象。

`len`, given a string, returns the length of the line in characters.

This only needs storage for a list of 1000 largest lines so far, which can be contrasted with

```
longest_lines = sorted(f, key=len)[1000:]
```

which will have to hold *the entire file in memory*.

## Section 72.7: Getting the minimum or maximum of several values

```
min(7,2,1,5)  
# Output: 1
```

```
max(7,2,1,5)  
# Output: 7
```

## Section 72.8: Minimum and Maximum of a sequence

Getting the minimum of a sequence (iterable) is equivalent of accessing the first element of a `sorted` sequence:

```
min([2, 7, 5])  
# Output: 2  
sorted([2, 7, 5])[0]  
# Output: 2
```

The maximum is a bit more complicated, because `sorted` keeps order and `max` returns the first encountered value.

In case there are no duplicates the maximum is the same as the last element of the sorted return:

```
max([2, 7, 5])  
# Output: 7  
sorted([2, 7, 5])[-1]  
# Output: 7
```

But not if there are multiple elements that are evaluated as having the maximum value:

```
class MyClass(object):  
    def __init__(self, value, name):  
        self.value = value  
        self.name = name  
  
    def __lt__(self, other):  
        return self.value < other.value  
  
    def __repr__(self):  
        return str(self.name)  
  
sorted([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])  
# Output: [second, first, third]  
max([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])  
# Output: first
```

Any iterable containing elements that support < or > operations are allowed.

# 第73章：计数

## 第73.1节：统计可迭代对象中所有元素的出现次数：collections.Counter

```
from collections import Counter

c = Counter(["a", "b", "c", "d", "a", "b", "a", "c", "d"])
c
# 输出: Counter({'a': 3, 'b': 2, 'c': 2, 'd': 2})
c["a"]
# 输出: 3

c[7]      # 不在列表中 (7 出现了 0 次!)
# 输出: 0
```

collections.Counter 可以用于任何可迭代对象，并统计每个元素的出现次数。

**注意：**一个例外是如果传入的是一个 dict 或其他类似 collections.Mapping 的类，则不会统计它们，而是创建一个包含这些值的 Counter：

```
Counter({"e": 2})
# 输出: Counter({'e': 2})

Counter({"e": "e"})      # 警告 Counter 不会验证值是否为整数
# 输出: Counter({'e': "e"})
```

## 第 73.2 节：获取最常见的值： collections.Counter.most\_common()

无法使用 collections.Counter 统计 Mapping 的 键，但我们可以统计 值：

```
from collections import Counter
adict = {'a': 5, 'b': 3, 'c': 5, 'd': 2, 'e': 2, 'q': 5}
Counter(adict.values())
# 输出: Counter({2: 2, 3: 1, 5: 3})
```

最常见的元素可以通过most\_common方法获得：

```
# 按从最常见到最不常见的值排序:
Counter(adict.values()).most_common()
# 输出: [(5, 3), (2, 2), (3, 1)]

# 获取最常见的值
Counter(adict.values()).most_common(1)
# 输出: [(5, 3)]

# 获取两个最常见的值
Counter(adict.values()).most_common(2)
# 输出: [(5, 3), (2, 2)]
```

## 第73.3节：计算序列中某个元素出现的次数：list.count() 和 tuple.count()

```
alist = [1, 2, 3, 4, 1, 2, 1, 3, 4]
```

# Chapter 73: Counting

## Section 73.1: Counting all occurrence of all items in an iterable: collections.Counter

```
from collections import Counter

c = Counter(["a", "b", "c", "d", "a", "b", "a", "c", "d"])
c
# Out: Counter({'a': 3, 'b': 2, 'c': 2, 'd': 2})
c["a"]
# Out: 3

c[7]      # not in the list (7 occurred 0 times!)
# Out: 0
```

The collections.Counter can be used for any iterable and counts every occurrence for every element.

**Note:** One exception is if a dict or another collections.Mapping-like class is given, then it will not count them, rather it creates a Counter with these values:

```
Counter({"e": 2})
# Out: Counter({'e': 2})

Counter({"e": "e"})      # warning Counter does not verify the values are int
# Out: Counter({'e': "e"})
```

## Section 73.2: Getting the most common value(-s): collections.Counter.most\_common()

Counting the keys of a Mapping isn't possible with collections.Counter but we can count the values:

```
from collections import Counter
adict = {'a': 5, 'b': 3, 'c': 5, 'd': 2, 'e': 2, 'q': 5}
Counter(adict.values())
# Out: Counter({2: 2, 3: 1, 5: 3})
```

The most common elements are available by the most\_common-method:

```
# Sorting them from most-common to least-common value:
Counter(adict.values()).most_common()
# Out: [(5, 3), (2, 2), (3, 1)]

# Getting the most common value
Counter(adict.values()).most_common(1)
# Out: [(5, 3)]

# Getting the two most common values
Counter(adict.values()).most_common(2)
# Out: [(5, 3), (2, 2)]
```

## Section 73.3: Counting the occurrences of one item in a sequence: list.count() and tuple.count()

```
alist = [1, 2, 3, 4, 1, 2, 1, 3, 4]
```

```
alist.count(1)
# 输出: 3

atuple = ('熊', '黄鼠狼', '熊', '青蛙')
atuple.count('熊')
# 输出: 2
atuple.count('狐狸')
# 输出: 0
```

## 第73.4节：计算字符串中子串出现的次数：str.count()

```
astring = '这是一个短文本'
astring.count('t')
# 输出: 4
```

这甚至适用于长度超过一个字符的子串：

```
astring.count('th')
# 输出: 1
astring.count('is')
# 输出: 2
astring.count('text')
# 输出: 1
```

这在collections.Counter中是不可能的，因为它只统计单个字符：

```
from collections import Counter
Counter(astring)
# 输出: Counter({'a': 1, 'e': 1, 'h': 2, 'i': 2, 'o': 1, 'r': 1, 's': 3, 't': 4, 'x': 1})
```

## 第73.5节：在numpy数组中计数出现次数

要统计numpy数组中某个值的出现次数，可以这样做：

```
>>> import numpy as np
>>> a=np.array([0,3,4,3,5,4,7])
>>> print np.sum(a==3)
2
```

逻辑是布尔表达式会生成一个数组，其中所有请求值的位置为1，其他位置为0。因此对这些值求和就得到出现次数。此方法适用于任意形状或数据类型的数组。

我用两种方法统计numpy中所有唯一值的出现次数。Unique和bincount。Unique会自动将多维数组展平，而bincount只适用于只包含正整数的一维数组。

```
>>> unique,counts=np.unique(a,return_counts=True)
>>> print unique,counts # counts[i] 是 unique[i] 在 a 中出现的次数
[0 3 4 5 7] [1 2 2 1 1]
>>> bin_count=np.bincount(a)
>>> print bin_count # bin_count[i] 是 i 在 a 中出现的次数
[1 0 0 2 2 1 0 1]
```

如果你的数据是 numpy 数组，通常使用 numpy 方法会比将数据转换为通用方法更快得多。

```
alist.count(1)
# Out: 3

atuple = ('bear', 'weasel', 'bear', 'frog')
atuple.count('bear')
# Out: 2
atuple.count('fox')
# Out: 0
```

## Section 73.4: Counting the occurrences of a substring in a string: str.count()

```
astring = 'thisisashorttext'
astring.count('t')
# Out: 4
```

This works even for substrings longer than one character:

```
astring.count('th')
# Out: 1
astring.count('is')
# Out: 2
astring.count('text')
# Out: 1
```

which would not be possible with `collections.Counter` which only counts single characters:

```
from collections import Counter
Counter(astring)
# Out: Counter({'a': 1, 'e': 1, 'h': 2, 'i': 2, 'o': 1, 'r': 1, 's': 3, 't': 4, 'x': 1})
```

## Section 73.5: Counting occurrences in numpy array

To count the occurrences of a value in a numpy array. This will work:

```
>>> import numpy as np
>>> a=np.array([0,3,4,3,5,4,7])
>>> print np.sum(a==3)
2
```

The logic is that the boolean statement produces a array where all occurrences of the requested values are 1 and all others are zero. So summing these gives the number of occurrences. This works for arrays of any shape or dtype.

There are two methods I use to count occurrences of all unique values in numpy. Unique and bincount. Unique automatically flattens multidimensional arrays, while bincount only works with 1d arrays only containing positive integers.

```
>>> unique,counts=np.unique(a,return_counts=True)
>>> print unique,counts # counts[i] is equal to occurrences of unique[i] in a
[0 3 4 5 7] [1 2 2 1 1]
>>> bin_count=np.bincount(a)
>>> print bin_count # bin_count[i] is equal to occurrences of i in a
[1 0 0 2 2 1 0 1]
```

If your data are numpy arrays it is generally much faster to use numpy methods then to convert your data to generic methods.

# 第74章：打印函数

## 第74.1节：打印基础

在 Python 3 及更高版本中，print 是一个函数，而不是关键字。

```
print('hello world!')  
# 输出: hello world!  
  
foo = 1  
bar = 'bar'  
baz = 3.14  
  
print(foo)  
# 输出: 1  
print(bar)  
# 输出: bar  
print(baz)  
# 输出: 3.14
```

您还可以向print传递多个参数：

```
print(foo, bar, baz)  
# 输出: 1 bar 3.14
```

另一种打印多个参数的方法是使用+

```
print(str(foo) + " " + bar + " " + str(baz))  
# 输出: 1 bar 3.14
```

不过，使用+打印多个参数时需要注意的是，参数的类型应该相同。如果不先将上述示例中的参数转换为string类型，直接打印会导致错误，因为它会尝试将数字1与字符串"bar"相加，然后再与数字3.14相加。

```
# 错误示范：  
# 类型:int str float  
print(foo + bar + baz)  
# 会导致错误
```

这是因为print中的内容会先被计算：

```
print(4 + 5)  
# 输出: 9  
print("4" + "5")  
# 输出: 45  
print([4] + [5])  
# 输出: [4, 5]
```

否则，使用+对于用户阅读变量的输出非常有帮助。下面的例子中输出非常容易阅读！

下面的脚本演示了这一点

```
import random  
# 告诉 Python 引入一个生成随机数的函数  
randnum = random.randint(0, 12)
```

# Chapter 74: The Print Function

## Section 74.1: Print basics

In Python 3 and higher, **print** is a function rather than a keyword.

```
print('hello world!')  
# out: hello world!  
  
foo = 1  
bar = 'bar'  
baz = 3.14  
  
print(foo)  
# out: 1  
print(bar)  
# out: bar  
print(baz)  
# out: 3.14
```

You can also pass a number of parameters to **print**:

```
print(foo, bar, baz)  
# out: 1 bar 3.14
```

Another way to **print** multiple parameters is by using a +

```
print(str(foo) + " " + bar + " " + str(baz))  
# out: 1 bar 3.14
```

What you should be careful about when using + to print multiple parameters, though, is that the type of the parameters should be the same. Trying to print the above example without the cast to **string** first would result in an error, because it would try to add the number 1 to the string "bar" and add that to the number 3.14.

```
# Wrong:  
# type:int str float  
print(foo + bar + baz)  
# will result in an error
```

This is because the content of **print** will be evaluated first:

```
print(4 + 5)  
# out: 9  
print("4" + "5")  
# out: 45  
print([4] + [5])  
# out: [4, 5]
```

Otherwise, using a + can be very helpful for a user to read output of variables In the example below the output is very easy to read!

The script below demonstrates this

```
import random  
# telling python to include a function to create random numbers  
randnum = random.randint(0, 12)
```

```
# 生成一个0到12之间的随机数并赋值给变量  
print("随机生成的数字是 - " + str(randnum))
```

你可以通过使用end参数防止print函数自动换行：

```
print("这行末尾没有换行... ", end="")  
print("看到了吗？")  
# 输出: 这行末尾没有换行... 看到了吗？
```

如果您想写入文件，可以将其作为参数file传递：

```
with open('my_file.txt', 'w+') as my_file:  
    print("this goes to the file!", file=my_file)
```

this goes to the file!

## 第74.2节：打印参数

你不仅可以打印文本。print还有几个参数可以帮助你。

参数 sep：在参数之间放置一个字符串。

你需要打印一个用逗号或其他字符串分隔的单词列表吗？

```
>>> print('apples', 'bananas', 'cherries', sep=', ', )  
apple, bananas, cherries  
>>> print('apple', 'banana', 'cherries', sep=', ', )  
apple, banana, cherries  
>>>
```

参数 end：在结尾使用除换行符以外的内容

没有end参数时，所有print()函数都会写一行然后跳到下一行的开头。你可以将其改为不换行（使用空字符串），或者通过使用两个换行符实现段落间双倍行距。

```
>>> print("<a", end=''); print(" class='jnidn' if 1 else '', end=''); print("/>")  
<a class='jnidn'>/>  
>>> print("paragraph1", end=""); print("paragraph2")paragraph1
```

```
paragraph2  
>>>
```

参数 file：将输出发送到sys.stdout以外的其他地方。

现在你可以将文本发送到stdout、文件或StringIO，而不必在意具体是哪种。如果它像文件一样发声，它就像文件一样工作。

```
>>> def sendit(out, *values, sep=' ', end=""):  
...     print(*values, sep=sep, end=end, file=out)  
...  
>>> sendit(sys.stdout, 'apples', 'bananas', 'cherries', sep="")  
apples bananas cherries  
>>> with open("delete-me.txt", "w+") as f:  
...     sendit(f, 'apples', 'bananas', 'cherries', sep=' ', end="")
```

```
# make a random number between 0 and 12 and assign it to a variable  
print("The randomly generated number was - " + str(randnum))
```

You can prevent the print function from automatically printing a newline by using the end parameter:

```
print("this has no newline at the end of it... ", end="")  
print("see?")  
# out: this has no newline at the end of it... see?
```

If you want to write to a file, you can pass it as the parameter file:

```
with open('my_file.txt', 'w+') as my_file:  
    print("this goes to the file!", file=my_file)
```

this goes to the file!

## Section 74.2: Print parameters

You can do more than just print text. print also has several parameters to help you.

Argument sep: place a string between arguments.

Do you need to print a list of words separated by a comma or some other string?

```
>>> print('apples', 'bananas', 'cherries', sep=', ', )  
apple, bananas, cherries  
>>> print('apple', 'banana', 'cherries', sep=', ', )  
apple, banana, cherries  
>>>
```

Argument end: use something other than a newline at the end

Without the end argument, all print() functions write a line and then go to the beginning of the next line. You can change it to do nothing (use an empty string of "")，or double spacing between paragraphs by using two newlines.

```
>>> print("<a", end=''); print(" class='jnidn' if 1 else '', end=''); print("/>")  
<a class='jnidn'>/>  
>>> print("paragraph1", end="\n\n"); print("paragraph2")  
paragraph1  
paragraph2  
>>>
```

Argument file: send output to someplace other than sys.stdout.

Now you can send your text to either stdout, a file, or StringIO and not care which you are given. If it quacks like a file, it works like a file.

```
>>> def sendit(out, *values, sep=' ', end='\n'):  
...     print(*values, sep=sep, end=end, file=out)  
...  
>>> sendit(sys.stdout, 'apples', 'bananas', 'cherries', sep='\t')  
apples     bananas     cherries  
>>> with open("delete-me.txt", "w+") as f:  
...     sendit(f, 'apples', 'bananas', 'cherries', sep=' ', end='\n')
```

```
...>>> 使用open("delete-me.txt", "rt") 作为 f:  
...     打印(f.read())
```

苹果 香蕉 樱桃

>>>

还有第四个参数 flush ，它会强制刷新流。

```
...>>> with open("delete-me.txt", "rt") as f:  
...     print(f.read())
```

apples bananas cherries

>>>

There is a fourth parameter flush which will forcibly flush the stream.

# 视频：机器学习A-Z：动手Python数据科学

向两位数据科学专家学习如何用Python创建机器学习算法。包含代码模板。



- ✓ 精通Python上的机器学习
- ✓ 对多种机器学习模型有良好的直觉
- ✓ 做出准确的预测
- ✓ 进行强有力的分析
- ✓ 构建稳健的机器学习模型
- ✓ 为您的业务创造强大的附加价值
- ✓ 将机器学习用于个人目的
- ✓ 处理强化学习、自然语言处理和深度学习等特定主题
- ✓ 掌握降维等高级技术
- ✓ 了解针对每种问题应选择哪种机器学习模型
- ✓ 构建一支强大的机器学习模型军团，并知道如何组合它们来解决任何问题

今天观看 →

# VIDEO: Machine Learning A-Z: Hands-On Python In Data Science

Learn to create Machine Learning Algorithms in Python from two Data Science experts. Code templates included.



- ✓ Master Machine Learning on Python
- ✓ Have a great intuition of many Machine Learning models
- ✓ Make accurate predictions
- ✓ Make powerful analysis
- ✓ Make robust Machine Learning models
- ✓ Create strong added value to your business
- ✓ Use Machine Learning for personal purpose
- ✓ Handle specific topics like Reinforcement Learning, NLP and Deep Learning
- ✓ Handle advanced techniques like Dimensionality Reduction
- ✓ Know which Machine Learning model to choose for each type of problem
- ✓ Build an army of powerful Machine Learning models and know how to combine them to solve any problem

Watch Today →

# 第75章：正则表达式 (Regex)

Python 通过 re 模块提供正则表达式功能。

正则表达式是由字符组合而成，被解释为匹配子串的规则。例如，表达式 'amount\D+\d+' 会匹配任何由单词 amount 加上一个整数构成的字符串，中间由一个或多个非数字字符分隔，如：amount=100, amount is 3, amount is equal to: 33 等等。

## 第75.1节：匹配字符串开头

re.match() 的第一个参数是正则表达式，第二个参数是要匹配的字符串：

导入 re

```
pattern = r"123"
string = "123zzb"

re.match(pattern, string)
# Out: <_sre.SRE_Match object; span=(0, 3), match='123'>

match = re.match(pattern, string)

match.group()
# Out: '123'
```

你可能注意到 pattern 变量是一个以 r 为前缀的字符串，这表示该字符串是一个 raw string

原始字符串字面量的语法与普通字符串字面量略有不同，即在原始字符串字面量中，反斜杠 \ 表示“仅仅是一个反斜杠”，不需要通过双写反斜杠来转义诸如换行符 ()、制表符 ()、退格符 ()、换页符 (\r) 等“转义序列”。而在普通字符串字面量中，每个反斜杠都必须双写，以避免被当作转义序列的起始。

因此，r"

" 是一个包含两个字符的字符串：\ 和 n。正则表达式模式也使用反斜杠，例如 \d 表示任意数字字符。我们可以通过使用原始字符串 (r"\d") 来避免对字符串进行双重转义 ("\\d")。

例如：

```
string = "\t123zzb" # 这里反斜杠被转义了，所以没有制表符，只有 'l' 和 't'
pattern = "\t123" # 这将匹配 (转义反斜杠) 后跟 123
re.match(pattern, string).group() # 无匹配
re.match(pattern, "123zzb").group() # 匹配 '123'

pattern = r"\t123"
re.match(pattern, string).group() # 匹配 '\t123'
```

匹配仅从字符串开头开始。如果想要在任意位置匹配，请使用 re.search，示例如下：

```
match = re.match(r"(123)", "a123zzb")

match 是 None
# 输出: True

match = re.search(r"(123)", "a123zzb")

match.group()
```

# Chapter 75: Regular Expressions (Regex)

Python makes regular expressions available through the re module.

Regular expressions are combinations of characters that are interpreted as rules for matching substrings. For instance, the expression 'amount\D+\d+' will match any string composed by the word amount plus an integral number, separated by one or more non-digits, such as:amount=100, amount is 3, amount is equal to: 33, etc.

## Section 75.1: Matching the beginning of a string

The first argument of re.match() is the regular expression, the second is the string to match:

```
import re

pattern = r"123"
string = "123zzb"

re.match(pattern, string)
# Out: <_sre.SRE_Match object; span=(0, 3), match='123'>

match = re.match(pattern, string)

match.group()
# Out: '123'
```

You may notice that the pattern variable is a string prefixed with r, which indicates that the string is a raw string literal.

A raw string literal has a slightly different syntax than a string literal, namely a backslash \ in a raw string literal means "just a backslash" and there's no need for doubling up backslashes to escape "escape sequences" such as newlines (\n), tabs (\t), backspaces (\b), form-feeds (\r), and so on. In normal string literals, each backslash must be doubled up to avoid being taken as the start of an escape sequence.

Hence, r"\n" is a string of 2 characters: \ and n. Regex patterns also use backslashes, e.g. \d refers to any digit character. We can avoid having to double escape our strings ("\\d") by using raw strings (r"\d").

For instance:

```
string = "\t123zzb" # here the backslash is escaped, so there's no tab, just 'l' and 't'
pattern = "\t123" # this will match \t (escaping the backslash) followed by 123
re.match(pattern, string).group() # no match
re.match(pattern, "\t123zzb").group() # matches '\t123'

pattern = r"\t123"
re.match(pattern, string).group() # matches '\t123'
```

Matching is done from the start of the string only. If you want to match anywhere use re.search instead:

```
match = re.match(r"(123)", "a123zzb")

match 是 None
# Out: True

match = re.search(r"(123)", "a123zzb")

match.group()
```

```
# Out: '123'
```

## 第75.2节：搜索

```
pattern = r"(your base)"
sentence = "All your base are belong to us."

match = re.search(pattern, sentence)
match.group(1)
# 输出: 'your base'

match = re.search(r"(belong.*)", sentence)
match.group(1)
# 输出: 'belong to us.'
```

搜索可以在字符串的任意位置进行，这与 `re.match` 不同。你也可以使用 `re.findall`。

你也可以在字符串开头搜索（使用`^`），

```
match = re.search(r"^\d+", "123zzb")
match.group(0)
# Out: '123'

match = re.search(r"^\d+", "a123zzb")
match is None
# 输出: True
```

在字符串结尾搜索（使用`$`），

```
match = re.search(r"\d+\$", "zzb123")
match.group(0)
# Out: '123'

match = re.search(r"\d+\$", "123zzb")
match is None
# 输出: True
```

或者两者同时（同时使用`^`和`$`）：

```
match = re.search(r"^\d+\$", "123")
match.group(0)
# Out: '123'
```

## 第75.3节：预编译模式

导入 `re`

```
precompiled_pattern = re.compile(r"(\d+)")
matches = precompiled_pattern.search("The answer is 41!")
matches.group(1)
# 输出: 41

matches = precompiled_pattern.search("还是42?")
matches.group(1)
# 输出: 42
```

编译一个模式允许它在程序中重复使用。不过，请注意Python会缓存最近使用的

```
# Out: '123'
```

## Section 75.2: Searching

```
pattern = r"(your base)"
sentence = "All your base are belong to us."

match = re.search(pattern, sentence)
match.group(1)
# Out: 'your base'

match = re.search(r"(belong.*)", sentence)
match.group(1)
# Out: 'belong to us.'
```

Searching is done anywhere in the string unlike `re.match`. You can also use `re.findall`.

You can also search at the beginning of the string (use `^`)，

```
match = re.search(r"^\d+", "123zzb")
match.group(0)
# Out: '123'

match = re.search(r"^\d+", "a123zzb")
match is None
# Out: True
```

at the end of the string (use `$`)，

```
match = re.search(r"\d+\$", "zzb123")
match.group(0)
# Out: '123'

match = re.search(r"\d+\$", "123zzb")
match is None
# Out: True
```

or both (use both `^` and `$`)：

```
match = re.search(r"^\d+\$", "123")
match.group(0)
# Out: '123'
```

## Section 75.3: Precompiled patterns

`import re`

```
precompiled_pattern = re.compile(r"(\d+)")
matches = precompiled_pattern.search("The answer is 41!")
matches.group(1)
# Out: 41

matches = precompiled_pattern.search("Or was it 42?")
matches.group(1)
# Out: 42
```

Compiling a pattern allows it to be reused later on in a program. However, note that Python caches recently-used

表达式 ([docs](#), [SO answer](#))，所以“程序如果一次只使用少量正则表达式，就不必担心编译正则表达式”。

#### 导入 re

```
precompiled_pattern = re.compile(r"^\d+")
matches = precompiled_pattern.match("答案是41！")
print(matches.group(1))
# 输出: 答案是41

matches = precompiled_pattern.match("还是42？")
print(matches.group(1))
# 输出: 还是42
```

它可以与 `re.match()` 一起使用。

## 第75.4节：标志

对于某些特殊情况，我们需要改变正则表达式的行为，这可以通过标志来实现。标志可以通过`flags`关键字或直接在表达式中设置。

#### 标志关键字

下面是 `re.search` 的一个示例，但它适用于 `re` 模块中的大多数函数。

```
m = re.search("b", "ABC")
m 是None
# 输出: True

m = re.search("b", "ABC", flags=re.IGNORECASE)
m.group()
# 输出: 'B'

m = re.search("a.b", "ABC", flags=re.IGNORECASE)m 是None
# 输出: True

m = re.search("a.b", "ABC", flags=re.IGNORECASE|re.DOTALL)
m.group()
# 输出: 'AB'
```

#### 常用标志

标志	简短描述
<code>re.IGNORECASE</code> , <code>re.I</code>	使模式忽略大小写
<code>re.DOTALL</code> , <code>re.S</code>	使.匹配包括换行符在内的所有字符
<code>re.MULTILINE</code> , <code>re.M</code>	使^匹配行的开头，\$匹配行的结尾
<code>re.DEBUG</code>	开启调试信息

有关所有可用标志的完整列表，请查看文档

#### 内联标志

来自文档：[\\_\\_\\_\\_\\_](#)

(?iLmsux) (来自集合'i'、'L'、'm'、's'、'u'、'x'中的一个或多个字母。)

expressions ([docs](#), [SO answer](#))，so “programs that use only a few regular expressions at a time needn't worry about compiling regular expressions”.

#### import re

```
import re

precompiled_pattern = re.compile(r"^\d+")
matches = precompiled_pattern.match("The answer is 41!")
print(matches.group(1))
# Out: The answer is 41

matches = precompiled_pattern.match("Or was it 42?")
print(matches.group(1))
# Out: Or was it 42
```

It can be used with `re.match()`.

## Section 75.4: Flags

For some special cases we need to change the behavior of the Regular Expression, this is done using flags. Flags can be set in two ways, through the `flags` keyword or directly in the expression.

#### Flags keyword

Below an example for `re.search` but it works for most functions in the `re` module.

```
m = re.search("b", "ABC")
m is None
# Out: True

m = re.search("b", "ABC", flags=re.IGNORECASE)
m.group()
# Out: 'B'

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE)
m is None
# Out: True

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE|re.DOTALL)
m.group()
# Out: 'A\nB'
```

#### Common Flags

Flag	Short Description
<code>re.IGNORECASE</code> , <code>re.I</code>	Makes the pattern ignore the case
<code>re.DOTALL</code> , <code>re.S</code>	Makes . match everything including newlines
<code>re.MULTILINE</code> , <code>re.M</code>	Makes ^ match the begin of a line and \$ the end of a line
<code>re.DEBUG</code>	Turns on debug information

For the complete list of all available flags check the [docs](#)

#### Inline flags

From the [docs](#):

(?iLmsux) (One or more letters from the set 'i', 'L', 'm', 's', 'u', 'x'.)

该分组匹配空字符串；这些字母设置相应的标志：re.I（忽略大小写）、re.L（依赖于区域设置）、re.M（多行模式）、re.S（点匹配所有字符）、re.U（依赖于Unicode）和re.X（详细模式），作用于整个正则表达式。如果你希望将标志作为正则表达式的一部分包含，而不是作为参数传递给re.compile()函数，这非常有用。

注意，(?x)标志会改变表达式的解析方式。它应当放在表达式字符串的最前面，或者放在一个或多个空白字符之后。如果标志前面有非空白字符，结果将是不确定的。

## 第75.5节：替换

可以使用 re.sub对字符串进行替换。

### 替换字符串

```
re.sub(r"t[0-9][0-9]", "foo", "my name t13 is t44 what t99 ever t44")
# 输出: 'my name foo is foo what foo ever foo'
```

### 使用分组引用

少量分组的替换可以按如下方式进行：

```
re.sub(r"t([0-9])([0-9])", r"t\2\1", "t13 t19 t81 t25")
# 输出: 't31 t91 t18 t52'
```

但是，如果你创建一个组ID像 '10'，这不起作用：\10 被读取为 'ID号1后跟0'。所以你必须更具体地使用 \g<i> 这种表示法：

```
re.sub(r"t([0-9])([0-9])", r"t\g<2>\g<1>", "t13 t19 t81 t25")
# 输出: 't31 t91 t18 t52'
```

### 使用替换函数

```
items = ["zero", "one", "two"]
re.sub(r"a\1([0-3])\1", lambda match: items[int(match.group(1))], "Items: a[0], a[1], something,
a[2]")
# 输出: 'Items: zero, one, something, two'
```

## 第75.6节：查找所有不重叠的匹配项

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# 输出: ['12', '945', '444', '558', '889']
```

请注意，位于"[0-9]{2,3}"之前的 r告诉Python将字符串按原样解释；作为“原始”字符串。

你也可以使用 re.finditer()，它的工作方式与 re.findall()相同，但返回的是带有SRE\_Match对象的迭代器，而不是字符串列表：

```
results = re.finditer(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
print(results)
# 输出: <callable-iterator object at 0x105245890>
for result in results:
    print(result.group(0))
''' 输出:
```

The group matches the empty string; the letters set the corresponding flags: re.I (ignore case), re.L (locale dependent), re.M (multi-line), re.S (dot matches all), re.U (Unicode dependent), and re.X (verbose), for the entire regular expression. This is useful if you wish to include the flags as part of the regular expression, instead of passing a flag argument to the re.compile() function.

Note that the (?x) flag changes how the expression is parsed. It should be used first in the expression string, or after one or more whitespace characters. If there are non-whitespace characters before the flag, the results are undefined.

## Section 75.5: Replacing

Replacements can be made on strings using [re.sub](#).

### Replacing strings

```
re.sub(r"t[0-9][0-9]", "foo", "my name t13 is t44 what t99 ever t44")
# Out: 'my name foo is foo what foo ever foo'
```

### Using group references

Replacements with a small number of groups can be made as follows:

```
re.sub(r"t([0-9])([0-9])", r"t\2\1", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

However, if you make a group ID like '10', [this doesn't work](#): \10 is read as 'ID number 1 followed by 0'. So you have to be more specific and use the \g<i> notation:

```
re.sub(r"t([0-9])([0-9])", r"t\g<2>\g<1>", "t13 t19 t81 t25")
# Out: 't31 t91 t18 t52'
```

### Using a replacement function

```
items = ["zero", "one", "two"]
re.sub(r"a\1([0-3])\1", lambda match: items[int(match.group(1))], "Items: a[0], a[1], something,
a[2]")
# Out: 'Items: zero, one, something, two'
```

## Section 75.6: Find All Non-Overlapping Matches

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# Out: ['12', '945', '444', '558', '889']
```

Note that the r before "[0-9]{2,3}" tells python to interpret the string as-is; as a "raw" string.

You could also use [re.finditer\(\)](#) which works in the same way as [re.findall\(\)](#) but returns an iterator with SRE\_Match objects instead of a list of strings:

```
results = re.finditer(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
print(results)
# Out: <callable-iterator object at 0x105245890>
for result in results:
    print(result.group(0))
''' Out:
12
945
444
558
```

## 第75.7节：检查允许的字符

如果你想检查一个字符串是否只包含某些特定字符，这里是a-z、A-Z和0-9，可以这样做，

导入 re

```
def is_allowed(string):
    characterRegex = re.compile(r'^[a-zA-Z0-9.]')
    string = characterRegex.search(string)
    return not bool(string)

print(is_allowed("abYZABYZ0099"))
# Out: 'True'

print(is_allowed("#*@#$%^"))
# Out: 'False'
```

你也可以将表达式行从 `[^a-zA-Z0-9.]` 修改为 `[^a-zA-Z0-9.]`，例如禁止大写字母。

部分来源：<http://stackoverflow.com/a/1325265/2697955>

## 第75.8节：使用正则表达式拆分字符串

你也可以使用正则表达式来拆分字符串。例如，

```
import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print(data)
# 输出: ['James', '94', 'Samantha', '417', 'Scarlett', '74']
```

## 第75.9节：分组

分组是用括号完成的。调用group()会返回由匹配的括号子组组成的字符串。

```
match.group() # 不带参数的group返回找到的整个匹配
# 输出: '123'
match.group(0) # 指定0与不指定参数的结果相同
# 输出: '123'
```

也可以向group()提供参数以获取特定的子组。

来自文档：[\\_\\_\\_\\_\\_](#)

如果只有一个参数，结果是一个字符串；如果有多个参数，结果是一个包含每个参数对应项的元组。

另一方面，调用groups()会返回包含子组的元组列表。

## Section 75.7: Checking for allowed characters

If you want to check that a string contains only a certain set of characters, in this case a-z, A-Z and 0-9, you can do so like this,

```
import re

def is_allowed(string):
    characterRegex = re.compile(r'^[a-zA-Z0-9.]')
    string = characterRegex.search(string)
    return not bool(string)

print(is_allowed("abYZABYZ0099"))
# Out: 'True'

print(is_allowed("#*@#$%^"))
# Out: 'False'
```

You can also adapt the expression line from `[^a-zA-Z0-9.]` to `[^a-zA-Z0-9.]`, to disallow uppercase letters for example.

Partial credit: <http://stackoverflow.com/a/1325265/2697955>

## Section 75.8: Splitting a string using regular expressions

You can also use regular expressions to split a string. For example,

```
import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print(data)
# Output: ['James', '94', 'Samantha', '417', 'Scarlett', '74']
```

## Section 75.9: Grouping

Grouping is done with parentheses. Calling group() returns a string formed of the matching parenthesized subgroups.

```
match.group() # Group without argument returns the entire match found
# Out: '123'
match.group(0) # Specifying 0 gives the same result as specifying no argument
# Out: '123'
```

Arguments can also be provided to group() to fetch a particular subgroup.

From the [docs](#):

If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument.

Calling groups() on the other hand, returns a list of tuples containing the subgroups.

```

sentence = "这是一个电话号码 672-123-456-9910"
pattern = r".*(phone).*?([\d-]+)"

match = re.match(pattern, sentence)

match.groups() # 作为括号子组的元组列表返回整个匹配
# 输出: ('phone', '672-123-456-9910')

m.group() # 整个匹配的字符串
# 输出: 'This is a phone number 672-123-456-9910'

m.group(0) # 整个匹配的字符串
# 输出: 'This is a phone number 672-123-456-9910'

m.group(1) # 第一个带括号的子组。
# 输出: 'phone'

m.group(2) # 第二个带括号的子组。
# 输出: '672-123-456-9910'

m.group(1, 2) # 多个参数返回一个元组。
# 输出: ('phone', '672-123-456-9910')

```

### 命名组

```

match = re.search(r'My name is (?P<name>[A-Za-z ]+)', 'My name is John Smith')
match.group('name')
# 输出: 'John Smith'

match.group(1)
# 输出: 'John Smith'

```

创建一个既可以通过名称也可以通过索引引用的捕获组。

### 非捕获组

使用(?:)创建一个组，但该组不会被捕获。这意味着你可以将其用作一个组，但它不会占用你的“组空间”。

```

re.match(r'(\d+)(\+(\\d+))?', '11+22').groups()
# 输出: ('11', '+22', '22')

re.match(r'(\d+)(?:\+(\\d+))?', '11+22').groups()
# 输出: ('11', '22')

```

此示例匹配11+22或11，但不匹配11+。这是因为+符号和第二个项被分组了。另一方面，+符号没有被捕获。

## 第75.10节：转义特殊字符

特殊字符（如下方的字符类括号[和]）不会被字面匹配：

```

match = re.search(r'[b]', 'a[b]c')
match.group()
# 输出: 'b'

```

通过转义特殊字符，可以对它们进行字面匹配：

```

match = re.search(r'\[b\]', 'a[b]c')
match.group()

```

```

sentence = "This is a phone number 672-123-456-9910"
pattern = r".*(phone).*?([\d-]+)"

match = re.match(pattern, sentence)

match.groups() # The entire match as a list of tuples of the parenthesized subgroups
# Out: ('phone', '672-123-456-9910')

m.group() # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(0) # The entire match as a string
# Out: 'This is a phone number 672-123-456-9910'

m.group(1) # The first parenthesized subgroup.
# Out: 'phone'

m.group(2) # The second parenthesized subgroup.
# Out: '672-123-456-9910'

m.group(1, 2) # Multiple arguments give us a tuple.
# Out: ('phone', '672-123-456-9910')

```

### Named groups

```

match = re.search(r'My name is (?P<name>[A-Za-z ]+)', 'My name is John Smith')
match.group('name')
# Out: 'John Smith'

match.group(1)
# Out: 'John Smith'

```

Creates a capture group that can be referenced by name as well as by index.

### Non-capturing groups

Using(?:) creates a group, but the group isn't captured. This means you can use it as a group, but it won't pollute your "group space".

```

re.match(r'(\d+)(\+\(\d+))?', '11+22').groups()
# Out: ('11', '+22', '22')

re.match(r'(\d+)(?:\+\(\d+))?', '11+22').groups()
# Out: ('11', '22')

```

This example matches 11+22 or 11, but not 11+. This is since the + sign and the second term are grouped. On the other hand, the + sign isn't captured.

## Section 75.10: Escaping Special Characters

Special characters (like the character class brackets [ and ] below) are not matched literally:

```

match = re.search(r'[b]', 'a[b]c')
match.group()
# Out: 'b'

```

By escaping the special characters, they can be matched literally:

```

match = re.search(r'\[b\]', 'a[b]c')
match.group()

```

```
# 输出: '[b]'
```

可以使用 `re.escape()` 函数来帮你完成这个操作：

```
re.escape('a[b]c')
# 输出: 'a|[b||]c'
match = re.search(re.escape('a[b]c'), 'a[b]c')
match.group()
# 输出: 'a[b]c'
```

`re.escape()` 函数会转义所有特殊字符，因此当你基于用户输入构造正则表达式时非常有用：

```
username = 'A.C.' # 假设这是用户输入的
re.findall(r'Hi {}'.format(username), 'Hi A.C.! Hi ABCD!')
# 输出: ['Hi A.C!', 'Hi ABCD!']
re.findall(r'Hi {}'.format(re.escape(username)), 'Hi A.C.! Hi ABCD!')
# 输出: ['Hi A.C.']}
```

## 第75.11节：仅在特定位置匹配表达式

通常你只想在特定位置匹配一个表达式（即在其他位置保持不变）。考虑以下句子：

一天一个苹果，医生远离我（我每天吃一个苹果）。

这里“apple”出现了两次，可以通过所谓的回溯控制动词来解决，这些动词由较新的regex模块支持。其思路是：

忘记这个 | 或者 这个 | 并且 这个 也 | （但保留这个）

以我们的苹果例子来说，就是：

```
import regex as re
string = "一天一个苹果，医生远离我（我每天吃一个苹果）。"
rx = re.compile(r"""
    \([^\(\)]*\) (*SKIP)(*FAIL) # 匹配括号内的内容并“丢弃”它
    |
    # 或者
    apple # 匹配一个苹果
    "", re.VERBOSE)
apples = rx.findall(string)
print(apples)
# 只有一个
```

这只匹配当“apple”出现在括号外时的情况。

工作原理如下：

- 在从**左到右**查找时，正则表达式引擎会消耗左侧的所有内容，`(*SKIP)`充当一个“总为真断言”。随后，它在`(*FAIL)`处正确失败并回溯。
- 现在它到达了`(*SKIP)` 从右到左（即回溯时）的阶段，此时禁止继续向左移动。引擎被告知丢弃左侧的所有内容并跳转到调用`(*SKIP)`的位置。

```
# Out: '[b]'
```

The `re.escape()` function can be used to do this for you:

```
re.escape('a[b]c')
# Out: 'a\\\[b\\]c'
match = re.search(re.escape('a[b]c'), 'a[b]c')
match.group()
# Out: 'a[b]c'
```

The `re.escape()` function escapes all special characters, so it is useful if you are composing a regular expression based on user input:

```
username = 'A.C.' # suppose this came from the user
re.findall(r'Hi {}'.format(username), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C!', 'Hi ABCD!']
re.findall(r'Hi {}'.format(re.escape(username)), 'Hi A.C.! Hi ABCD!')
# Out: ['Hi A.C.'])
```

## Section 75.11: Match an expression only in specific locations

Often you want to match an expression only in *specific* places (leaving them untouched in others, that is). Consider the following sentence:

An apple a day keeps the doctor away (I eat an apple everyday).

Here the “apple” occurs twice which can be solved with so called *backtracking control verbs* which are supported by the newer `regex` module. The idea is:

`forget_this | or this | and this as well | (but keep this)`

With our apple example, this would be:

```
import regex as re
string = "An apple a day keeps the doctor away (I eat an apple everyday)."
rx = re.compile(r"""
    \([^\(\)]*\) (*SKIP)(*FAIL) # match anything in parentheses and "throw it away"
    |
    # or
    apple # match an apple
    "", re.VERBOSE)
apples = rx.findall(string)
print(apples)
# only one
```

This matches “apple” only when it can be found outside of the parentheses.

Here's how it works:

- While looking from **left to right**, the regex engine consumes everything to the left, the `(*SKIP)` acts as an “always-true-assertion”. Afterwards, it correctly fails on `(*FAIL)` and backtracks.
- Now it gets to the point of `(*SKIP)` **from right to left** (aka while backtracking) where it is forbidden to go any further to the left. Instead, the engine is told to throw away anything to the left and jump to the point where the `(*SKIP)` was invoked.

## 第75.12节：使用`re.finditer`迭代匹配项

你可以使用`re.finditer`来迭代字符串中的所有匹配项。与`re.findall`相比，它提供了额外的信息，比如匹配在字符串中的位置（索引）：

导入 `re`

```
text = '你可以尝试在这个字符串中找到一只蚂蚁'  
pattern = 'an?\w' # 查找"an", 后面可以有也可以没有一个单词字符  
  
for match in re.finditer(pattern, text):  
    # 匹配的起始索引 (整数)  
    sStart = match.start()  
  
    # 匹配的最终索引 (整数)  
    sEnd = match.end()  
  
    # 完整匹配 (字符串)  
    sGroup = match.group()  
  
    # 打印匹配结果  
    print('匹配 "{}" 发现于: [{},{}].format(sGroup, sStart,sEnd)')
```

结果：

```
匹配 "an" 发现于: [5,7]  
匹配 "an" 发现于: [20,22]  
匹配 "ant" 发现于: [23,26]
```

## Section 75.12: Iterating over matches using `re.finditer`

You can use `re.finditer` to iterate over all matches in a string. This gives you (in comparison to `re.findall` extra information, such as information about the match location in the string (indexes):

```
import re  
text = 'You can try to find an ant in this string'  
pattern = 'an?\w' # find 'an' either with or without a following word character  
  
for match in re.finditer(pattern, text):  
    # Start index of match (integer)  
    sStart = match.start()  
  
    # Final index of match (integer)  
    sEnd = match.end()  
  
    # Complete match (string)  
    sGroup = match.group()  
  
    # Print match  
    print('Match "{}" found at: [{},{}].format(sGroup, sStart,sEnd)')
```

Result:

```
Match "an" found at: [5,7]  
Match "an" found at: [20,22]  
Match "ant" found at: [23,26]
```

# 第76章：复制数据

## 第76.1节：复制字典

字典对象有一个方法copy。它执行字典的浅拷贝。

```
>>> d1 = {1:[]}
>>> d2 = d1.copy()
>>> d1 is d2
False
>>> d1[1] is d2[1]
True
```

## 第76.2节：执行浅拷贝

浅拷贝是对集合的拷贝，但不拷贝其元素。

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.copy(c)
>>> c is d
False
>>> c[0] is d[0]
True
```

## 第76.3节：执行深拷贝

如果有嵌套列表，最好也克隆嵌套列表。这个操作称为深拷贝。

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.deepcopy(c)
>>> c is d
False
>>> c[0] is d[0]
False
```

## 第76.4节：对列表执行浅拷贝

你可以使用切片来创建列表的浅拷贝。

```
>>> l1 = [1,2,3]
>>> l2 = l1[:]      # 执行浅拷贝。
>>> l2
[1,2,3]
>>> l1 is l2
False
```

## 第76.5节：拷贝集合

集合也有一个copy方法。你可以使用此方法执行浅拷贝。

```
>>> s1 = {}
>>> s2 = s1.copy()
>>> s1 is s2
False
```

# Chapter 76: Copying data

## Section 76.1: Copy a dictionary

A dictionary object has the method `copy`. It performs a shallow copy of the dictionary.

```
>>> d1 = {1:[]}
>>> d2 = d1.copy()
>>> d1 is d2
False
>>> d1[1] is d2[1]
True
```

## Section 76.2: Performing a shallow copy

A shallow copy is a copy of a collection without performing a copy of its elements.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.copy(c)
>>> c is d
False
>>> c[0] is d[0]
True
```

## Section 76.3: Performing a deep copy

If you have nested lists, it is desirable to clone the nested lists as well. This action is called deep copy.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.deepcopy(c)
>>> c is d
False
>>> c[0] is d[0]
False
```

## Section 76.4: Performing a shallow copy of a list

You can create shallow copies of lists using slices.

```
>>> l1 = [1,2,3]
>>> l2 = l1[:]      # Perform the shallow copy.
>>> l2
[1,2,3]
>>> l1 is l2
False
```

## Section 76.5: Copy a set

Sets also have a `copy`method. You can use this method to perform a shallow copy.

```
>>> s1 = {}
>>> s2 = s1.copy()
>>> s1 is s2
False
```

```
False  
>>> s2.add(3)  
>>> s1  
{[]}  
>>> s2  
{3, []}
```

```
False  
>>> s2.add(3)  
>>> s1  
{[]}  
>>> s2  
{3, []}
```

# 第77章：上下文管理器（“with”语句）

虽然Python的上下文管理器被广泛使用，但很少有人理解其使用背后的目的。这些语句通常用于读写文件，帮助应用程序节省系统内存并通过确保特定资源仅在某些进程中使用来改善资源管理。本文介绍并演示了Python上下文管理器的使用。

## 第77.1节：上下文管理器和with语句简介

上下文管理器是一个对象，当上下文（代码块）开始和结束时会收到通知。你通常会与with语句一起使用它。它负责处理通知。

例如，文件对象就是上下文管理器。当上下文结束时，文件对象会自动关闭：

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()

# open_file对象已自动关闭。
```

上述示例通常通过使用as关键字来简化：

```
with open(filename) as open_file:
    file_contents = open_file.read()

# open_file对象已自动关闭。
```

任何导致代码块执行结束的情况都会调用上下文管理器的exit方法。这包括异常，当错误导致你提前退出打开的文件或连接时，这非常有用。

在脚本退出时不正确关闭文件/连接是一个坏习惯，可能会导致数据丢失或其他问题。通过使用上下文管理器，你可以确保始终采取预防措施，以防止以这种方式造成的损害或丢失。此功能在Python 2.5中添加。

## 第77.2节：编写你自己的上下文管理器

上下文管理器是实现了两个魔法方法`_enter_()`和`_exit_()`的任何对象（尽管它也可以实现其他方法）：

```
class AContextManager():

    def __enter__(self):
        print("Entered")
        # 可选地返回一个对象
        return "A-instance"

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exited" + (" (带有异常)" if exc_type else ""))
        # 如果想要抑制异常，返回 True
```

如果上下文以异常退出，该异常的信息将作为三元组传递：`exc_type`, `exc_value`, `traceback`（这些变量与`sys.exc_info()`函数返回的相同）。如果上下文

# Chapter 77: Context Managers (“with” Statement)

While Python's context managers are widely used, few understand the purpose behind their use. These statements, commonly used with reading and writing files, assist the application in conserving system memory and improve resource management by ensuring specific resources are only in use for certain processes. This topic explains and demonstrates the use of Python's context managers.

## Section 77.1: Introduction to context managers and the with statement

A context manager is an object that is notified when a context (a block of code) starts and ends. You commonly use one with the `with` statement. It takes care of the notifying.

For example, file objects are context managers. When a context ends, the file object is closed automatically:

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

The above example is usually simplified by using the `as` keyword:

```
with open(filename) as open_file:
    file_contents = open_file.read()

# the open_file object has automatically been closed.
```

Anything that ends execution of the block causes the context manager's exit method to be called. This includes exceptions, and can be useful when an error causes you to prematurely exit from an open file or connection. Exiting a script without properly closing files/connections is a bad idea, that may cause data loss or other problems. By using a context manager you can ensure that precautions are always taken to prevent damage or loss in this way. This feature was added in Python 2.5.

## Section 77.2: Writing your own context manager

A context manager is any object that implements two magic methods `_enter_()` and `_exit_()` (although it can implement other methods as well):

```
class AContextManager():

    def __enter__(self):
        print("Entered")
        # optionally return an object
        return "A-instance"

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exited" + (" (with an exception)" if exc_type else ""))
        # return True if you want to suppress the exception
```

If the context exits with an exception, the information about that exception will be passed as a triple `exc_type`, `exc_value`, `traceback` (these are the same variables as returned by the `sys.exc_info()` function). If the context

正常退出，这三个参数都将是 `None`。

如果发生异常并传递给 `__exit__` 方法，该方法可以返回 `True` 以抑制异常，否则异常将在 `__exit__` 函数结束时重新抛出。

```
with AContextManager() as a:  
    print("a 是 %r" % a)  
# 已进入  
# a 是 'A-instance'  
# 已退出  
  
with AContextManager() as a:  
    print("a 是 %d" % a)  
# 已进入  
# 已退出 (带有异常)  
# 最近一次调用的追踪 (最近的调用最后) :  
#   文件 "<stdin>", 第 2 行, 位于 <module>  
# TypeError: %d 格式: 需要数字, 而不是字符串
```

请注意，在第二个示例中，即使在 `with` 语句体中间发生异常，`__exit__` 处理程序仍然会被执行，然后异常才会传播到外层作用域。

如果你只需要一个 `_exit__` 方法，可以返回上下文管理器的实例：

```
类 MyContextManager :  
    def __enter__(self):  
        return self  
  
    定义 __exit__(self):  
        打印('某事')
```

## 第77.3节：使用生成器语法编写你自己的上下文管理器

也可以借助 `contextlib.contextmanager` 装饰器使用生成器语法编写上下文管理器：

```
导入 contextlib  
  
@contextlib.contextmanager  
def context_manager(num):  
    打印('进入')  
    yield num + 1  
    打印('退出')  
  
使用 context_manager(2) 作为 cm：  
# 当达到上下文管理器的“yield”点时，执行以下指令。  
  
# 'cm' 将拥有 yield 语句产生的值  
print('正中间, cm = {}'.format(cm))
```

输出结果：

```
进入  
正中间, cm = 3  
退出
```

exits normally, all three of these arguments will be `None`.

If an exception occurs and is passed to the `__exit__` method, the method can return `True` in order to suppress the exception, or the exception will be re-raised at the end of the `__exit__` function.

```
with AContextManager() as a:  
    print("a 是 %r" % a)  
# Entered  
# a is 'A-instance'  
# Exited  
  
with AContextManager() as a:  
    print("a 是 %d" % a)  
# Entered  
# Exited (with an exception)  
# Traceback (most recent call last):  
#   File "<stdin>", line 2, in <module>  
#     # TypeError: %d format: a number is required, not str
```

Note that in the second example even though an exception occurs in the middle of the body of the `with`-statement, the `__exit__` handler still gets executed, before the exception propagates to the outer scope.

If you only need an `__exit__` method, you can return the instance of the context manager:

```
class MyContextManager:  
    def __enter__(self):  
        return self  
  
    def __exit__(self):  
        打印('something')
```

## Section 77.3: Writing your own contextmanager using generator syntax

It is also possible to write a context manager using generator syntax thanks to the `contextlib.contextmanager` decorator:

```
import contextlib  
  
@contextlib.contextmanager  
def context_manager(num):  
    打印('Enter')  
    yield num + 1  
    打印('Exit')  
  
with context_manager(2) as cm:  
    # the following instructions are run when the 'yield' point of the context  
    # manager is reached.  
    # 'cm' will have the value that was yielded  
    print('Right in the middle with cm = {}'.format(cm))
```

produces:

```
Enter  
Right in the middle with cm = 3  
Exit
```

该装饰器通过将生成器转换为上下文管理器，简化了编写上下文管理器的任务。yield 表达式之前的所有内容成为\_\_enter\_\_方法，yield 产生的值成为生成器返回的值（可以在 with 语句中绑定到变量），yield 表达式之后的所有内容成为\_\_exit\_\_方法。

如果上下文管理器需要处理异常，可以在生成器中编写try..except..finally代码块，任何在with代码块中引发的异常都会被该异常处理块捕获。

```
@contextlib.contextmanager
def error_handling_context_manager(num):
    print("进入")
    try:
        yield num + 1
    except ZeroDivisionError:
        print("捕获错误")
    finally:
        print("清理中")
    print("退出")

with error_handling_context_manager(-1) as cm:
    print("除以 cm = {}".format(cm))
    print(2 / cm)
```

这将产生：

```
进入
除以 cm = 0
捕获错误
清理中
退出
```

## 第77.4节：多个上下文管理器

你可以同时打开多个上下文管理器：

```
使用open(input_path) 作为 input_file，使用open(output_path, 'w') 作为 output_file：

# 对两个文件进行操作。

# 例如，将 input_file 的内容复制到 output_file 中
for line in input_file:
    output_file.write(line + "\n")
```

这与嵌套使用上下文管理器的效果相同：

```
with open(input_path) as input_file:
    with open(output_path, 'w') as output_file:
        for line in input_file:
            output_file.write(line + "\n")
```

## 第77.5节：赋值给目标

许多上下文管理器在进入时会返回一个对象。你可以在with

语句中将该对象赋值给一个新名字。

例如，使用数据库连接的with语句可以给你一个游标对象：

The decorator simplifies the task of writing a context manager by converting a generator into one. Everything before the yield expression becomes the \_\_enter\_\_ method, the value yielded becomes the value returned by the generator (which can be bound to a variable in the with statement), and everything after the yield expression becomes the \_\_exit\_\_ method.

If an exception needs to be handled by the context manager, a `try .. except .. finally`-block can be written in the generator and any exception raised in the `with`-block will be handled by this exception block.

```
@contextlib.contextmanager
def error_handling_context_manager(num):
    print("Enter")
    try:
        yield num + 1
    except ZeroDivisionError:
        print("Caught error")
    finally:
        print("Cleaning up")
    print("Exit")

with error_handling_context_manager(-1) as cm:
    print("Dividing by cm = {}".format(cm))
    print(2 / cm)
```

This produces:

```
Enter
Dividing by cm = 0
Caught error
Cleaning up
Exit
```

## Section 77.4: Multiple context managers

You can open several content managers at the same time:

```
with open(input_path) as input_file, open(output_path, 'w') as output_file:
    # do something with both files.

    # e.g. copy the contents of input_file into output_file
    for line in input_file:
        output_file.write(line + '\n')
```

It has the same effect as nesting context managers:

```
with open(input_path) as input_file:
    with open(output_path, 'w') as output_file:
        for line in input_file:
            output_file.write(line + '\n')
```

## Section 77.5: Assigning to a target

Many context managers return an object when entered. You can assign that object to a new name in the with statement.

For example, using a database connection in a with statement could give you a cursor object:

```
with database_connection as cursor:  
    cursor.execute(sql_query)
```

文件对象返回它们自身，这使得可以在一个表达式中既打开文件对象又将其用作上下文管理器：

```
with open(filename) as open_file:  
    file_contents = open_file.read()
```

## 第77.6节：管理资源

```
class File():  
    def __init__(self, filename, mode):  
        self.filename = filename  
        self.mode = mode  
  
    def __enter__(self):  
        self.open_file = open(self.filename, self.mode)  
        return self.open_file  
  
    def __exit__(self, *args):  
        self.open_file.close()
```

`__init__()` 方法用于设置对象，在本例中设置文件名和打开文件的模式。`__enter__()` 打开并返回文件，`__exit__()` 仅关闭文件。

使用这些魔法方法（`__enter__`, `__exit__`）可以让你实现可以方便地与 `with` 语句一起使用的对象。

使用 `File` 类：

```
for _ in range(10000):  
    with File('foo.txt', 'w') as f:  
        f.write('foo')
```

```
with database_connection as cursor:  
    cursor.execute(sql_query)
```

File objects return themselves, this makes it possible to both open the file object and use it as a context manager in one expression:

```
with open(filename) as open_file:  
    file_contents = open_file.read()
```

## Section 77.6: Manage Resources

```
class File():  
    def __init__(self, filename, mode):  
        self.filename = filename  
        self.mode = mode  
  
    def __enter__(self):  
        self.open_file = open(self.filename, self.mode)  
        return self.open_file  
  
    def __exit__(self, *args):  
        self.open_file.close()
```

`__init__()` method sets up the object, in this case setting up the file name and mode to open file. `__enter__()` opens and returns the file and `__exit__()` just closes it.

Using these magic methods (`__enter__`, `__exit__`) allows you to implement objects which can be used easily `with` the `with` statement.

Use `File` class:

```
for _ in range(10000):  
    with File('foo.txt', 'w') as f:  
        f.write('foo')
```

# 第78章：\_\_name\_\_ 特殊变量

\_\_name\_\_ 特殊变量用于检查文件是否作为模块被导入，并通过它们的 \_\_name\_\_ 属性识别函数、类、模块对象。

## 第78.1节：\_\_name\_\_ == '\_\_main\_\_'

特殊变量 \_\_name\_\_ 不是由用户设置的。它主要用于检查模块是被自身运行还是因为执行了 import 而被运行。为了避免模块在被导入时运行其代码的某些部分，可以检查 if \_\_name\_\_ == '\_\_main\_\_'。

假设 module\_1.py 只有一行代码：

```
import module2.py
```

让我们看看根据 module2.py

的不同情况会发生什么

### module2.py

```
print('hello')
```

运行 module1.py 会打印 hello

运行 module2.py 会打印 hello

情况 2

### module2.py

```
if __name__ == '__main__':
    print('hello')
```

运行 module1.py 不会打印任何内容

运行 module2.py 会打印 hello

## 第78.2节：在日志记录中的使用

配置内置 logging 功能时，一个常见的模式是使用当前模块的 \_\_name\_\_ 创建一个日志记录器：

```
logger = logging.getLogger(__name__)
```

这意味着模块的完全限定名称将出现在日志中，使得更容易看出消息来自哪里。

## 第78.3节：function\_class\_or\_module.\_\_name\_\_

函数、类或模块的特殊属性 \_\_name\_\_ 是一个包含其名称的字符串。

```
import os
```

# Chapter 78: The \_\_name\_\_ special variable

The \_\_name\_\_ special variable is used to check whether a file has been imported as a module or not, and to identify a function, class, module object by their \_\_name\_\_ attribute.

## Section 78.1: \_\_name\_\_ == '\_\_main\_\_'

The special variable \_\_name\_\_ is not set by the user. It is mostly used to check whether or not the module is being run by itself or run because an `import` was performed. To avoid your module to run certain parts of its code when it gets imported, check `if __name__ == '__main__'`.

Let `module_1.py` be just one line long:

```
import module2.py
```

And let's see what happens, depending on `module2.py`

### Situation 1

#### module2.py

```
print('hello')
```

Running `module1.py` will print hello

Running `module2.py` will print hello

### Situation 2

#### module2.py

```
if __name__ == '__main__':
    print('hello')
```

Running `module1.py` will print nothing

Running `module2.py` will print hello

## Section 78.2: Use in logging

When configuring the built-in `logging` functionality, a common pattern is to create a logger with the \_\_name\_\_ of the current module:

```
logger = logging.getLogger(__name__)
```

This means that the fully-qualified name of the module will appear in the logs, making it easier to see where messages have come from.

## Section 78.3: function\_class\_or\_module.\_\_name\_\_

The special attribute \_\_name\_\_ of a function, class or module is a string containing its name.

```
import os
```

```

class C:
    pass

def f(x):
    x += 2
    return x

print(f)
# <function f at 0x029976B0>
print(f.__name__)
# f

print(C)
# <class '__main__.C'>
print(C.__name__)
# C

print(os)
# <module 'os' from '/spam/eggs/'>
print(os.__name__)
# os

```

然而，`__name__` 属性并不是引用类、方法或函数的变量名，而是定义时赋予它的名称。

```

def f():
    pass

print(f.__name__)
# f - 如预期

g = f
print(g.__name__)
# f - 即使变量名为 g, 函数名仍然是 f

```

这可以用于调试等用途：

```

def enter_exit_info(func):
    def wrapper(*arg, **kw):
        print '-- entering', func.__name__
        res = func(*arg, **kw)
        print '-- exiting', func.__name__
        return res
    return wrapper

@enter_exit_info
def f(x):
    print 'In:', x
    res = x + 2
    print 'Out:', res
    return res

```

`a = f(2)`

```

# Outputs:
# -- entering f
# In: 2
# Out: 4
# -- exiting f

```

```

class C:
    pass

def f(x):
    x += 2
    return x

print(f)
# <function f at 0x029976B0>
print(f.__name__)
# f

print(C)
# <class '__main__.C'>
print(C.__name__)
# C

print(os)
# <module 'os' from '/spam/eggs/'>
print(os.__name__)
# os

```

The `__name__` attribute is not, however, the name of the variable which references the class, method or function, rather it is the name given to it when defined.

```

def f():
    pass

print(f.__name__)
# f - as expected

g = f
print(g.__name__)
# f - even though the variable is named g, the function is still named f

```

This can be used, among others, for debugging:

```

def enter_exit_info(func):
    def wrapper(*arg, **kw):
        print '-- entering', func.__name__
        res = func(*arg, **kw)
        print '-- exiting', func.__name__
        return res
    return wrapper

@enter_exit_info
def f(x):
    print 'In:', x
    res = x + 2
    print 'Out:', res
    return res

a = f(2)

# Outputs:
# -- entering f
# In: 2
# Out: 4
# -- exiting f

```

# 第79章：检查路径是否存在及权限

## 参数

	详情
os.F_OK	作为 access() 的 mode 参数传递的值，用于测试路径是否存在。
os.R_OK	包含在 access() 的 mode 参数中以测试路径的可读性的值。
os.W_OK	包含在 access() 的 mode 参数中以测试路径的可写性的值。
os.X_OK	包含在 access() 的 mode 参数中以确定路径是否可执行的值。

## 第 79.1 节：使用 os.access 进行检查

os.access 是检查目录是否存在且是否可读写的更好解决方案。

```
import os
path = "/home/myFiles/directory1"

## 检查路径是否存在
os.access(path, os.F_OK)

## 检查路径是否可读
os.access(path, os.R_OK)

## 检查路径是否可写
os.access(path, os.W_OK)

## 检查路径是否可执行
os.access(path, os.E_OK)
```

也可以同时执行所有检查

```
os.access(path, os.F_OK & os.R_OK & os.W_OK & os.E_OK)
```

以上所有方法在允许访问时返回True，不允许时返回False。这些方法在Unix和Windows系统上均可用。

# Chapter 79: Checking Path Existence and Permissions

## Parameter

	Details
os.F_OK	Value to pass as the mode parameter of access() to test the existence of path.
os.R_OK	Value to include in the mode parameter of access() to test the readability of path.
os.W_OK	Value to include in the mode parameter of access() to test the writability of path.
os.X_OK	Value to include in the mode parameter of access() to determine if path can be executed.

## Section 79.1: Perform checks using os.access

os.access is much better solution to check whether directory exists and it's accessible for reading and writing.

```
import os
path = "/home/myFiles/directory1"

## Check if path exists
os.access(path, os.F_OK)

## Check if path is Readable
os.access(path, os.R_OK)

## Check if path is Writable
os.access(path, os.W_OK)

## Check if path is Executable
os.access(path, os.E_OK)
```

also it's possible to perform all checks together

```
os.access(path, os.F_OK & os.R_OK & os.W_OK & os.E_OK)
```

All the above returns **True** if access is allowed and **False** if not allowed. These are available on unix and windows.

# 视频：机器学习 、数据科学和使用 Pyth on的深度学习

完整的动手机器学习教程，涵盖数据科学、Tensorflow、人工智能和神经网络



- ✓ 使用Tensorflow和Keras构建人工神经网络
- ✓ 使用深度学习对图像、数据和情感进行分类
- ✓ 使用线性回归、多项式回归和多元回归进行预测
- ✓ 使用Matplotlib和Seaborn进行数据可视化
- ✓ 利用Apache Spark的MLlib实现大规模机器学习
- ✓ 理解强化学习——以及如何构建吃豆人机器人
- ✓ 使用K-Means聚类、支持向量机（SVM）、K近邻（KNN）、决策树、朴素贝叶斯和主成分分析（PCA）对数据进行分类
- ✓ 使用训练/测试和K折交叉验证选择和调优模型
- ✓ 使用基于物品和基于用户的协同过滤构建电影推荐系统

今天观看 →

# VIDEO: Machine Learning, Data Science and Deep Learning with Python

Complete hands-on machine learning tutorial with data science, Tensorflow, artificial intelligence, and neural networks



- ✓ Build artificial neural networks with Tensorflow and Keras
- ✓ Classify images, data, and sentiments using deep learning
- ✓ Make predictions using linear regression, polynomial regression, and multivariate regression
- ✓ Data Visualization with Matplotlib and Seaborn
- ✓ Implement machine learning at massive scale with Apache Spark's MLlib
- ✓ Understand reinforcement learning - and how to build a Pac-Man bot
- ✓ Classify data using K-Means clustering, Support Vector Machines (SVM), KNN, Decision Trees, Naive Bayes, and PCA
- ✓ Use train/test and K-Fold cross validation to choose and tune your models
- ✓ Build a movie recommender system using item-based and user-based collaborative filtering

Watch Today →

# 第80章：创建Python包

## 第80.1节：介绍

每个包都需要一个setup.py文件来描述该包。

考虑以下简单包的目录结构：

```
+-- package_name
|   |
|   +-- __init__.py
|
+-- setup.py
```

\_\_init\_\_.py仅包含一行代码def foo(): return 100.

下面的setup.py将定义该包：

```
from setuptools import setup

setup(
    name='package_name',           # 包名
    version='0.1',                 # 版本
    description='Package Description', # 简短描述
    url='http://example.com',       # 包网址
    install_requires=[],           # 该包依赖的包列表。
    packages=['package_name'],      # 安装该包后提供的模块名称列表。
)
```

virtualenv 是测试包安装而不修改其他Python环境的好工具：

```
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
$ python setup.py install
正在运行安装
...
已安装 .../package_name-0.1- ..... egg
...
$ python
>>> import package_name
>>> package_name.foo()
100
```

## 第80.2节：上传到PyPI

一旦你的setup.py完全可用（参见介绍），上传你的包到PyPI就非常简单。

### 设置.pypirc文件

该文件存储登录名和密码以验证你的账户。它通常保存在你的主目录中。

```
# .pypirc文件
```

# Chapter 80: Creating Python packages

## Section 80.1: Introduction

Every package requires a [setup.py](#) file which describes the package.

Consider the following directory structure for a simple package:

```
+-- package_name
|   |
|   +-- __init__.py
|
+-- setup.py
```

The \_\_init\_\_.py contains only the line `def foo(): return 100`.

The following setup.py will define the package:

```
from setuptools import setup
```

```
setup(
    name='package_name',           # package name
    version='0.1',                 # version
    description='Package Description', # short description
    url='http://example.com',       # package URL
    install_requires=[],           # list of packages this package depends
                                   # on.
    packages=['package_name'],      # List of module names that installing
                                   # this package will provide.
)
```

[virtualenv](#) is great to test package installs without modifying your other Python environments:

```
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
$ python setup.py install
running install
...
Installed .../package_name-0.1-....egg
...
$ python
>>> import package_name
>>> package_name.foo()
100
```

## Section 80.2: Uploading to PyPI

Once your setup.py is fully functional (see Introduction), it is very easy to upload your package to [PyPI](#).

### Setup a .pypirc File

This file stores logins and passwords to authenticate your accounts. It is typically stored in your home directory.

```
# .pypirc file
```

```
[distutils]
index-servers =
    pypi
    pypitest

[pypi]
repository=https://pypi.python.org/pypi
username=your_username
password=你的密码

[pypitest]
repository=https://testpypi.python.org/pypi
username=你的用户名
password=你的密码
```

使用 `twine` 上传包更安全，因此请确保已安装该工具。

```
$ pip install twine
```

#### 注册并上传到 testpypi (可选)

注意: PyPI 不允许覆盖已上传的包，因此建议先在专用测试服务器上测试部署，例如 `testpypi`。将讨论此选项。  
上传前请考虑为你的包采用版本控制方案，如日历版本控制或语义版本控制。

登录或在 `testpypi` 创建新账户。注册仅需首次进行，虽然多次注册也无害。

```
$ python setup.py register -r pypitest
```

在你的包的根目录下：

```
$ twine upload dist/* -r pypitest
```

您的包现在应该可以通过您的账户访问。

#### 测试

创建一个测试虚拟环境。尝试从 `testpypi` 或 PyPI 使用 `pip install` 安装您的包。

```
# 使用 virtualenv
$ mkdir testenv
$ cd testenv
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
# 从 testpypi 测试
(.virtualenv) pip install --verbose --extra-index-url https://testpypi.python.org/pypi
package_name
...
# 或者从 PyPI 测试
(.virtualenv) $ pip install package_name
...

(.virtualenv) $ python
Python 3.5.1 (默认, 2016年1月27日, 19:16:39)
[GCC 4.2.1 兼容 Apple LLVM 7.0.2 (clang-700.1.81)] 运行于 darwin
输入 "help", "copyright", "credits" 或 "license" 获取更多信息。
>>> import package_name
```

```
[distutils]
index-servers =
    pypi
    pypitest

[pypi]
repository=https://pypi.python.org/pypi
username=your_username
password=your_password

[pypitest]
repository=https://testpypi.python.org/pypi
username=your_username
password=your_password
```

It is [safer](#) to use `twine` for uploading packages, so make sure that is installed.

```
$ pip install twine
```

#### Register and Upload to testpypi (optional)

**Note:** [PyPI does not allow overwriting uploaded packages](#), so it is prudent to first test your deployment on a dedicated test server, e.g. `testpypi`. This option will be discussed. Consider a [versioning scheme](#) for your package prior to uploading such as [calendar versioning](#) or [semantic versioning](#).

Either log in, or create a new account at [testpypi](#). Registration is only required the first time, although registering more than once is not harmful.

```
$ python setup.py register -r pypitest
```

While in the root directory of your package:

```
$ twine upload dist/* -r pypitest
```

Your package should now be accessible through your account.

#### Testing

Make a test virtual environment. Try to `pip install` your package from either `testpypi` or PyPI.

```
# Using virtualenv
$ mkdir testenv
$ cd testenv
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
# Test from testpypi
(.virtualenv) pip install --verbose --extra-index-url https://testpypi.python.org/pypi
package_name
...
# Or test from PyPI
(.virtualenv) $ pip install package_name
...

(.virtualenv) $ python
Python 3.5.1 (default, Jan 27 2016, 19:16:39)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import package_name
```

```
>>> package_name.foo()  
100
```

如果成功，您的包至少可以被导入。您也可以考虑在最终上传到 PyPI 之前测试您的 API。如果测试期间包失败，不用担心。您仍然可以修复它，重新上传到 testpypi 并再次测试。

## 注册并上传到 PyPI

确保已安装 twine：

```
$ pip install twine
```

登录或在 PyPI 创建新账户。

```
$ python setup.py register -r pypi  
$ twine upload dist/*
```

就是这样！您的包 现在已上线。

如果您发现了一个错误，只需上传您包的新版本。

## 文档

别忘了为您的包至少包含某种文档。PyPi 默认采用的格式语言是 reStructuredText。

## 自述文件

如果您的包没有大量文档，请在 README.rst 文件中包含能帮助其他用户的内容。当文件准备好后，还需要另一个文件告诉 PyPi 显示它。

创建 setup.cfg 文件，并在其中放入以下两行：

```
[metadata]  
description-file = README.rst
```

请注意，如果您尝试将 Markdown 文件放入您的包中，PyPi 会将其作为纯文本文件读取，不会进行任何格式化。

## 许可

在您的软件包中放置一个 LICENSE.txt 文件，通常是非常受欢迎的，文件中包含一个开源许可证，以告知用户他们是否可以在商业项目中使用您的软件包，或者您的代码是否可以与他们的许可证兼容。

以更易读的方式，一些许可证在 TL;DR 中进行了说明。

## 第80.3节：使包可执行

如果你的包不仅仅是一个库，而是包含一段代码，该代码在安装包时可以作为演示或独立应用程序使用，请将该代码放入 \_\_main\_\_.py 文件中。

将 \_\_main\_\_.py 放在 package\_name 文件夹中。这样你就可以直接从控制台运行它：

```
python -m package_name
```

```
>>> package_name.foo()  
100
```

If successful, your package is least importable. You might consider testing your API as well before your final upload to PyPI. If your package failed during testing, do not worry. You can still fix it, re-upload to testpypi and test again.

## Register and Upload to PyPI

Make sure twine is installed:

```
$ pip install twine
```

Either log in, or create a new account at [PyPI](#).

```
$ python setup.py register -r pypi  
$ twine upload dist/*
```

That's it! Your package is [now live](#).

If you discover a bug, simply upload a new version of your package.

## Documentation

Don't forget to include at least some kind of documentation for your package. PyPi takes as the default formatting language reStructuredText.

## Readme

If your package doesn't have a big documentation, include what can help other users in README.rst file. When the file is ready, another one is needed to tell PyPi to show it.

Create setup.cfg file and put these two lines in it:

```
[metadata]  
description-file = README.rst
```

Note that if you try to put Markdown file into your package, PyPi will read it as a pure text file without any formatting.

## Licensing

It's often more than welcome to put a LICENSE.txt file in your package with one of the [OpenSource licenses](#) to tell users if they can use your package for example in commercial projects or if your code is usable with their license.

In more readable way some licenses are explained at [TL;DR](#).

## Section 80.3: Making package executable

If your package isn't only a library, but has a piece of code that can be used either as a showcase or a standalone application when your package is installed, put that piece of code into \_\_main\_\_.py file.

Put the \_\_main\_\_.py in the package\_name folder. This way you will be able to run it directly from console:

```
python -m package_name
```

如果没有`__main__.py`文件，使用此命令将无法运行该包，并会打印以下错误：

```
python: No module named package_name.__main__; 'package_name' 是一个包，不能直接执行。
```

If there's no `__main__.py` file available, the package won't run with this command and this error will be printed:

```
python: No module named package_name.__main__; 'package_name' is a package and cannot be directly  
executed.
```

# 第81章：“pip”模块的使用：PyPI包管理器

有时你可能需要在Python内部使用pip包管理器，例如当某些导入可能引发错误时 `ImportError`，您想要处理该异常。如果您在Windows上解包 `Python_root/Scripts/pip.exe` 内部存储了 `__main__.py` 文件，其中从pip包导入了 `main` 类。这意味着每当您使用pip可执行文件时，都会使用pip包。有关将pip作为可执行文件的用法，请参见：pip：PyPI包管理器

## 第81.1节：命令使用示例

```
import pip

command = 'install'
parameter = 'selenium'
second_param = 'numpy' # 您可以根据需要提供任意数量的软件包名称
switch = '--upgrade'

pip.main([command, parameter, second_param, switch])
```

仅需的参数是必需的，因此 `pip.main(['freeze'])` 和 `pip.main(['freeze', '', ''])` 都是可接受的。

### 批量安装

可以在一次调用中传递多个软件包名称，但如果其中一个安装/升级失败，整个安装过程将停止并以状态“1”结束。

```
import pip

installed = pip.get_installed_distributions()
list = []
for i in installed:
    list.append(i.key)

pip.main(['install']+list+['--upgrade'])
```

如果你不想在某些安装失败时停止，可以在循环中调用安装。

```
for i in installed:
    pip.main(['install']+i.key+['--upgrade'])
```

## 第81.2节：处理`ImportError`异常

当你将python文件作为模块使用时，不必总是检查包是否已安装，但这对于脚本仍然有用。

```
if __name__ == '__main__':
    try:
        import requests
    except ImportError:
        print("要使用此模块，你需要安装'requests'模块")
        t = input('安装requests吗？y/n: ')
        if t == 'y':
```

# Chapter 81: Usage of "pip" module: PyPI Package Manager

Sometimes you may need to use pip package manager inside python eg. when some imports may raise `ImportError` and you want to handle the exception. If you unpack on Windows `Python_root/Scripts/pip.exe` inside is stored `__main__.py` file, where `main` class from pip package is imported. This means pip package is used whenever you use pip executable. For usage of pip as executable see: pip: PyPI Package Manager

## Section 81.1: Example use of commands

```
import pip

command = 'install'
parameter = 'selenium'
second_param = 'numpy' # You can give as many package names as needed
switch = '--upgrade'

pip.main([command, parameter, second_param, switch])
```

Only needed parameters are obligatory, so both `pip.main(['freeze'])` and `pip.main(['freeze', '', ''])` are acceptable.

### Batch install

It is possible to pass many package names in one call, but if one install/upgrade fails, whole installation process stops and ends with status '1'.

```
import pip

installed = pip.get_installed_distributions()
list = []
for i in installed:
    list.append(i.key)

pip.main(['install']+list+['--upgrade'])
```

If you don't want to stop when some installs fail, call installation in loop.

```
for i in installed:
    pip.main(['install']+i.key+['--upgrade'])
```

## Section 81.2: Handling `ImportError` Exception

When you use python file as module there is no need always check if package is installed but it is still useful for scripts.

```
if __name__ == '__main__':
    try:
        import requests
    except ImportError:
        print("To use this module you need 'requests' module")
        t = input('Install requests? y/n: ')
        if t == 'y':
```

```

import pip
pip.main(['install', 'requests'])
import requests
import os
import sys
pass
else:
    import os
    import sys
    print('某些功能可能不可用。')
else:
    import requests
    import os
    import sys

```

## 第81.3节：强制安装

例如，许多3.4版本的软件包在3.6版本上运行良好，但如果没有针对特定平台的发行版，则无法安装，但有解决方法。在.whl文件（称为wheel）中，命名约定决定是否可以在指定平台上安装软件包。例如。  
scikit\_learn-0.18.1-cp36-cp36m-win\_amd64.whl[package\_name]-[version]-[python interpreter]-[python-interpreter]-[操作系统].whl。如果wheel文件的名称被更改，使平台不匹配，pip仍会尝试安装软件包，即使平台或python版本不匹配。在最新版本的pip模块中，移除名称中的平台或解释器会导致错误kjhfkjdf.whl 不是有效的wheel文件名。

或者，可以使用7-zip等解压工具解包.whl文件。- 它通常包含分发元数据文件夹和源文件文件夹。除非该wheel包含安装脚本，否则这些源文件可以直接解压到 site-packages目录，如果包含安装脚本，则必须先运行该脚本。

```

import pip
pip.main(['install', 'requests'])
import requests
import os
import sys
pass
else:
    import os
    import sys
    print('某些功能可能不可用。')
else:
    import requests
    import os
    import sys

```

## Section 81.3: Force install

Many packages for example on version 3.4 would run on 3.6 just fine, but if there are no distributions for specific platform, they can't be installed, but there is workaround. In .whl files (known as wheels) naming convention decide whether you can install package on specified platform. Eg.  
scikit\_learn-0.18.1-cp36-cp36m-win\_amd64.whl[package\_name]-[version]-[python interpreter]-[python-interpreter]-[Operating System].whl. If name of wheel file is changed, so platform does match, pip tries to install package even if platform or python version does not match. Removing platform or interpreter from name will rise an error in newest version of pip module kjhfkjdf.whl **is not** a valid wheel filename..

Alternatively .whl file can be unpacked using an archiver as 7-zip. - It usually contains distribution meta folder and folder with source files. These source files can be simply unpacked to site-packages directory unless this wheel contain installation script, if so, it has to be run first.

# 第82章：pip：PyPI包管理器

pip 是 Python 包索引中使用最广泛的包管理器，默认安装在较新版本的 Python 中。

## 第82.1节：安装软件包

要安装名为SomePackage的软件包的最新版本：

```
$ pip install SomePackage
```

要安装软件包的特定版本：

```
$ pip install SomePackage==1.0.4
```

要指定安装软件包的最低版本：

```
$ pip install SomePackage>=1.0.4
```

如果命令在Linux/Unix上显示权限被拒绝错误，则在命令前使用sudo

### 从需求文件安装

```
$ pip install -r requirements.txt
```

需求文件的每一行都表示要安装的内容，类似于 pip install 的参数，文件格式的详细信息见：Requirements File Format。

安装包后，可以使用freeze命令进行检查：

```
$ pip freeze
```

## 第82.2节：列出所有使用 `pip` 安装的包

列出已安装的包：

```
$ pip list  
# 示例输出  
docutils (0.9.1)  
Jinja2 (2.6)  
Pygments (1.5)  
Sphinx (1.1.2)
```

列出过时的包，并显示可用的最新版本：

```
$ pip list --outdated  
# 示例输出  
docutils (当前版本: 0.9.1 最新版本: 0.10)  
Sphinx (当前版本: 1.1.2 最新版本: 1.1.3)
```

## 第82.3节：升级包

运行

# Chapter 82: pip: PyPI Package Manager

pip is the most widely-used package manager for the Python Package Index, installed by default with recent versions of Python.

## Section 82.1: Install Packages

To install the latest version of a package named SomePackage:

```
$ pip install SomePackage
```

To install a specific version of a package:

```
$ pip install SomePackage==1.0.4
```

To specify a minimum version to install for a package:

```
$ pip install SomePackage>=1.0.4
```

If commands shows permission denied error on Linux/Unix then use sudo with the commands

### Install from requirements files

```
$ pip install -r requirements.txt
```

Each line of the requirements file indicates something to be installed, and like arguments to pip install, Details on the format of the files are here: [Requirements File Format](#).

After install the package you can check it using freeze command:

```
$ pip freeze
```

## Section 82.2: To list all packages installed using `pip`

To list installed packages:

```
$ pip list  
# example output  
docutils (0.9.1)  
Jinja2 (2.6)  
Pygments (1.5)  
Sphinx (1.1.2)
```

To list outdated packages, and show the latest version available:

```
$ pip list --outdated  
# example output  
docutils (Current: 0.9.1 Latest: 0.10)  
Sphinx (Current: 1.1.2 Latest: 1.1.3)
```

## Section 82.3: Upgrade Packages

Running

```
$ pip install --upgrade SomePackage
```

将升级软件包SomePackage及其所有依赖项。此外，pip 会在升级前自动删除该软件包的旧版本。

要升级 pip 本身，请执行

```
$ pip install --upgrade pip
```

在 Unix 系统上

```
$ python -m pip install --upgrade pip
```

在 Windows 机器上。

## 第 82.4 节：卸载软件包

卸载软件包的方法：

```
$ pip uninstall SomePackage
```

## 第82.5节：在Linux上更新所有过时的软件包

pip 当前不包含允许用户一次性更新所有过时软件包的标志。然而，可以通过在Linux环境中管道连接命令来实现：

```
pip list --outdated --local | grep -v '^-\e' | cut -d = -f 1 | xargs -n1 pip install -U
```

该命令获取本地虚拟环境中的所有软件包并检查它们是否过时。从该列表中获取软件包名称，然后将其传递给pip install -U命令。此过程结束后，所有本地软件包应已更新。

## 第82.6节：在Windows上更新所有过时的软件包

pip 当前不包含允许用户一次性更新所有过时软件包的标志。然而，可以通过在Windows环境中管道连接命令来实现：

```
for /F "delims= " %i in ('pip list --outdated --local') do pip install -U %i
```

该命令获取本地虚拟环境中的所有软件包并检查它们是否过时。从该列表中获取软件包名称，然后将其传递给pip install -U命令。此过程结束后，所有本地软件包应已更新。

## 第82.7节：创建系统上所有软件包的requirements.txt文件

pip 通过提供freeze选项来协助创建requirements.txt文件。

```
pip freeze > requirements.txt
```

这将把系统上安装的所有软件包及其版本的列表保存到当前文件夹中名为requirements.txt的文件中。

```
$ pip install --upgrade SomePackage
```

will upgrade package SomePackage and all its dependencies. Also, pip automatically removes older version of the package before upgrade.

To upgrade pip itself, do

```
$ pip install --upgrade pip
```

on Unix or

```
$ python -m pip install --upgrade pip
```

on Windows machines.

## Section 82.4: Uninstall Packages

To uninstall a package:

```
$ pip uninstall SomePackage
```

## Section 82.5: Updating all outdated packages on Linux

pip doesn't current contain a flag to allow a user to update all outdated packages in one shot. However, this can be accomplished by piping commands together in a Linux environment:

```
pip list --outdated --local | grep -v '^-\e' | cut -d = -f 1 | xargs -n1 pip install -U
```

This command takes all packages in the local virtualenv and checks if they are outdated. From that list, it gets the package name and then pipes that to a pip install -U command. At the end of this process, all local packages should be updated.

## Section 82.6: Updating all outdated packages on Windows

pip doesn't current contain a flag to allow a user to update all outdated packages in one shot. However, this can be accomplished by piping commands together in a Windows environment:

```
for /F "delims= " %i in ('pip list --outdated --local') do pip install -U %i
```

This command takes all packages in the local virtualenv and checks if they are outdated. From that list, it gets the package name and then pipes that to a pip install -U command. At the end of this process, all local packages should be updated.

## Section 82.7: Create a requirements.txt file of all packages on the system

pip assists in creating requirements.txt files by providing the [freeze](#) option.

```
pip freeze > requirements.txt
```

This will save a list of all packages and their version installed on the system to a file named requirements.txt in the current folder.

## 第82.8节：使用特定版本的Python和pip

如果你同时安装了Python 3和Python 2，可以指定pip使用哪个版本的Python。当软件包仅支持Python 2或3，或者你希望同时测试两者时，这非常有用。

如果你想为Python 2安装软件包，运行以下任一命令：

```
pip install [package]
```

或者：

```
pip2 install [package]
```

如果你想为Python 3安装软件包，执行：

```
pip3 install [package]
```

您也可以通过以下方式调用安装包到特定的 Python 安装环境：

```
\pathohat\python.exe -m pip install some_package # 在 Windows 上 或  
/usr/bin/python25 -m pip install some_package # 在 OS-X/Linux 上
```

在 OS-X/Linux/Unix 平台上，重要的是要区分系统版本的 Python（升级它可能会导致系统无法运行）和用户版本的 Python。您可能需要根据您尝试升级的版本，在这些命令前加上 sudo 并输入密码。

同样在 Windows 上，某些 Python 安装，尤其是作为其他软件包一部分的安装，可能会安装在系统目录中——这些需要您以管理员模式运行命令窗口进行升级——如果您发现需要这样做，强烈建议使用命令如 `python -c"import sys;print(sys.path);"` 或 `py -3.5 -c"import sys;print(sys.path);"` 来检查您要升级的是哪个 Python 安装，也可以用 `pip --version` 来检查您正在运行的是哪个 pip。

在 Windows 上，如果您同时安装了 Python 2 和 Python 3，并且 Python 3 版本大于 3.4，那么系统路径中可能也会有 Python 启动器 py。您可以使用如下技巧：

```
py -3 -m pip install -U some_package # 安装/升级 some_package 到最新的 Python 3 版本  
py -3.3 -m pip install -U some_package # 安装/升级 some_package 到 Python 3.3 (如果存在)  
py -2 -m pip install -U some_package # 安装/升级 some_package 到最新的 Python 2 - 64 位版本 (如果存在)
```

```
py -2.7-32 -m pip install -U some_package # 安装/升级 some_package 到 Python 2.7 - 32 位版本 (如果存在)
```

如果您运行和维护多个 Python 版本，强烈建议您了解 Python 的 `virtualenv` 或 `venv` 虚拟环境，这些工具允许您隔离 Python 版本和所安装的包。

## 第 82.9 节：仅创建当前虚拟环境中的包的 requirements.txt 文件

pip 通过提供freeze选项来协助创建requirements.txt文件。

```
pip freeze --local > requirements.txt
```

--local 参数只会输出安装在虚拟环境中的软件包及其版本列表。

## Section 82.8: Using a certain Python version with pip

If you have both Python 3 and Python 2 installed, you can specify which version of Python you would like pip to use. This is useful when packages only support Python 2 or 3 or when you wish to test with both.

If you want to install packages for Python 2, run either:

```
pip install [package]
```

or:

```
pip2 install [package]
```

If you would like to install packages for Python 3, do:

```
pip3 install [package]
```

You can also invoke installation of a package to a specific python installation with:

```
\path\to\that\python.exe -m pip install some_package # on Windows OR  
/usr/bin/python25 -m pip install some_package # on OS-X/Linux
```

On OS-X/Linux/Unix platforms it is important to be aware of the distinction between the system version of python, (which upgrading make render your system inoperable), and the user version(s) of python. You **may**, depending on which you are trying to upgrade, need to prefix these commands with sudo and input a password.

Likewise on Windows some python installations, especially those that are a part of another package, can end up installed in system directories - those you will have to upgrade from a command window running in Admin mode - if you find that it looks like you need to do this it is a **very** good idea to check which python installation you are trying to upgrade with a command such as `python -c"import sys;print(sys.path);"` or `py -3.5 -c"import sys;print(sys.path);"` you can also check which pip you are trying to run with `pip --version`

On Windows, if you have both python 2 and python 3 installed, and on your path and your python 3 is greater than 3.4 then you will probably also have the python launcher py on your system path. You can then do tricks like:

```
py -3 -m pip install -U some_package # Install/Upgrade some_package to the latest python 3  
py -3.3 -m pip install -U some_package # Install/Upgrade some_package to python 3.3 if present  
py -2 -m pip install -U some_package # Install/Upgrade some_package to the latest python 2 - 64 bit  
if present  
py -2.7-32 -m pip install -U some_package # Install/Upgrade some_package to python 2.7 - 32 bit if  
present
```

If you are running & maintaining multiple versions of python I would strongly recommend reading up about the python `virtualenv` or `venv` [virtual environments](#) which allow you to isolate both the version of python and which packages are present.

## Section 82.9: Create a requirements.txt file of packages only in the current virtualenv

pip assists in creating requirements.txt files by providing the [freeze](#) option.

```
pip freeze --local > requirements.txt
```

The [--local](#) parameter will only output a list of packages and versions that are installed locally to a virtualenv.

全局包将不会被列出。

## 第82.10节：安装尚未作为wheel发布在pip上的包

许多纯Python包尚未作为wheel发布在Python包索引上，但仍然可以正常安装。

然而，某些Windows上的包会出现令人头疼的vcvarsall.bat未找到错误。

问题在于你尝试安装的包包含C或C++扩展，并且目前在Python包索引pypi上没有预编译的wheel可用，而在Windows上你没有构建此类项目所需的工具链。

最简单的解决办法是访问Christoph Gohlke的优秀网站，找到你需要的库的合适版本。包名中的“-cp NN-”必须与你的Python版本匹配，例如，如果你使用的是Windows 32位Python（即使是在Win64上），名称中必须包含“-win32-”，如果使用64位Python，则必须包含“-win\_amd64-”，然后Python版本也必须匹配，例如Python 3.4的文件名必须包含“-cp34-”，等等。这基本上就是pip在pypi网站上为你完成的魔法。

或者，你需要获取与你使用的Python版本相对应的Windows开发工具包，任何你尝试构建的包所接口的库的头文件，可能还包括对应Python版本的Python头文件等。

Python 2.7使用Visual Studio 2008，Python 3.3和3.4使用Visual Studio 2010，Python 3.5及以上版本使用Visual Studio 2015。

- 安装“[Visual C++ Compiler Package for Python 2.7](#)”，该软件包可从微软网站获得，或者安装“Windows SDK for Windows 7 and .NET Framework 4”(v7.1)，该软件包也可从微软网站获得，或者
- 安装[Visual Studio 2015社区版](#)，（或任何更高版本，发布后），确保你选择安装C和C++支持的选项不再是默认 - 据说下载和安装可能需要8小时，所以请确保第一次尝试时这些选项已设置。

然后你可能需要找到与你所需库匹配版本的头文件，并将其下载到合适的位置。

最后你可以让pip来构建——当然，如果该包有你尚未安装的依赖项，你可能还需要找到它们的头文件。

替代方案：同样值得关注的是，无论是在pypi还是Christop的网站上，寻找你所需包的稍早版本，这些版本可能是纯Python实现或为你的平台和Python版本预编译的，如果找到，可以先使用这些版本，直到你的包正式发布。同样，如果你使用的是最新版本的Python，可能需要一些时间让包维护者跟进，所以对于确实需要特定包的项目，暂时可能需要使用稍旧版本的Python。你也可以查看包的源码站点，看看是否有预编译或纯Python的分支版本，或者寻找提供你所需功能但可用的替代包——一个例子是Pillow，这是一个积极维护的替代PIL的包，后者已经6年未更新且不支持Python 3。

后记，我鼓励遇到此问题的任何人前往该包的错误跟踪系统，添加或提出（如果尚无）一个工单，礼貌地请求包维护者为你的特定平台和Python组合在pypi上提供wheel包，如果这样做，通常情况会随着时间改善，有些包维护者并未意识到他们遗漏了某些用户可能使用的组合。

Global packages will not be listed.

## Section 82.10: Installing packages not yet on pip as wheels

Many, pure python, packages are not yet available on the Python Package Index as wheels but still install fine. However, some packages on Windows give the dreaded vcvarsall.bat not found error.

The problem is that the package that you are trying to install contains a C or C++ extension and is not *currently* available as a pre-built wheel from the python package index, pypi, and on windows you do not have the tool chain needed to build such items.

The simplest answer is to go to [Christoph Gohlke's](#) excellent site and locate the **appropriate** version of the libraries that you need. By appropriate in the package name a **-cpNN-** has to match your version of python, i.e. if you are using windows 32 bit python *even on win64* the name must include **-win32-** and if using the 64 bit python it must include **-win\_amd64-** and then the python version must match, i.e. for Python 34 the filename **must** include **-cp34-**, etc. this is basically the magic that pip does for you on the pypi site.

Alternatively, you need to get the appropriate windows development kit for the version of python that you are using, the headers for any library that the package you are trying to build interfaces to, possibly the python headers for the version of python, etc.

Python 2.7 used Visual Studio 2008, Python 3.3 and 3.4 used Visual Studio 2010, and Python 3.5+ uses Visual Studio 2015.

- Install “[Visual C++ Compiler Package for Python 2.7](#)”, which is available from Microsoft’s website **or**
- Install “[Windows SDK for Windows 7 and .NET Framework 4](#)”(v7.1), which is available from Microsoft’s website **or**
- Install [Visual Studio 2015 Community Edition](#), (or any later version, when these are released), **ensuring you select the options to install C & C++ support** no longer the default - I am told that this can take up to **8 hours** to download and install so make **sure** that those options are set on the first try.

**Then** you *may need* to locate the header files, *at the matching revision* for any libraries that your desired package links to and download those to an appropriate locations.

**Finally** you can let pip do your build - of course if the package has dependencies that you don’t yet have you may also need to find the header files for them as well.

**Alternatives:** It is also worth looking out, *both on pypi or Christop’s site*, for any slightly earlier version of the package that you are looking for that is either pure python or pre-built for your platform and python version and possibly using those, if found, until your package does become available. Likewise if you are using the very latest version of python you may find that it takes the package maintainers a little time to catch up so for projects that really **need** a specific package you may have to use a slightly older python for the moment. You can also check the packages source site to see if there is a forked version that is available pre-built or as pure python and searching for alternative packages that provide the functionality that you require but are available - one example that springs to mind is the [Pillow](#), actively maintained, drop in replacement for [PIL](#) currently not updated in 6 years and not available for python 3.

**Afterword**, I would encourage anybody who is having this problem to go to the bug tracker for the package and add to, or raise if there isn’t one already, a ticket **politely** requesting that the package maintainers provide a wheel on pypi for your specific combination of platform and python, if this is done then normally things will get better with time, some package maintainers don’t realise that they have missed a given combination that people may be using.

## 关于安装预发布版本的说明

Pip遵循语义化版本控制规则， 默认偏好已发布的包而非预发布版本。因此，如果某个包已发布为V0.98且还有一个候选发布版本V1.0-rc1， pip install 的默认行为是安装V0.98——如果你想安装候选版本，建议先在虚拟环境中测试，可以通过--pip install --pre 包名或--pip install --pre --upgrade 包名来启用安装。在许多情况下，预发布或候选版本可能没有为所有平台和版本组合构建wheel包，因此你更可能遇到上述问题。

## 关于安装开发版本的说明

你也可以使用pip从github和其他位置安装包的开发版本，由于此类代码处于不断变化中，几乎不可能有为其构建的wheel包，因此任何非纯净包都需要构建工具的支持，且可能随时出现问题，强烈建议用户仅在虚拟环境中安装此类包。

此类装置有三种选择：

1. 下载压缩快照，大多数在线版本控制系统都提供下载代码压缩快照的选项。可以手动下载，然后使用pip install 安装  
path/to/downloaded/file 注意对于大多数压缩格式，pip会处理解压到缓存区等操作。
2. 让pip帮你下载并安装：pip install URL/of/package/repository - 你可能还需要使用--trusted-host、--client-cert和/or--proxy标志才能正确工作，尤其是在企业环境中。例如：

```
> py -3.5-32 -m venv demo-pip
> demo-pip\Scripts\activate.bat
> python -m pip install -U pip
正在收集 pip
使用缓存的 pip-9.0.1-py2.py3-none-any.whl
正在安装收集的包：pip
发现已存在安装：pip 8.1.1
正在卸载 pip-8.1.1：
成功卸载 pip-8.1.1
成功安装 pip-9.0.1
> pip install git+https://github.com/sphinx-doc/sphinx/
正在收集 git+https://github.com/sphinx-doc/sphinx/
正在克隆 https://github.com/sphinx-doc/sphinx/ 到 c:\users\steve-
~1\appdata\local\temp\pip-04yn9hpp-build
正在收集 six>=1.5 (来自 Sphinx==1.7.dev20170506)
使用缓存的 six-1.10.0-py2.py3-none-any.whl
正在收集 Jinja2>=2.3 (来自 Sphinx==1.7.dev20170506)
使用缓存的 Jinja2-2.9.6-py2.py3-none-any.whl
正在收集 Pygments>=2.0 (来自 Sphinx==1.7.dev20170506)
使用缓存的 Pygments-2.2.0-py2.py3-none-any.whl
正在收集 docutils>=0.11 (来自 Sphinx==1.7.dev20170506)
使用缓存的 docutils-0.13.1-py3-none-any.whl
正在收集 snowballstemmer>=1.1 (来自 Sphinx==1.7.dev20170506)
使用缓存的 snowballstemmer-1.2.1-py2.py3-none-any.whl
正在收集 babel!=2.0,>=1.3 (来自 Sphinx==1.7.dev20170506)
使用缓存的 Babel-2.4.0-py2.py3-none-any.whl
正在收集 alabaster<0.8,>=0.7 (来自 Sphinx==1.7.dev20170506)
使用缓存的 alabaster-0.7.10-py2.py3-none-any.whl
正在收集 imagesize (来自 Sphinx==1.7.dev20170506)
使用缓存的 imagesize-0.7.1-py2.py3-none-any.whl
正在收集 requests>=2.0.0 (来自 Sphinx==1.7.dev20170506)
使用缓存的 requests-2.13.0-py2.py3-none-any.whl
```

## Note on Installing Pre-Releases

Pip follows the rules of [Semantic Versioning](#) and by default prefers released packages over pre-releases. So if a given package has been released as V0.98 and there is also a release candidate V1.0-rc1 the default behaviour of pip install will be to install V0.98 - if you wish to install the release candidate, *you are advised to test in a virtual environment first*, you can enable do so with --pip install --pre package-name or --pip install --pre --upgrade package-name. In many cases pre-releases or release candidates may not have wheels built for all platform & version combinations so you are more likely to encounter the issues above.

## Note on Installing Development Versions

You can also use pip to install development versions of packages from github and other locations, since such code is in flux it is very unlikely to have wheels built for it, so any impure packages will require the presence of the build tools, and they may be broken at any time so the user is **strongly** encouraged to only install such packages in a virtual environment.

Three options exist for such installations:

1. Download compressed snapshot, most online version control systems have the option to download a compressed snapshot of the code. This can be downloaded manually and then installed with pip install path/to/downloaded/file note that for most compression formats pip will handle unpacking to a cache area, etc.
2. Let pip handle the download & install for you with: pip install URL/of/package/repository - you may also need to use the --trusted-host, --client-cert and/or --proxy flags for this to work correctly, especially in a corporate environment. e.g:

```
> py -3.5-32 -m venv demo-pip
> demo-pip\Scripts\activate.bat
> python -m pip install -U pip
Collecting pip
Using cached pip-9.0.1-py2.py3-none-any.whl
Installing collected packages: pip
Found existing installation: pip 8.1.1
Uninstalling pip-8.1.1:
Successfully uninstalled pip-8.1.1
Successfully installed pip-9.0.1
> pip install git+https://github.com/sphinx-doc/sphinx/
Collecting git+https://github.com/sphinx-doc/sphinx/
Cloning https://github.com/sphinx-doc/sphinx/ to c:\users\steve-
~1\appdata\local\temp\pip-04yn9hpp-build
Collecting six>=1.5 (from Sphinx==1.7.dev20170506)
Using cached six-1.10.0-py2.py3-none-any.whl
Collecting Jinja2>=2.3 (from Sphinx==1.7.dev20170506)
Using cached Jinja2-2.9.6-py2.py3-none-any.whl
Collecting Pygments>=2.0 (from Sphinx==1.7.dev20170506)
Using cached Pygments-2.2.0-py2.py3-none-any.whl
Collecting docutils>=0.11 (from Sphinx==1.7.dev20170506)
Using cached docutils-0.13.1-py3-none-any.whl
Collecting snowballstemmer>=1.1 (from Sphinx==1.7.dev20170506)
Using cached snowballstemmer-1.2.1-py2.py3-none-any.whl
Collecting babel!=2.0,>=1.3 (from Sphinx==1.7.dev20170506)
Using cached Babel-2.4.0-py2.py3-none-any.whl
Collecting alabaster<0.8,>=0.7 (from Sphinx==1.7.dev20170506)
Using cached alabaster-0.7.10-py2.py3-none-any.whl
Collecting imagesize (from Sphinx==1.7.dev20170506)
Using cached imagesize-0.7.1-py2.py3-none-any.whl
Collecting requests>=2.0.0 (from Sphinx==1.7.dev20170506)
Using cached requests-2.13.0-py2.py3-none-any.whl
```

```

正在收集 typing (来自 Sphinx==1.7.dev20170506)
使用缓存的 typing-3.6.1.tar.gz
已满足需求：setuptools 在 f:\toolbuild\temp\demo-pip\lib\site-packages 中 (来自
Sphinx==1.7.dev20170506)
正在收集 sphinxcontrib-websupport (来自 Sphinx==1.7.dev20170506)
正在下载 sphinxcontrib_websupport-1.0.0-py2.py3-none-any.whl
正在收集 colorama>=0.3.5 (来自 Sphinx==1.7.dev20170506)
使用缓存的 colorama-0.3.9-py2.py3-none-any.whl
收集 MarkupSafe>=0.23 (来自 Jinja2>=2.3->Sphinx==1.7.dev20170506)
使用缓存的 MarkupSafe-1.0.tar.gz
收集 pytz>=0a (来自 babel!=2.0,>=1.3->Sphinx==1.7.dev20170506)
使用缓存的 pytz-2017.2-py2.py3-none-any.whl
收集 sqlalchemy>=0.9 (来自 sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
正在下载 SQLAlchemy-1.1.9.tar.gz (5.2MB)
100% |#####
正在下载 Whoosh-2.7.4-py2.py3-none-any.whl (468kB)
100% |#####
安装收集的包：six, MarkupSafe, Jinja2, Pygments, docutils, snowballstemmer, pytz,
babel, alabaster, imagesize, requests, typing, sqlalchemy, whoosh, sphinxcontrib-websupport,
colorama, Sphinx
正在运行 MarkupSafe 的 setup.py 安装... 完成
正在运行 typing 的 setup.py 安装... 完成
正在运行 sqlalchemy 的 setup.py 安装... 完成
正在运行 Sphinx 的 setup.py 安装... 完成
成功安装 Jinja2-2.9.6 MarkupSafe-1.0 Pygments-2.2.0 Sphinx-1.7.dev20170506
alabaster-0.7.10 babel-2.4.0 colorama-0.3.9 docutils-0.13.1 imagesize-0.7.1 pytz-2017.2
requests-2.13.0 six-1.10.0 snowballstemmer-1.2.1 sphinxcontrib-websupport-1.0.0 sqlalchemy-1.1.9
typing-3.6.1 whoosh-2.7.4

```

### 注意 URL 前的 git+ 前缀。

- 使用 git、mercurial 或其他可接受的工具（最好是分布式版本控制系统工具）克隆仓库，并使用 pip 安装 path/to/cloned/repo - 这将同时处理任何 requirements.txt 文件并执行构建和安装步骤，你也可以手动切换到克隆的仓库目录并运行 pip install -r requirements.txt 然后运行 python setup.py install 来达到同样效果。这种方法的最大优点是，虽然初次克隆操作可能比快照下载耗时更长，但你可以通过以下命令更新到最新版本，以 git 为例：git pull origin master，如果当前版本有错误，你可以使用 pip uninstall package-name 然后用 git checkout 命令回退到仓库历史中的早期版本并重新尝试。

```

Collecting typing (from Sphinx==1.7.dev20170506)
Using cached typing-3.6.1.tar.gz
Requirement already satisfied: setuptools in f:\toolbuild\temp\demo-pip\lib\site-packages (from
Sphinx==1.7.dev20170506)
Collecting sphinxcontrib-websupport (from Sphinx==1.7.dev20170506)
Downloading sphinxcontrib_websupport-1.0.0-py2.py3-none-any.whl
Collecting colorama>=0.3.5 (from Sphinx==1.7.dev20170506)
Using cached colorama-0.3.9-py2.py3-none-any.whl
Collecting MarkupSafe>=0.23 (from Jinja2>=2.3->Sphinx==1.7.dev20170506)
Using cached MarkupSafe-1.0.tar.gz
Collecting pytz>=0a (from babel!=2.0,>=1.3->Sphinx==1.7.dev20170506)
Using cached pytz-2017.2-py2.py3-none-any.whl
Collecting sqlalchemy>=0.9 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
Downloading SQLAlchemy-1.1.9.tar.gz (5.2MB)
100% |#####
Collecting whoosh>=2.0 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
Downloading Whoosh-2.7.4-py2.py3-none-any.whl (468kB)
100% |#####
Installing collected packages: six, MarkupSafe, Jinja2, Pygments, docutils, snowballstemmer, pytz,
babel, alabaster, imagesize, requests, typing, sqlalchemy, whoosh, sphinxcontrib-websupport,
colorama, Sphinx
Running setup.py install for MarkupSafe ... done
Running setup.py install for typing ... done
Running setup.py install for sqlalchemy ... done
Running setup.py install for Sphinx ... done
Successfully installed Jinja2-2.9.6 MarkupSafe-1.0 Pygments-2.2.0 Sphinx-1.7.dev20170506
alabaster-0.7.10 babel-2.4.0 colorama-0.3.9 docutils-0.13.1 imagesize-0.7.1 pytz-2017.2
requests-2.13.0 six-1.10.0 snowballstemmer-1.2.1 sphinxcontrib-websupport-1.0.0 sqlalchemy-1.1.9
typing-3.6.1 whoosh-2.7.4

```

### Note the git+ prefix to the URL.

- Clone the repository using git, mercurial or other acceptable tool, preferably a DVCS tool, and use pip install path/to/cloned/repo - this will both process any requirements.txt file and perform the build and setup steps, you can manually change directory to your cloned repository and run pip install -r requirements.txt and then python setup.py install to get the same effect. The big advantages of this approach is that while the initial clone operation may take longer than the snapshot download you can update to the latest with, in the case of git: git pull origin master and if the current version contains errors you can use pip uninstall package-name then use git checkout commands to move back through the repository history to earlier version(s) and re-try.

# 第83章：解析命令行参数

大多数命令行工具依赖于程序执行时传入的参数。这些程序不是通过提示输入，而是期望设置数据或特定标志（变为布尔值）。这允许用户和其他程序在启动时向 Python 文件传递数据。本节解释并演示了 Python 中命令行参数的实现和使用。

## 第83.1节：argparse中的Hello world

下面的程序向用户问好。它接受一个位置参数，即用户的名字，也可以指定问候语。

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('name',
                    help='用户名')
)

parser.add_argument('-g', '--greeting',
                    default='Hello',
                    help='可选的替代问候语'
)
args = parser.parse_args()

print("{greeting}, {name}!".format(
    greeting=args.greeting,
    name=args.name)
)

$ python hello.py --help
usage: hello.py [-h] [-g GREETING] name

positional arguments:
  name            用户的名字

optional arguments:
  -h, --help      显示此帮助信息并退出
  -g GREETING, --greeting GREETING
                  可选的替代问候语

$ python hello.py world
Hello, world!
$ python hello.py John -g Howdy
Howdy, John!
```

有关更多细节，请阅读[argparse文档](#)。

## 第83.2节：使用带有argv的命令行参数

每当从命令行调用Python脚本时，用户可以提供额外的命令行参数，这些参数将传递给脚本。程序员可以从系统变量[sys.argv](#)中获取这些参数（"argv"是大多数编程语言中使用的传统名称，意为"参数向量"）。

# Chapter 83: Parsing Command Line arguments

Most command line tools rely on arguments passed to the program upon its execution. Instead of prompting for input, these programs expect data or specific flags (which become booleans) to be set. This allows both the user and other programs to run the Python file passing it data as it starts. This section explains and demonstrates the implementation and usage of command line arguments in Python.

## Section 83.1: Hello world in argparse

The following program says hello to the user. It takes one positional argument, the name of the user, and can also be told the greeting.

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('name',
                    help='name of user')
)

parser.add_argument('-g', '--greeting',
                    default='Hello',
                    help='optional alternate greeting'
)
args = parser.parse_args()

print("{greeting}, {name}!".format(
    greeting=args.greeting,
    name=args.name)
)

$ python hello.py --help
usage: hello.py [-h] [-g GREETING] name

positional arguments:
  name            name of user

optional arguments:
  -h, --help      show this help message and exit
  -g GREETING, --greeting GREETING
                  optional alternate greeting

$ python hello.py world
Hello, world!
$ python hello.py John -g Howdy
Howdy, John!
```

For more details please read the [argparse documentation](#).

## Section 83.2: Using command line arguments with argv

Whenever a Python script is invoked from the command line, the user may supply additional **command line arguments** which will be passed on to the script. These arguments will be available to the programmer from the system variable [sys.argv](#) ("argv" is a traditional name used in most programming languages, and it means "argument vector").

按照惯例，`sys.argv`列表中的第一个元素是Python脚本本身的名字，而其余元素是用户调用脚本时传入的标记。

```
# cli.py
import sys
print(sys.argv)

$ python cli.py
=> ['cli.py']

$ python cli.py fizz
=> ['cli.py', 'fizz']

$ python cli.py fizz buzz
=> ['cli.py', 'fizz', 'buzz']
```

这是另一个使用`argv`的示例。我们首先去掉`sys.argv`的第一个元素，因为它包含脚本名称。然后将剩余的参数合并成一个句子，最后打印该句子，并在前面加上当前登录用户的名称（以模拟聊天程序）。

```
import getpass
import sys

words = sys.argv[1:]
sentence = " ".join(words)
print("[%s] %s" % (getpass.getuser(), sentence))
```

当“手动”解析多个非位置参数时常用的算法是遍历  
the `sys.argv` 列表。一种方法是遍历列表并弹出每个元素：

```
# 反转并复制 sys.argv
argv = reversed(sys.argv)
# 提取第一个元素
arg = argv.pop()
# 当没有更多参数可弹出时停止迭代
while len(argv) > 0:
    if arg in ('-f', '--foo'):
        print('检测到 foo!')
    elif arg in ('-b', '--bar'):
        print('见到 bar!')
    elif arg in ('-a', '--with-arg'):
        arg = arg.pop()
        print('见到值: {}'.format(arg))
    # 获取下一个值
    arg = argv.pop()
```

## 第83.3节：使用

`argparse` 设置互斥参数

如果你想让两个或更多参数互斥，可以使用函数  
`argparse.ArgumentParser.add_mutually_exclusive_group()`。在下面的示例中，`foo` 或 `bar` 可以存在，  
但不能同时存在。

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
```

By convention, the first element in the `sys.argv` list is the name of the Python script itself, while the rest of the elements are the tokens passed by the user when invoking the script.

```
# cli.py
import sys
print(sys.argv)

$ python cli.py
=> ['cli.py']

$ python cli.py fizz
=> ['cli.py', 'fizz']

$ python cli.py fizz buzz
=> ['cli.py', 'fizz', 'buzz']
```

Here's another example of how to use `argv`. We first strip off the initial element of `sys.argv` because it contains the script's name. Then we combine the rest of the arguments into a single sentence, and finally print that sentence prepending the name of the currently logged-in user (so that it emulates a chat program).

```
import getpass
import sys

words = sys.argv[1:]
sentence = " ".join(words)
print("[%s] %s" % (getpass.getuser(), sentence))
```

The algorithm commonly used when "manually" parsing a number of non-positional arguments is to iterate over the `sys.argv` list. One way is to go over the list and pop each element of it:

```
# reverse and copy sys.argv
argv = reversed(sys.argv)
# extract the first element
arg = argv.pop()
# stop iterating when there's no more args to pop()
while len(argv) > 0:
    if arg in ('-f', '--foo'):
        print('seen foo!')
    elif arg in ('-b', '--bar'):
        print('seen bar!')
    elif arg in ('-a', '--with-arg'):
        arg = arg.pop()
        print('seen value: {}'.format(arg))
    # get the next value
    arg = argv.pop()
```

## Section 83.3: Setting mutually exclusive arguments with argparse

If you want two or more arguments to be mutually exclusive. You can use the function  
`argparse.ArgumentParser.add_mutually_exclusive_group()`. In the example below, either `foo` or `bar` can exist  
but not both at the same time.

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
```

```
group.add_argument("-f", "--foo")
group.add_argument("-b", "--bar")
args = parser.parse_args()
print "foo = ", args.foo
print "bar = ", args.bar
```

如果您尝试运行脚本时同时指定--foo和--bar参数，脚本将会弹出以下消息。

错误：参数 -b/--bar：不允许与参数 -f/--foo 一起使用

## 第83.4节：使用 docopt 的基本示例

[docopt](#) 颠覆了命令行参数解析。你不需要解析参数，只需编写程序的用法字符串，[docopt](#) 解析用法字符串并用它来提取命令行参数。

```
"""
用法：
script_name.py [-a] [-b] <路径>
```

选项：

```
-a      打印所有内容。
-b      在路径中加入更多的蜜蜂。
```

从 docopt 导入 docopt

```
if __name__ == "__main__":
args = docopt(__doc__)
    import pprint; pprint.pprint(args)
```

示例运行：

```
$ python script_name.py
用法：
script_name.py [-a] [-b] <路径>
$ python script_name.py something
{'-a': False,
'-b': False,
'<路径>': 'something'}
$ python script_name.py something -a
{'-a': True,
'-b': False,
'<路径>': 'something'}
$ python script_name.py -b something -a
{'-a': True,
'-b': True,
'<路径>': 'something'}
```

## 第83.5节：使用argparse自定义解析器错误消息

您可以根据脚本需求创建解析器错误消息。这是通过 `argparse.ArgumentParser.error` 函数。下面的示例展示了当给出 `--foo` 但未给出 `--bar` 时，脚本向标准错误输出 (`stderr`) 打印用法和错误信息。

```
import argparse
```

```
group.add_argument("-f", "--foo")
group.add_argument("-b", "--bar")
args = parser.parse_args()
print "foo = ", args.foo
print "bar = ", args.bar
```

If you try to run the script specifying both --foo and --bar arguments, the script will complain with the below message.

error: argument -b/--bar: not allowed **with** argument -f/--foo

## Section 83.4: Basic example with docopt

[docopt](#) turns command-line argument parsing on its head. Instead of parsing the arguments, you just **write the usage string** for your program, and docopt **parses the usage string** and uses it to extract the command line arguments.

```
"""
Usage:
    script_name.py [-a] [-b] <path>

Options:
    -a      Print all the things.
    -b      Get more bees into the path.
"""
from docopt import docopt
```

```
if __name__ == "__main__":
args = docopt(__doc__)
    import pprint; pprint.pprint(args)
```

Sample runs:

```
$ python script_name.py
Usage:
    script_name.py [-a] [-b] <path>
$ python script_name.py something
{'-a': False,
'-b': False,
'<path>': 'something'}
$ python script_name.py something -a
{'-a': True,
'-b': False,
'<path>': 'something'}
$ python script_name.py -b something -a
{'-a': True,
'-b': True,
'<path>': 'something'}
```

## Section 83.5: Custom parser error message with argparse

You can create parser error messages according to your script needs. This is through the `argparse.ArgumentParser.error` function. The below example shows the script printing a usage and an error message to `stderr` when `--foo` is given but not `--bar`.

```
import argparse
```

```

parser = argparse.ArgumentParser()
parser.add_argument("-f", "--foo")
parser.add_argument("-b", "--bar")
args = parser.parse_args()
if args.foo and args.bar is None:
    parser.error("--foo requires --bar. You did not specify bar.")

print "foo =", args.foo
print "bar =", args.bar

```

假设你的脚本名为 sample.py，且我们运行：python sample.py --foo ds\_in\_fridge

脚本将报错如下：

```

usage: sample.py [-h] [-f FOO] [-b BAR]
sample.py: error: --foo requires --bar. You did not specify bar.

```

## 第83.6节：使用 argparse.add\_argument\_group()进行参数的概念分组

当你创建一个 argparse.ArgumentParser() 并用 '-h' 运行程序时，会自动显示一个用法信息，说明你可以用哪些参数运行你的软件。默认情况下，位置参数和条件参数被分为两类，例如，下面是一个小脚本 (example.py) 以及运行 python example.py -h 时的输出。

```

import argparse

parser = argparse.ArgumentParser(description='简单示例')
parser.add_argument('name', help='问候对象', default='World')
parser.add_argument('--bar_this')
parser.add_argument('--bar_that')
parser.add_argument('--foo_this')
parser.add_argument('--foo_that')
args = parser.parse_args()

用法 : example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                  [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                  名称

```

简单示例

位置参数：

名称 要问候的人

可选参数：

- h, --help 显示此帮助信息并退出
- bar\_this BAR\_THIS
- bar\_that BAR\_THAT
- foo\_this FOO\_THIS
- foo\_that FOO\_THAT

在某些情况下，您可能希望将参数进一步划分为不同的概念部分，以帮助用户。例如，您可能希望将所有输入选项放在一组中，所有输出格式选项放在另一组中。上述示例可以调整为将--foo\_\*参数与--bar\_\*参数分开，如下所示。

```

import argparse

parser = argparse.ArgumentParser(description='简单示例')

```

```

parser = argparse.ArgumentParser()
parser.add_argument("-f", "--foo")
parser.add_argument("-b", "--bar")
args = parser.parse_args()
if args.foo and args.bar is None:
    parser.error("--foo requires --bar. You did not specify bar.")

print "foo =", args.foo
print "bar =", args.bar

```

Assuming your script name is sample.py, and we run: python sample.py --foo ds\_in\_fridge

The script will complain with the following:

```

usage: sample.py [-h] [-f FOO] [-b BAR]
sample.py: error: --foo requires --bar. You did not specify bar.

```

## Section 83.6: Conceptual grouping of arguments with argparse.add\_argument\_group()

When you create an argparse.ArgumentParser() and run your program with '-h' you get an automated usage message explaining what arguments you can run your software with. By default, positional arguments and conditional arguments are separated into two categories, for example, here is a small script (example.py) and the output when you run python example.py -h.

```

import argparse

parser = argparse.ArgumentParser(description='Simple example')
parser.add_argument('name', help='Who to greet', default='World')
parser.add_argument('--bar_this')
parser.add_argument('--bar_that')
parser.add_argument('--foo_this')
parser.add_argument('--foo_that')
args = parser.parse_args()

usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                  [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                  name

```

Simple example

positional arguments:

name	Who to greet
------	--------------

optional arguments:

- h, --help show this **help** message **and** exit
- bar\_this BAR\_THIS
- bar\_that BAR\_THAT
- foo\_this FOO\_THIS
- foo\_that FOO\_THAT

There are some situations where you want to separate your arguments into further conceptual sections to assist your user. For example, you may wish to have all the input options in one group, and all the output formatting options in another. The above example can be adjusted to separate the --foo\_\* args from the --bar\_\* args like so.

```

import argparse

parser = argparse.ArgumentParser(description='Simple example')

```

```

parser.add_argument('name', help='问候对象', default='World')
# 创建两个参数组
foo_group = parser.add_argument_group(title='Foo 选项')
bar_group = parser.add_argument_group(title='Bar 选项')
# 向这些组添加参数
foo_group.add_argument('--bar_this')
foo_group.add_argument('--bar_that')
bar_group.add_argument('--foo_this')
bar_group.add_argument('--foo_that')
args = parser.parse_args()

```

当运行 `python example.py -h` 时，会产生以下输出：

```

用法：example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                  [--foo_this FOO_THIS] [--foo_that FOO_THAT]
名称

位置参数：
名称          要问候的人

可选参数：
-h, --help    显示此 help 信息并退出

Foo 选项：
--bar_this BAR_THIS
--bar_that BAR_THAT

Bar 选项：
--foo_this FOO_THIS
--foo_that FOO_THAT

```

## 第 83.7 节：使用 docopt 和 docopt\_dispatch 的高级示例

与 docopt 一样，使用 [docopt\_dispatch] 时，你需要在入口模块的 `_doc_` 变量中编写你的 `--help`。在那里，你调用 `dispatch` 并以文档字符串作为参数，这样它就可以对其进行解析器。

完成这一步后，不再手动处理参数（通常会导致高环路复杂度的 `if/else` 结构），而是将其交给 `dispatch`，只需说明你想如何处理这组参数。

这就是 `dispatch.on` 装饰器的作用：你给它应该触发函数的参数或参数序列，函数将以匹配的值作为参数执行。

""以开发模式或生产模式运行某些操作。

```

用法：run.py --development <host> <port>
      run.py --production <host> <port>
      run.py items add <item>
      run.py items delete <item>

      ...
from docopt_dispatch import dispatch

@dispatch.on('--development')
def development(host, port, **kwargs):
    print('处于*开发*模式')

```

```

parser.add_argument('name', help='Who to greet', default='World')
# Create two argument groups
foo_group = parser.add_argument_group(title='Foo options')
bar_group = parser.add_argument_group(title='Bar options')
# Add arguments to those groups
foo_group.add_argument('--bar_this')
foo_group.add_argument('--bar_that')
bar_group.add_argument('--foo_this')
bar_group.add_argument('--foo_that')
args = parser.parse_args()

```

Which produces this output when `python example.py -h` is run:

```

usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                  [--foo_this FOO_THIS] [--foo_that FOO_THAT]
name

Simple example

positional arguments:
  name                   Who to greet

optional arguments:
  -h, --help              show this help message and exit

Foo options:
  --bar_this BAR_THIS
  --bar_that BAR_THAT

Bar options:
  --foo_this FOO_THIS
  --foo_that FOO_THAT

```

## Section 83.7: Advanced example with docopt and docopt\_dispatch

As with docopt, with [docopt\_dispatch] you craft your `--help` in the `_doc_` variable of your entry-point module. There, you call `dispatch` with the doc string as argument, so it can run the parser over it.

That being done, instead of handling manually the arguments (which usually ends up in a high cyclomatic `if/else` structure), you leave it to `dispatch` giving only how you want to handle the set of arguments.

This is what the `dispatch.on` decorator is for: you give it the argument or sequence of arguments that should trigger the function, and that function will be executed with the matching values as parameters.

```

"""Run something in development or production mode.

Usage: run.py --development <host> <port>
      run.py --production <host> <port>
      run.py items add <item>
      run.py items delete <item>

      ...
from docopt_dispatch import dispatch

@dispatch.on('--development')
def development(host, port, **kwargs):
    print('in *development* mode')

```

```
@dispatch.on('--production')
def development(host, port, **kwargs):
    print('处于*生产*模式')

@dispatch.on('items', 'add')
def items_add(item, **kwargs):
    print('添加项目中...')

@dispatch.on('items', 'delete')
def items_delete(item, **kwargs):
    print('删除项目中...')

if __name__ == '__main__':
    dispatch(__doc__)
```

```
@dispatch.on('--production')
def development(host, port, **kwargs):
    print('in *production* mode')

@dispatch.on('items', 'add')
def items_add(item, **kwargs):
    print('adding item...')

@dispatch.on('items', 'delete')
def items_delete(item, **kwargs):
    print('deleting item...')

if __name__ == '__main__':
    dispatch(__doc__)
```

# 第84章：子进程库

## 参数

	详情
args	单个可执行文件，或可执行文件及参数序列 - 'ls', ['ls', '-la']
shell	是否在 shell 下运行？POSIX 系统上的默认 shell 是 /bin/sh。
cwd	子进程的工作目录。

## 第 84.1 节：使用 Popen 提供更多灵活性

使用 `subprocess.Popen` 比 `subprocess.call` 提供对启动进程的更细粒度控制。

### 启动子进程

```
process = subprocess.Popen([r'C:\path\app.exe', 'arg1', '--flag', 'arg'])
```

`Popen` 的签名与 `call` 函数非常相似；然而，`Popen` 会立即返回，而不像 `call` 那样等待子进程完成。

### 等待子进程完成

```
process = subprocess.Popen([r'C:\path\app.exe', 'arg1', '--flag', 'arg'])
process.wait()
```

### 从子进程读取输出

```
process = subprocess.Popen([r'C:\path\app.exe'], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
```

```
# 这将阻塞直到进程完成
stdout, stderr = process.communicate()
print stdout
print stderr
```

### 对正在运行的子进程进行交互式访问

即使子进程尚未完成，您也可以对 `stdin` 和 `stdout` 进行读写。这在自动化另一个程序的功能时可能很有用。

### 向子进程写入

```
process = subprocess.Popen([r'C:\path\app.exe'], stdout = subprocess.PIPE, stdin =
subprocess.PIPE)
```

```
process.stdin.write('line of input') # 写入输入
out.readline() # 从 stdout 读取一行# 对读取的行进行逻辑处理。
```

但是，如果你只需要一组输入和输出，而不是动态交互，应该使用 `communicate()` 而不是直接访问 `stdin` 和 `stdout`。

### 从子进程读取流

如果你想逐行查看子进程的输出，可以使用以下代码片段：

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.readline()
```

# Chapter 84: Subprocess Library

## Parameter

	Details
args	A single executable, or sequence of executable and arguments - 'ls', ['ls', '-la']
shell	Run under a shell? The default shell to /bin/sh on POSIX.
cwd	Working directory of the child process.

## Section 84.1: More flexibility with Popen

Using `subprocess.Popen` give more fine-grained control over launched processes than `subprocess.call`.

### Launching a subprocess

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

The signature for `Popen` is very similar to the `call` function; however, `Popen` will return immediately instead of waiting for the subprocess to complete like `call` does.

### Waiting on a subprocess to complete

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
process.wait()
```

### Reading output from a subprocess

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout=subprocess.PIPE, stderr=subprocess.PIPE)

# This will block until process completes
stdout, stderr = process.communicate()
print stdout
print stderr
```

### Interactive access to running subprocesses

You can read and write on `stdin` and `stdout` even while the subprocess hasn't completed. This could be useful when automating functionality in another program.

### Writing to a subprocess

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout = subprocess.PIPE, stdin =
subprocess.PIPE)
```

```
process.stdin.write('line of input\n') # Write input
line = process.stdout.readline() # Read a line from stdout
# Do logic on line read.
```

However, if you only need one set of input and output, rather than dynamic interaction, you should use `communicate()` rather than directly accessing `stdin` and `stdout`.

### Reading a stream from a subprocess

In case you want to see the output of a subprocess line by line, you can use the following snippet:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.readline()
```

如果子命令输出没有行结束符，上述代码片段将无法工作。你可以按字符读取输出，方法如下：

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.read(1)
```

传递给read方法的参数1指定read每次读取1个字符。你可以通过不同的数字指定读取任意数量的字符。负数或0告诉read将内容作为单个字符串读取，直到遇到文件结束符（EOF）（见此处）。

在上述两个代码片段中，`process.poll()` 在子进程结束之前一直是 `None`。这个方法用于在没有更多输出可读时退出循环。

同样的操作也可以应用于子进程的 `stderr`。

## 第84.2节：调用外部命令

最简单的用例是使用 `subprocess.call` 函数。它接受一个列表作为第一个参数。列表中的第一个元素应该是你想调用的外部应用程序。列表中的其他元素是将传递给该应用程序的参数。

```
subprocess.call([r'C:\patho\app.exe', 'arg1', '--flag', 'arg'])
```

对于 `shell` 命令，设置 `shell=True` 并以字符串形式提供命令，而不是列表。

```
subprocess.call('echo "Hello, world"', shell=True)
```

注意，上述两个命令只返回子进程的 `exit status`。此外，使用 `shell=True` 时要注意安全问题（参见 [here](#)）。

如果你想获取子进程的标准输出，可以用 `subprocess.check_output` 替代 `subprocess.call`。有关更高级的用法，请参考[此处](#)。

## 第84.3节：如何创建命令列表参数

允许运行命令的`subprocess`方法需要以列表形式提供命令（至少使用 `shell_mode=True`）。

创建该列表的规则并不总是容易遵循，尤其是对于复杂命令。

幸运的是，有一个非常有用的工具可以实现这一点：`shlex`。创建用作命令的列表的最简单方法如下：

```
import shlex
cmd_to_subprocess = shlex.split(command_used_in_the_shell)
```

一个简单的例子：

```
import shlex
shlex.split('ls --color -l -t -r')

out: ['ls', '--color', '-l', '-t', '-r']
```

in the case the subcommand output do not have EOL character, the above snippet does not work. You can then read the output character by character as follows:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.read(1)
```

The 1 specified as argument to the `read` method tells `read` to read 1 character at time. You can specify to read as many characters you want using a different number. Negative number or 0 tells to read to read as a single string until the EOF is encountered ([see here](#)).

In both the above snippets, the `process.poll()` is `None` until the subprocess finishes. This is used to exit the loop once there is no more output to read.

The same procedure could be applied to the `stderr` of the subprocess.

## Section 84.2: Calling External Commands

The simplest use case is using the `subprocess.call` function. It accepts a list as the first argument. The first item in the list should be the external application you want to call. The other items in the list are arguments that will be passed to that application.

```
subprocess.call([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg'])
```

For shell commands, set `shell=True` and provide the command as a string instead of a list.

```
subprocess.call('echo "Hello, world"', shell=True)
```

Note that the two command above return only the `exit status` of the subprocess. Moreover, pay attention when using `shell=True` since it provides security issues ([see here](#)).

If you want to be able to get the standard output of the subprocess, then substitute the `subprocess.call` with `subprocess.check_output`. For more advanced use, refer to this.

## Section 84.3: How to create the command list argument

The `subprocess` method that allows running commands needs the command in form of a list (at least using `shell_mode=True`).

The rules to create the list are not always straightforward to follow, especially with complex commands. Fortunately, there is a very helpful tool that allows doing that: `shlex`. The easiest way of creating the list to be used as command is the following:

```
import shlex
cmd_to_subprocess = shlex.split(command_used_in_the_shell)
```

A simple example:

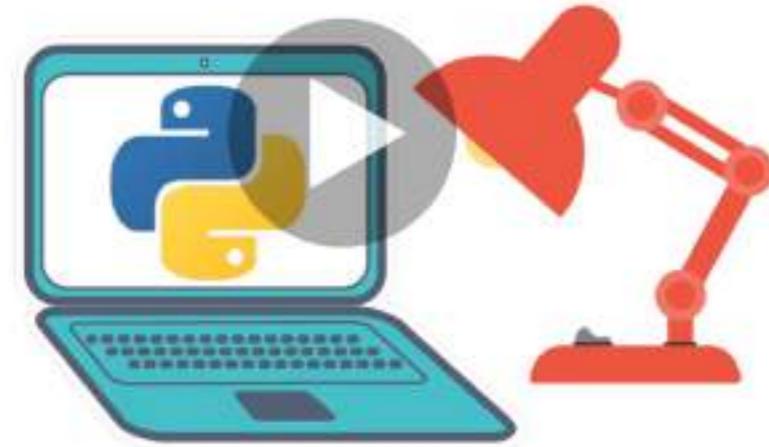
```
import shlex
shlex.split('ls --color -l -t -r')

out: ['ls', '--color', '-l', '-t', '-r']
```

# 视频：完整的Python 训练营：从零开始 成为Python 3高手

像专业人士一样学习Python！从基础开始，直到  
创建你自己的应用程序和游戏！

## COMPLETE PYTHON 3 BOOTCAMP



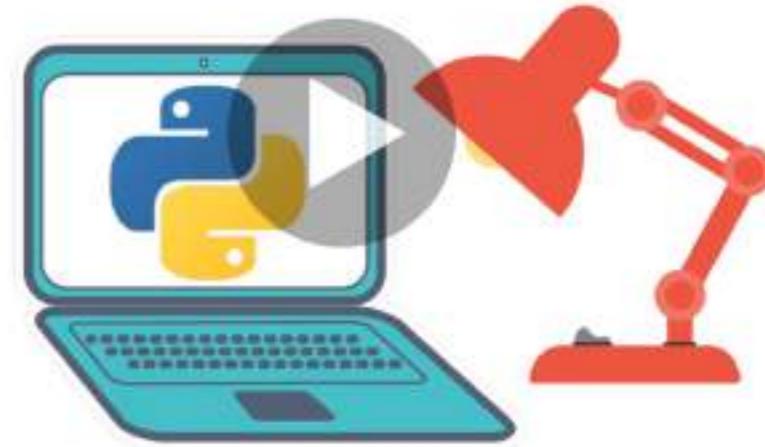
- ✓ 学习专业使用Python，掌握Python 2和Python 3！
- ✓ 用Python创建游戏，比如井字棋和二十一点！
- ✓ 学习高级Python功能，如collections模块和时间戳的使用！
- ✓ 学习使用面向对象编程和类！
- ✓ 理解复杂主题，如装饰器。
- ✓ 理解如何使用Jupyter Notebook并创建.py文件
- ✓ 了解如何在Jupyter Notebook系统中创建图形用户界面（GUI）！
- ✓ 从零开始全面理解Python！

今天观看 →

# VIDEO: Complete Python Bootcamp: Go from zero to hero in Python 3

Learn Python like a Professional! Start from the basics and go all the way to creating your own applications and games!

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ Learn to use Python professionally, learning both Python 2 and Python 3!
- ✓ Create games with Python, like Tic Tac Toe and Blackjack!
- ✓ Learn advanced Python features, like the collections module and how to work with timestamps!
- ✓ Learn to use Object Oriented Programming with classes!
- ✓ Understand complex topics, like decorators.
- ✓ Understand how to use both the Jupyter Notebook and create .py files
- ✓ Get an understanding of how to create GUIs in the Jupyter Notebook system!
- ✓ Build a complete understanding of Python from the ground up!

Watch Today →

# 第85章：setup.py

## 参数

	用法
姓名	你的发行版名称。
版本	你的发行版的版本字符串。
软件包	要包含的Python软件包列表（即包含模块的目录）。这可以手动指定，但通常使用调用setuptools.find_packages()来代替。
py_modules	要包含的顶级Python模块列表（即单个.py文件）。

## 第85.1节：setup.py的目的

setup脚本是使用Distutils构建、分发和安装模块的所有活动的核心。它的目的是正确安装软件。

如果你只想分发一个名为foo的模块，包含在文件foo.py中，那么你的setup脚本可以像这样简单：

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

要为此模块创建源代码分发包，您需要创建一个包含上述代码的安装脚本 setup.py，并在终端运行以下命令：

```
python setup.py sdist
```

sdist 将创建一个归档文件（例如，Unix 上的 tar 包，Windows 上的 ZIP 文件），其中包含您的安装脚本 setup.py 和模块 foo.py。归档文件将命名为 foo-1.0.tar.gz（或 .zip），解压后生成目录 foo-1.0。

如果最终用户希望安装您的 foo 模块，只需下载 foo-1.0.tar.gz（或 .zip），解压，并—从 foo-1.0 目录—运行

```
python setup.py install
```

## 第 85.2 节：在 setup.py 中使用源代码控制元数据

[setuptools\\_scm](#) 是一个官方认可的包，可以使用 Git 或 Mercurial 的元数据来确定包的版本号，并查找要包含的 Python 包及包数据。

```
from setuptools import setup, find_packages

setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
)
```

此示例同时使用了这两种功能；如果只想使用 SCM 元数据来确定版本，请将对 find\_packages() 的调用替换为手动的包列表，或者如果只想使用包查找器，则移除 use\_scm\_version=True。

# Chapter 85: setup.py

## Parameter

	Usage
name	Name of your distribution.
version	Version string of your distribution.
packages	List of Python packages (that is, directories containing modules) to include. This can be specified manually, but a call to <code>setuptools.find_packages()</code> is typically used instead.
py_modules	List of top-level Python modules (that is, single .py files) to include.

## Section 85.1: Purpose of setup.py

The setup script is the center of all activity in building, distributing, and installing modules using the Distutils. Its purpose is the correct installation of the software.

If all you want to do is distribute a module called foo, contained in a file foo.py, then your setup script can be as simple as this:

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

To create a source distribution for this module, you would create a setup script, setup.py, containing the above code, and run this command from a terminal:

```
python setup.py sdist
```

sdist will create an archive file (e.g., tarball on Unix, ZIP file on Windows) containing your setup script setup.py, and your module foo.py. The archive file will be named foo-1.0.tar.gz (or .zip), and will unpack into a directory foo-1.0.

If an end-user wishes to install your foo module, all she has to do is download foo-1.0.tar.gz (or .zip), unpack it, and—from the foo-1.0 directory—run

```
python setup.py install
```

## Section 85.2: Using source control metadata in setup.py

[setuptools\\_scm](#) is an officially-blessed package that can use Git or Mercurial metadata to determine the version number of your package, and find Python packages and package data to include in it.

```
from setuptools import setup, find_packages

setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
)
```

This example uses both features; to only use SCM metadata for the version, replace the call to `find_packages()` with your manual package list, or to only use the package finder, remove `use_scm_version=True`.

## 第85.3节：向你的 Python 包中添加命令行脚本

包中添加命令行脚本

Python 包中的命令行脚本很常见。你可以组织你的包，使得当用户安装该包时，脚本会自动添加到他们的路径中。

假设你有一个名为 `greetings` 的包，其中包含命令行脚本 `hello_world.py`。

```
greetings/  
greetings/  
    __init__.py  
    hello_world.py
```

你可以通过运行以下命令来执行该脚本：

```
python greetings/greetings/hello_world.py
```

但是如果你想这样运行它：

```
hello_world.py
```

你可以通过在 `setup.py` 中的 `setup()` 里添加脚本（scripts）来实现，方法如下：

```
from setuptools import setup  
setup(  
    name='greetings',  
    scripts=['hello_world.py'])
```

当你现在安装 `greetings` 包时，`hello_world.py` 将被添加到你的路径中。

另一种可能是添加一个入口点：

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

这样你只需像这样运行它：

```
greetings
```

## 第85.4节：添加安装选项

如前面示例所示，该脚本的基本用法是：

```
python setup.py install
```

但还有更多选项，比如安装该软件包后，可以更改代码并进行测试，而无需重新安装。实现方法是使用：

```
python setup.py develop
```

如果你想执行特定操作，比如编译 `Sphinx` 文档或构建 `fortran` 代码，你可以像这样创建你自己的选项：

```
cmdclasses = dict()
```

## Section 85.3: Adding command line scripts to your python package

Command line scripts inside python packages are common. You can organise your package in such a way that when a user installs the package, the script will be available on their path.

If you had the `greetings` package which had the command line script `hello_world.py`.

```
greetings/  
greetings/  
    __init__.py  
    hello_world.py
```

You could run that script by running:

```
python greetings/greetings/hello_world.py
```

However if you would like to run it like so:

```
hello_world.py
```

You can achieve this by adding scripts to your `setup()` in `setup.py` like this:

```
from setuptools import setup  
setup(  
    name='greetings',  
    scripts=['hello_world.py'])
```

When you install the `greetings` package now, `hello_world.py` will be added to your path.

Another possibility would be to add an entry point:

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

This way you just have to run it like:

```
greetings
```

## Section 85.4: Adding installation options

As seen in previous examples, basic use of this script is:

```
python setup.py install
```

But there is even more options, like installing the package and have the possibility to change the code and test it without having to re-install it. This is done using:

```
python setup.py develop
```

If you want to perform specific actions like compiling a `Sphinx` documentation or building `fortran` code, you can create your own option like this:

```
cmdclasses = dict()
```

```
class BuildSphinx(Command):
    """构建 Sphinx 文档。"""

    description = '构建 Sphinx 文档'
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):
        pass

    def run(self):
        import sphinx
        sphinx.build_main(['setup.py', '-b', 'html', './doc', './doc/_build/html'])
        sphinx.build_main(['setup.py', '-b', 'man', './doc', './doc/_build/man'])

cmdclasses['build_sphinx'] = BuildSphinx

setup(
...
cmdclass=cmdclasses,
)
```

initialize\_options 和 finalize\_options 将分别在 run 函数执行前后运行，正如它们的名字所示。

之后，你将能够调用你的选项：

```
python setup.py build_sphinx
```

```
class BuildSphinx(Command):
    """Build Sphinx documentation."""

    description = 'Build Sphinx documentation'
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):
        pass

    def run(self):
        import sphinx
        sphinx.build_main(['setup.py', '-b', 'html', './doc', './doc/_build/html'])
        sphinx.build_main(['setup.py', '-b', 'man', './doc', './doc/_build/man'])

cmdclasses['build_sphinx'] = BuildSphinx

setup(
...
cmdclass=cmdclasses,
)
```

initialize\_options 和 finalize\_options 将会在 run 函数执行前后运行，正如它们的名字所示。

之后，你将能够调用你的选项：

```
python setup.py build_sphinx
```

# 第86章：递归

## 第86.1节：递归的是什么、如何以及何时

递归发生在一个函数调用导致该函数在原始函数调用终止之前再次被调用。例如，考虑著名的数学表达式  $x!$  (即阶乘运算)。阶乘运算对所有非负整数定义如下：

- 如果数字是0，那么答案是1。
- 否则，答案是该数字乘以比该数字小一的数字的阶乘。

在Python中，阶乘操作的一个简单实现可以定义为如下函数：

```
def factorial(n):
    如果 n == 0:
        返回 1
    否则:
        return n * factorial(n - 1)
```

递归函数有时可能难以理解，所以让我们一步步来分析。考虑表达式`factorial(3)`。这个以及所有函数调用都会创建一个新的环境。环境基本上就是一个将标识符（例如 `n`、`factorial`、`print` 等）映射到对应值的表。在任何时刻，你都可以使用`locals()`访问当前环境。在第一次函数调用中，唯一定义的局部变量是 `n = 3`。因此，打印`locals()`会显示`{'n': 3}`。由于 `n == 3`，返回值变成了 `n * factorial(n - 1)`。

接下来的这一步可能会有点让人困惑。看我们的新表达式，我们已经知道了 `n` 的值。然而，我们还不知道 `factorial(n - 1)` 的值。首先，`n - 1` 计算结果为 2。然后，2 作为 `n` 的值传递给 `factorial`。由于这是一个新的函数调用，会创建第二个环境来存储这个新的 `n`。设 A 为第一个环境，B 为第二个环境。环境 A 仍然存在且等于`{'n': 3}`，但当前环境是 B（等于`{'n': 2}`）。看函数体，返回值依然是 `n * factorial(n - 1)`。先不计算这个表达式，我们将其代入原始的返回表达式。这样做时，我们在脑中丢弃了 B，所以记得相应地替换 `n`（即 B 中的 `n` 引用被替换为 `n - 1`，使用的是 A 中的 `n`）。现在，原始返回表达式变成了 `n * ((n - 1) * factorial((n - 1) - 1))`。花点时间确保你理解为什么会这样。

现在，让我们计算 `factorial((n - 1) - 1)` 部分。由于 A 中的 `n == 3`，我们传入了 1 给 `factorial`。

因此，我们创建了一个新的环境 C，等于`{'n': 1}`。同样，返回值是 `n * factorial(n - 1)`。现在让我们像之前调整原始返回表达式那样，替换 `factorial((n - 1) - 1)` 在“原始”返回表达式中的部分。“原始”表达式现在是 `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))`。

快完成了。现在，我们需要计算 `factorial((n - 2) - 1)`。这次传入的是 0。因此，这个值为 1。现在，让我们进行最后一次替换。“原始”返回表达式现在是 `n * ((n - 1) * ((n - 2) * 1))`。回想一下，原始返回表达式是在 A 下计算的，表达式变为 `3 * ((3 - 1) * ((3 - 2) * 1))`。当然，这个值为 6。为了确认这是正确答案，回想  $3! = 3 \cdot 2 \cdot 1 = 6$ 。在继续阅读之前，请确保你完全理解环境的概念以及它们如何应用于递归。

语句 `if n == 0: return 1` 被称为基例（base case）。这是因为它不包含递归。基例是绝对必要的。没有基例，你将陷入无限递归。话虽如此，只要你至少有一个基例，你可以有任意多个情况。例如，我们也可以等价地将 `factorial` 写成

# Chapter 86: Recursion

## Section 86.1: The What, How, and When of Recursion

Recursion occurs when a function call causes that same function to be called again before the original function call terminates. For example, consider the well-known mathematical expression  $x!$  (i.e. the factorial operation). The factorial operation is defined for all nonnegative integers as follows:

- If the number is 0, then the answer is 1.
- Otherwise, the answer is that number times the factorial of one less than that number.

In Python, a naïve implementation of the factorial operation can be defined as a function as follows:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Recursion functions can be difficult to grasp sometimes, so let's walk through this step-by-step. Consider the expression `factorial(3)`. This and *all* function calls create a new **environment**. An environment is basically just a table that maps identifiers (e.g. `n`, `factorial`, `print`, etc.) to their corresponding values. At any point in time, you can access the current environment using `locals()`. In the first function call, the only local variable that gets defined is `n = 3`. Therefore, printing `locals()` would show `{'n': 3}`. Since `n == 3`, the return value becomes `n * factorial(n - 1)`.

At this next step is where things might get a little confusing. Looking at our new expression, we already know what `n` is. However, we don't yet know what `factorial(n - 1)` is. First, `n - 1` evaluates to 2. Then, 2 is passed to `factorial` as the value for `n`. Since this is a new function call, a second environment is created to store this new `n`. Let A be the first environment and B be the second environment. A still exists and equals `{'n': 3}`, however, B (which equals `{'n': 2}`) is the current environment. Looking at the function body, the return value is, again, `n * factorial(n - 1)`. Without evaluating this expression, let's substitute it into the original return expression. By doing this, we're mentally discarding B, so remember to substitute `n` accordingly (i.e. references to B's `n` are replaced with `n - 1` which uses A's `n`). Now, the original return expression becomes `n * ((n - 1) * factorial((n - 1) - 1))`. Take a second to ensure that you understand why this is so.

Now, let's evaluate the `factorial((n - 1) - 1)` portion of that. Since A's `n == 3`, we're passing 1 into `factorial`. Therefore, we are creating a new environment C which equals `{'n': 1}`. Again, the return value is `n * factorial(n - 1)`. So let's replace `factorial((n - 1) - 1)` of the “original” return expression similarly to how we adjusted the original return expression earlier. The “original” expression is now `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))`.

Almost done. Now, we need to evaluate `factorial((n - 2) - 1)`. This time, we're passing in 0. Therefore, this evaluates to 1. Now, let's perform our last substitution. The “original” return expression is now `n * ((n - 1) * ((n - 2) * 1))`. Recalling that the original return expression is evaluated under A, the expression becomes `3 * ((3 - 1) * ((3 - 2) * 1))`. This, of course, evaluates to 6. To confirm that this is the correct answer, recall that  $3! = 3 \cdot 2 \cdot 1 = 6$ . Before reading any further, be sure that you fully understand the concept of environments and how they apply to recursion.

The statement `if n == 0: return 1` is called a base case. This is because, it exhibits no recursion. A base case is absolutely required. Without one, you'll run into infinite recursion. With that said, as long as you have at least one base case, you can have as many cases as you want. For example, we could have equivalently written `factorial` as

如下：

```
def factorial(n):
    如果 n == 0:
        return 1
    elif n == 1:
        返回 1
    否则:
        return n * factorial(n - 1)
```

你也可以有多个递归情况，但我们不会深入讨论，因为这相对少见且通常难以在脑海中处理。

你也可以有“并行”递归函数调用。例如，考虑斐波那契数列，其定义如下：

- 如果数字是0，那么答案是0。
- 如果数字是1，那么答案是1。
- 否则，答案是前两个斐波那契数的和。

我们可以这样定义它：

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

我不会像讲解factorial(3)那样详细讲解这个函数，但fib(5)的最终返回值等同于以下（语法上无效的）表达式：

```
(  
fib((n - 2) - 2)  
+  
(  
fib(((n - 2) - 1) - 2)  
+  
fib(((n - 2) - 1) - 1)  
)  
+  
(  
    fib((n - 1) - 2)  
+  
    (  
        fib(((n - 1) - 1) - 2)  
+  
        fib(((n - 1) - 1) - 1) - 1  
)  
)  
)
```

follows:

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

You may also have multiple recursion cases, but we won't get into that since it's relatively uncommon and is often difficult to mentally process.

You can also have “parallel” recursive function calls. For example, consider the [Fibonacci sequence](#) which is defined as follows:

- If the number is 0, then the answer is 0.
- If the number is 1, then the answer is 1.
- Otherwise, the answer is the sum of the previous two Fibonacci numbers.

We can define this as follows:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

I won't walk through this function as thoroughly as I did with factorial(3), but the final return value of fib(5) is equivalent to the following (*syntactically invalid*) expression:

```
(  
fib((n - 2) - 2)  
+  
(  
    fib(((n - 2) - 1) - 2)  
+  
    fib(((n - 2) - 1) - 1)  
)  
+  
(  
    fib(((n - 1) - 2) - 2)  
+  
    fib(((n - 1) - 2) - 1)  
)  
+  
(  
    fib(((n - 1) - 1) - 2)  
+  
    (  
        fib((((n - 1) - 1) - 1) - 2)  
+  
        fib((((n - 1) - 1) - 1) - 1)  
)  
)  
)
```

这变成了 $(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))$ , 当然计算结果是5。

现在, 让我们介绍几个更多的词汇:

- 尾调用 (tail call) 就是递归函数调用中, 在返回值之前执行的最后一个操作。明确来说, `return foo(n - 1)` 是尾调用, 但`return foo(n - 1) + 1` 不是 (因为加法是最后一个操作)。
- 尾调用优化 (Tail call optimization, TCO)** 是一种自动减少递归函数中递归调用的方法。
- 尾调用消除 (Tail call elimination, TCE) 是将尾调用简化为可以在不使用递归的情况下计算的表达式。TCO 是 TCE 的一种类型。

尾调用优化有多方面的好处:

- 解释器可以最小化环境占用的内存。由于没有计算机拥有无限内存, 过多的递归函数调用会导致栈溢出。
- 解释器可以减少栈帧切换的次数。

Python 没有实现尾调用优化 (TCO), 原因有很多。因此, 需要其他技术来绕过这个限制。选择的方法取决于具体的使用场景。凭借一些直觉, 阶乘 (factorial) 和斐波那契数列 (fib) 的定义可以相对容易地转换为如下的迭代代码:

阶乘 (factorial) 和斐波那契数列 (fib) 可以相对容易地转换为如下的迭代代码:

```
def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product

def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

这通常是手动消除递归的最高效方法, 但对于更复杂的函数来说, 可能会变得相当困难。

另一个有用的工具是 Python 的 `lru_cache` 装饰器, 它可以用来减少冗余计算的次数。

你现在已经了解了如何避免在Python中使用递归, 但你应该什么时候使用递归呢? 答案是“不常用”。所有递归函数都可以用迭代方式实现。这只是一个如何实现的问题。然而, 在某些罕见情况下, 递归是可以接受的。当预期输入不会导致递归函数调用次数显著增加时, 递归在Python中较为常见。

如果你对递归感兴趣, 我强烈建议你学习函数式语言, 如Scheme或Haskell。在这些语言中, 递归更为有用。

请注意, 上述斐波那契数列的例子虽然很好地展示了如何在Python中应用定义并随后使用lru缓存, 但其运行效率较低, 因为每个非基例都会进行两次递归调用。函数调用次数呈指数级增长, 达到  $n$ 。

相当不直观的是, 更高效的实现方式是使用线性递归:

```
def fib(n):
    if n <= 1:
```

This becomes  $(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))$  which of course evaluates to 5.

Now, let's cover a few more vocabulary terms:

- A **tail call** is simply a recursive function call which is the last operation to be performed before returning a value. To be clear, `return foo(n - 1)` is a tail call, but `return foo(n - 1) + 1` is not (since the addition is the last operation).
- Tail call optimization (TCO)** is a way to automatically reduce recursion in recursive functions.
- Tail call elimination (TCE)** is the reduction of a tail call to an expression that can be evaluated without recursion. TCE is a type of TCO.

Tail call optimization is helpful for a number of reasons:

- The interpreter can minimize the amount of memory occupied by environments. Since no computer has unlimited memory, excessive recursive function calls would lead to a [stack overflow](#).
- The interpreter can reduce the number of [stack frame](#) switches.

Python has no form of TCO implemented for [a number of reasons](#). Therefore, other techniques are required to skirt this limitation. The method of choice depends on the use case. With some intuition, the definitions of factorial and fib can relatively easily be converted to iterative code as follows:

```
def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product

def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

This is usually the most efficient way to manually eliminate recursion, but it can become rather difficult for more complex functions.

Another useful tool is Python's `lru_cache` decorator which can be used to reduce the number of redundant calculations.

You now have an idea as to how to avoid recursion in Python, but when *should* you use recursion? The answer is “not often”. All recursive functions can be implemented iteratively. It's simply a matter of figuring out how to do so. However, there are rare cases in which recursion is okay. Recursion is common in Python when the expected inputs wouldn't cause a significant number of recursive function calls.

If recursion is a topic that interests you, I implore you to study functional languages such as Scheme or Haskell. In such languages, recursion is much more useful.

Please note that the above example for the Fibonacci sequence, although good at showing how to apply the definition in python and later use of the lru cache, has an inefficient running time since it makes 2 recursive calls for each non base case. The number of calls to the function grows exponentially to  $n$ .

Rather non-intuitively a more efficient implementation would use linear recursion:

```
def fib(n):
    if n <= 1:
```

```

    return (n, 0)
else:
    (a, b) = fib(n - 1)
    return (a + b, a)

```

但是那个函数有返回一对数字的问题。这强调了有些函数确实从递归中获益不大。

## 第86.2节：使用递归进行树的遍历

假设我们有如下树结构：

```

根节点
- A
- AA
- - AB
- B
- BA
- - BB
- - BBA

```

现在，如果我们想列出所有元素的名称，可以用一个简单的for循环来实现。我们假设有一个函数get\_name()用于返回节点名称的字符串，一个函数get\_children()用于返回给定节点的所有子节点列表，还有一个函数get\_root()用于获取根节点。

```

root = get_root(tree)
for node in get_children(root):
    print(get_name(node))
    for child in get_children(node):
        print(get_name(child))
        for grand_child in get_children(child):
            print(get_name(grand_child))
# 打印: A, AA, AB, B, BA, BB, BBA

```

这种方法运行良好且快速，但如果子节点本身还有子节点呢？而那些子节点可能还有更多子节点……如果你事先不知道会有多少个该怎么办？解决这个问题的方法是使用递归。

```

def list_tree_names(node):
    for child in get_children(node):
        print(get_name(child))
        list_tree_names(node=child)

list_tree_names(node=get_root(tree))
# 打印: A, AA, AB, B, BA, BB, BBA

```

也许你不想打印，而是想返回所有节点名称的扁平列表。这可以通过传递一个滚动列表作为参数来实现。

```

def list_tree_names(node, lst=[]):
    for child in get_children(node):
        lst.append(get_name(child))
        list_tree_names(node=child, lst=lst)
    return lst

list_tree_names(node=get_root(tree))
# 返回 [A, 'AA', 'AB', 'B', 'BA', 'BB', 'BBA']

```

```

    return (n, 0)
else:
    (a, b) = fib(n - 1)
    return (a + b, a)

```

But that one has the issue of returning a *pair* of numbers. This emphasizes that some functions really do not gain much from recursion.

## Section 86.2: Tree exploration with recursion

Say we have the following tree:

```

root
- A
- AA
- - AB
- B
- BA
- - BB
- - BBA

```

Now, if we wish to list all the names of the elements, we could do this with a simple for-loop. We assume there is a function get\_name() to return a string of the name of a node, a function get\_children() to return a list of all the sub-nodes of a given node in the tree, and a function get\_root() to get the root node.

```

root = get_root(tree)
for node in get_children(root):
    print(get_name(node))
    for child in get_children(node):
        print(get_name(child))
        for grand_child in get_children(child):
            print(get_name(grand_child))
# prints: A, AA, AB, B, BA, BB, BBA

```

This works well and fast, but what if the sub-nodes, got sub-nodes of its own? And those sub-nodes might have more sub-nodes... What if you don't know beforehand how many there will be? A method to solve this is the use of recursion.

```

def list_tree_names(node):
    for child in get_children(node):
        print(get_name(child))
        list_tree_names(node=child)

list_tree_names(node=get_root(tree))
# prints: A, AA, AB, B, BA, BB, BBA

```

Perhaps you wish to not print, but return a flat list of all node names. This can be done by passing a rolling list as a parameter.

```

def list_tree_names(node, lst=[]):
    for child in get_children(node):
        lst.append(get_name(child))
        list_tree_names(node=child, lst=lst)
    return lst

list_tree_names(node=get_root(tree))
# returns [A, 'AA', 'AB', 'B', 'BA', 'BB', 'BBA']

```

## 第86.3节：从1到n的数字之和

如果我想计算从1到n的数字之和，其中 n是一个自然数，我可以做 $1 + 2 + 3 + 4 + \dots +$  (几个小时后) + n。或者，我可以写一个for循环：

```
n = 0
for i in range (1, n+1):
    n += i
```

或者我可以使用一种称为递归的技术：

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

递归相比上述两种方法有优势。递归比写出  $1 + 2 + 3$  来计算从 1 到 3 的和所需时间更少。对于 `recursion(4)`，可以使用递归向后计算：

函数调用： $(4 \rightarrow 4 + 3 \rightarrow 4 + 3 + 2 \rightarrow 4 + 3 + 2 + 1 \rightarrow$

$> 10)$  而 for 循环则严格向前计算： $(1 \rightarrow 1 + 2 \rightarrow 1 + 2 + 3 \rightarrow 1 + 2 + 3 + 4 \rightarrow 10)$ 。有时递归解法比迭代解法更简单。这在实现链表反转时尤为明显。

## 第86.4节：增加最大递归深度

递归的最大深度是有限制的，这取决于Python的实现。当达到限制时，会引发`RuntimeError`异常：

`RuntimeError: 超出最大递归深度`

以下是一个可能导致此错误的程序示例：

```
def cursing(depth):
    try:
        cursing(depth + 1) # 实际上，是再次递归
    except RuntimeError as RE:
        print('我递归了 {} 次 !'.format(depth))
cursing(0)
# Out: 我递归了1083次！
```

可以通过使用以下方法更改递归深度限制

```
sys.setrecursionlimit(limit)
```

您可以通过运行以下命令来检查当前的递归限制参数：

```
sys.getrecursionlimit()
```

使用我们新的限制运行上述相同方法，得到

```
sys.setrecursionlimit(2000)
cursing(0)
# 输出：我递归了1997次！
```

## Section 86.3: Sum of numbers from 1 to n

If I wanted to find out the sum of numbers from 1 to n where n is a natural number, I can do `1 + 2 + 3 + 4 + ... +` (several hours later) + n. Alternatively, I could write a `for` loop:

```
n = 0
for i in range (1, n+1):
    n += i
```

Or I could use a technique known as recursion:

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

Recursion has advantages over the above two methods. Recursion takes less time than writing out `1 + 2 + 3` for a sum from 1 to 3. For `recursion(4)`, recursion can be used to work backwards:

Function calls: ( $4 \rightarrow 4 + 3 \rightarrow 4 + 3 + 2 \rightarrow 4 + 3 + 2 + 1 \rightarrow 10$ )

Whereas the `for` loop is working strictly forwards: ( $1 \rightarrow 1 + 2 \rightarrow 1 + 2 + 3 \rightarrow 1 + 2 + 3 + 4 \rightarrow 10$ ). Sometimes the recursive solution is simpler than the iterative solution. This is evident when implementing a reversal of a linked list.

## Section 86.4: Increasing the Maximum Recursion Depth

There is a limit to the depth of possible recursion, which depends on the Python implementation. When the limit is reached, a `RuntimeError` exception is raised:

`RuntimeError: Maximum Recursion Depth Exceeded`

Here's a sample of a program that would cause this error:

```
def cursing(depth):
    try:
        cursing(depth + 1) # actually, re-cursing
    except RuntimeError as RE:
        print('I recursed {} times!'.format(depth))
cursing(0)
# Out: I recursed 1083 times!
```

It is possible to change the recursion depth limit by using

```
sys.setrecursionlimit(limit)
```

You can check what the current parameters of the limit are by running:

```
sys.getrecursionlimit()
```

Running the same method above with our new limit we get

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

从Python 3.5开始，异常类型为RecursionError，继承自RuntimeError。

## 第86.5节：尾递归——不良实践

当函数唯一返回的内容是递归调用时，称为尾递归。

以下是使用尾递归编写的倒计时示例：

```
def countdown(n):
    如果 n == 0:
        打印 "Blastoff!"
    else:
        打印 n
countdown(n-1)
```

任何可以通过迭代完成的计算也可以通过递归完成。下面是一个使用尾递归编写的 find\_max 版本：

```
def find_max(seq, max_so_far):
    if not seq:
        return max_so_far
    if max_so_far < seq[0]:
        return find_max(seq[1:], seq[0])
    else:
        return find_max(seq[1:], max_so_far)
```

在 Python 中，尾递归被认为是不好的编程习惯，因为 Python 编译器不对尾递归调用进行优化。像这种情况的递归解决方案比等效的迭代解决方案使用更多的系统资源。

## 第86.6节：通过栈内省进行尾递归优化

默认情况下，Python 的递归栈不能超过1000帧。这个限制可以通过设置 sys.setrecursionlimit(15000) 这种方法更快，但会消耗更多内存。相反，我们也可以通过栈内省来解决尾递归问题。

```
#!/usr/bin/env python2.4
# 该程序展示了一个实现尾调用优化的Python装饰器。它通过抛出异常（当它是自己的祖父时）并捕获此类异常来重新调用栈，从而实现尾调用优化。

import sys

class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs

def tail_call_optimized(g):
"""

该函数装饰一个函数以实现尾调用优化。它通过抛出异常（当它是自己的祖父时）并捕获此类异常来模拟尾调用优化。

```

该函数在装饰的函数失败时

From Python 3.5, the exception is a RecursionError, which is derived from RuntimeError.

## Section 86.5: Tail Recursion - Bad Practice

When the only thing returned from a function is a recursive call, it is referred to as tail recursion.

Here's an example countdown written using tail recursion:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

Any computation that can be made using iteration can also be made using recursion. Here is a version of find\_max written using tail recursion:

```
def find_max(seq, max_so_far):
    if not seq:
        return max_so_far
    if max_so_far < seq[0]:
        return find_max(seq[1:], seq[0])
    else:
        return find_max(seq[1:], max_so_far)
```

Tail recursion is considered a bad practice in Python, since the Python compiler does not handle optimization for tail recursive calls. The recursive solution in cases like this use more system resources than the equivalent iterative solution.

## Section 86.6: Tail Recursion Optimization Through Stack Introspection

By default Python's recursion stack cannot exceed 1000 frames. This can be changed by setting the sys.setrecursionlimit(15000) which is faster however, this method consumes more memory. Instead, we can also solve the Tail Recursion problem using stack introspection.

```
#!/usr/bin/env python2.4
# This program shows off a python decorator which implements tail call optimization. It
# does this by throwing an exception if it is its own grandparent, and catching such
# exceptions to recall the stack.

import sys

class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs

    def tail_call_optimized(g):
"""

This function decorates a function with tail call optimization. It does this by throwing an exception
if it is its own grandparent, and catching such
exceptions to fake the tail call optimization.

This function fails if the decorated

```

函数在非尾调用上下文中递归。

```
def func(*args, **kwargs):
    f = sys._getframe()
    if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
        raise TailRecursionException(args, kwargs)
    else:
        while 1:
            try:
                return g(*args, **kwargs)
            except TailRecursionException, e:
                args = e.args
                kwargs = e.kwargs
                func.__doc__ = g.__doc__
    return func
```

为了优化递归函数，我们可以使用@tail\_call\_optimized装饰器来调用我们的函数。以下是使用上述装饰器的一些常见递归示例：

阶乘示例：

```
@tail_call_optimized
def 阶乘(n, acc=1):
    "计算阶乘"
    if n == 0:
        return acc
    return 阶乘(n-1, n*acc)

print 阶乘(10000)
# 打印一个非常大的数字,
# 但不会达到递归限制。
```

斐波那契示例：

```
@tail_call_optimized
def 斐波那契(i, 当前 = 0, 下一个 = 1):
    if i == 0:
        return 当前
    else:
        return fib(i - 1, next, current + next)

print fib(10000)
# 也会打印一个大数字,
# 但不会达到递归限制。
```

function recurses in a non-tail context.

```
def func(*args, **kwargs):
    f = sys._getframe()
    if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
        raise TailRecursionException(args, kwargs)
    else:
        while 1:
            try:
                return g(*args, **kwargs)
            except TailRecursionException, e:
                args = e.args
                kwargs = e.kwargs
                func.__doc__ = g.__doc__
    return func
```

To optimize the recursive functions, we can use the @tail\_call\_optimized decorator to call our function. Here's a few of the common recursion examples using the decorator described above:

Factorial Example:

```
@tail_call_optimized
def factorial(n, acc=1):
    "calculate a factorial"
    if n == 0:
        return acc
    return factorial(n-1, n*acc)

print factorial(10000)
# prints a big, big number,
# but doesn't hit the recursion limit.
```

Fibonacci Example:

```
@tail_call_optimized
def fib(i, current = 0, next = 1):
    if i == 0:
        return current
    else:
        return fib(i - 1, next, current + next)

print fib(10000)
# also prints a big number,
# but doesn't hit the recursion limit.
```

# 第87章：类型提示

## 第87.1节：为函数添加类型

让我们举一个函数的例子，该函数接收两个参数并返回它们的和：

```
def two_sum(a, b):
    return a + b
```

通过查看这段代码，无法安全且毫无疑问地确定函数参数的类型 `two_sum`。它在传入 `int` 值时均可正常工作：

```
print(two_sum(2, 1)) # 结果: 3
```

和字符串：

```
print(two_sum("a", "b")) # 结果: "ab"
```

以及其他值，例如列表、元组等。

由于 Python 类型的动态特性，许多类型都适用于给定的操作，任何类型检查器都无法合理断定是否应该允许调用此函数。

为了帮助我们的类型检查器，我们现在可以在函数定义中提供类型提示，指明我们允许的类型。

为了表示我们只允许 `int` 类型，我们可以将函数定义改为：

```
def two_sum(a: int, b: int):
    return a + b
```

注解跟在参数名后面，用 : 字符分隔。

同样地，为了表示只允许 `str` 类型，我们可以将函数定义改为：

```
def two_sum(a: str, b: str):
    return a + b
```

除了指定参数的类型外，还可以指明函数调用的返回值类型。这是通过在参数列表的右括号后但函数声明末尾的 : 之前添加 -> 字符，后跟类型来实现的：

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

现在我们已经指明调用 `two_sum` 时的返回值类型应为 `int`。同样，我们可以为 `str`、`float`、`list`、`set` 等定义合适的类型。

虽然类型提示主要被类型检查器和集成开发环境使用，但有时你可能需要获取它们。这可以通过使用 `__annotations__` 特殊属性来完成：

```
two_sum.__annotations__
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

# Chapter 87: Type Hints

## Section 87.1: Adding types to a function

Let's take an example of a function which receives two arguments and returns a value indicating their sum:

```
def two_sum(a, b):
    return a + b
```

By looking at this code, one can not safely and without doubt indicate the type of the arguments for function `two_sum`. It works both when supplied with `int` values:

```
print(two_sum(2, 1)) # result: 3
```

and with strings:

```
print(two_sum("a", "b")) # result: "ab"
```

and with other values, such as `lists`, `tuples` et cetera.

Due to this dynamic nature of python types, where many are applicable for a given operation, any type checker would not be able to reasonably assert whether a call for this function should be allowed or not.

To assist our type checker we can now provide type hints for it in the Function definition indicating the type that we allow.

To indicate that we only want to allow `int` types we can change our function definition to look like:

```
def two_sum(a: int, b: int):
    return a + b
```

Annotations follow the argument name and are separated by a : character.

Similarly, to indicate only `str` types are allowed, we'd change our function to specify it:

```
def two_sum(a: str, b: str):
    return a + b
```

Apart from specifying the type of the arguments, one could also indicate the return value of a function call. This is done by adding the -> character followed by the type after the closing parenthesis in the argument list but before the : at the end of the function declaration:

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

Now we've indicated that the return value when calling `two_sum` should be of type `int`. Similarly we can define appropriate values for `str`, `float`, `list`, `set` and others.

Although type hints are mostly used by type checkers and IDEs, sometimes you may need to retrieve them. This can be done using the `__annotations__` special attribute:

```
two_sum.__annotations__
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

## 第87.2节：NamedTuple（具名元组）

使用typing模块中的NamedTuple函数可以创建带有类型提示的具名元组：

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

请注意，生成类型的名称是函数的第一个参数，但应将其赋值给同名变量，以便类型检查器的工作。

## 第87.3节：泛型类型

typing.TypeVar 是一个泛型类型工厂。它的主要目的是作为泛型函数/类/方法注解的参数/占位符：

```
import typing

T = typing.TypeVar("T")

def get_first_element(l: typing.Sequence[T]) -> T:
    """获取序列的第一个元素。””
    return l[0]
```

## 第87.4节：变量和属性

变量使用注释进行注解：

```
x = 3 # type: int
x = negate(x)
x = '类型检查器可能会捕捉到这个错误'

Python 3.x 版本 ≥ 3.6
```

从 Python 3.6 开始，也有了变量注解的新语法。上面的代码可能使用的形式是

```
x: int = 3
```

与注释不同，也可以仅为之前未声明的变量添加类型提示，而不为其赋值：

```
y: int
```

此外，如果这些类型提示用于模块或类级别，可以使用以下方式获取类型提示  
typing.get\_type\_hints(class\_or\_module)：

```
class Foo:
    x: int
    y: str = 'abc'

print(typing.get_type_hints(Foo))
# ChainMap({'x': <class 'int'>, 'y': <class 'str'>}, {})
```

或者，也可以通过使用\_\_annotations\_\_特殊变量或属性来访问：

```
x: int
print(__annotations__)
```

## Section 87.2: NamedTuple

Creating a namedtuple with type hints is done using the function NamedTuple from the typing module:

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

Note that the name of the resulting type is the first argument to the function, but it should be assigned to a variable with the same name to ease the work of type checkers.

## Section 87.3: Generic Types

The `typing.TypeVar` is a generic type factory. Its primary goal is to serve as a parameter/placeholder for generic function/class/method annotations:

```
import typing

T = typing.TypeVar("T")

def get_first_element(l: typing.Sequence[T]) -> T:
    """Gets the first element of a sequence.”
    return l[0]
```

## Section 87.4: Variables and Attributes

Variables are annotated using comments:

```
x = 3 # type: int
x = negate(x)
x = 'a type-checker might catch this error'

Python 3.x Version ≥ 3.6
```

Starting from Python 3.6, there is also [new syntax for variable annotations](#). The code above might use the form

```
x: int = 3
```

Unlike with comments, it is also possible to just add a type hint to a variable that was not previously declared, without setting a value to it:

```
y: int
```

Additionally if these are used in the module or the class level, the type hints can be retrieved using  
typing.get\_type\_hints(class\_or\_module):

```
class Foo:
    x: int
    y: str = 'abc'

print(typing.get_type_hints(Foo))
# ChainMap({'x': <class 'int'>, 'y': <class 'str'>}, {})
```

Alternatively, they can be accessed by using the \_\_annotations\_\_ special variable or attribute:

```
x: int
print(__annotations__)
```

```
# {'x': <class 'int'>}
```

```
class C:  
    s: str  
    print(C.__annotations__)  
    # {'s': <class 'str'>}
```

## 第87.5节：类成员和方法

```
class A:  
    x = None # 类型: float  
    def __init__(self, x: float) -> None:  
        """  
  
    self 不应被注解  
        init 应注解为返回 None  
        """  
        self.x = x  
  
    @classmethod  
    def from_int(cls, x: int) -> 'A':  
        """  
  
    cls 不应被注解  
    使用前向引用通过字符串字面量 'A' 引用当前类  
    """  
        return cls(float(x))
```

由于注解在函数定义时被求值，因此需要当前类的前向引用。

当引用一个类时，如果导入该类会导致循环导入，也可以使用前向引用。

## 第87.6节：关键字参数的类型提示

```
def hello_world(greeting: str = 'Hello'):  
    print(greeting + ' world!')
```

注意等号两边的空格，这与关键字参数通常的写法不同。

```
# {'x': <class 'int'>}
```

```
class C:  
    s: str  
    print(C.__annotations__)  
    # {'s': <class 'str'>}
```

## Section 87.5: Class Members and Methods

```
class A:  
    x = None # type: float  
    def __init__(self, x: float) -> None:  
        """  
        self should not be annotated  
        init should be annotated to return None  
        """  
        self.x = x  
  
    @classmethod  
    def from_int(cls, x: int) -> 'A':  
        """  
        cls should not be annotated  
        Use forward reference to refer to current class with string literal 'A'  
        """  
        return cls(float(x))
```

Forward reference of the current class is needed since annotations are evaluated when the function is defined.  
Forward references can also be used when referring to a class that would cause a circular import if imported.

## Section 87.6: Type hints for keyword arguments

```
def hello_world(greeting: str = 'Hello'):  
    print(greeting + ' world!')
```

Note the spaces around the equal sign as opposed to how keyword arguments are usually styled.

# 第88章：异常

执行过程中检测到的错误称为异常，异常并非无条件致命。大多数异常不会被程序处理；可以编写程序来处理选定的异常。Python中有专门的功能来处理异常和异常逻辑。此外，异常具有丰富的类型层次结构，全部继承自BaseException类型。

## 第88.1节：捕获异常

使用try...except来捕获异常。你应该尽可能指定精确的异常类型：

```
尝试:  
x = 5 / 0  
except ZeroDivisionError as e:  
    # `e` 是异常对象  
    print("遇到除以零错误！异常是：" , e)  
    # 处理异常情况  
x = 0  
finally:  
    print "结束"  
    # 无论如何都会执行。
```

指定的异常类——在本例中为 ZeroDivisionError——会捕获该类或该异常任何子类的异常。

例如，ZeroDivisionError 是 ArithmeticError 的子类：

```
>>> ZeroDivisionError.__bases__  
(<class 'ArithmeticError'>,)
```

因此，以下代码仍然会捕获 ZeroDivisionError：

```
尝试:  
    5/ 0  
except ArithmeticError:  
    print("捕获到算术错误")
```

## 第88.2节：不要捕获所有异常！

虽然通常很想捕获所有 Exception：

```
尝试:  
    very_difficult_function()  
except Exception:  
    # 记录日志 / 尝试重新连接 / 优雅退出  
finally:  
    print "结束"  
    # 无论如何都会执行。
```

甚至包括所有内容（包括BaseException及其所有子类，包括Exception）：

```
尝试:  
    even_more_difficult_function()  
except:  
    pass # 执行所需的操作
```

# Chapter 88: Exceptions

Errors detected during execution are called exceptions and are not unconditionally fatal. Most exceptions are not handled by programs; it is possible to write programs that handle selected exceptions. There are specific features in Python to deal with exceptions and exception logic. Furthermore, exceptions have a rich type hierarchy, all inheriting from the BaseException type.

## Section 88.1: Catching Exceptions

Use `try...except` to catch exceptions. You should specify as precise an exception as you can:

```
try:  
    x = 5 / 0  
except ZeroDivisionError as e:  
    # `e` is the exception object  
    print("Got a divide by zero! The exception was:", e)  
    # handle exceptional case  
    x = 0  
finally:  
    print "The END"  
    # it runs no matter what execute.
```

The exception class that is specified - in this case, `ZeroDivisionError` - catches any exception that is of that class or of any subclass of that exception.

For example, `ZeroDivisionError` is a subclass of `ArithmeticError`:

```
>>> ZeroDivisionError.__bases__  
(<class 'ArithmeticError'>,)
```

And so, the following will still catch the `ZeroDivisionError`:

```
try:  
    5 / 0  
except ArithmeticError:  
    print("Got arithmetic error")
```

## Section 88.2: Do not catch everything!

While it's often tempting to catch every `Exception`:

```
try:  
    very_difficult_function()  
except Exception:  
    # log / try to reconnect / exit graciously  
finally:  
    print "The END"  
    # it runs no matter what execute.
```

Or even everything (that includes BaseException and all its children including `Exception`):

```
try:  
    even_more_difficult_function()  
except:  
    pass # do whatever needed
```

在大多数情况下，这是一种不良的做法。它可能捕获超出预期的异常，例如`SystemExit`、`KeyboardInterrupt`和`MemoryError`——这些通常应与一般的系统或逻辑错误区别对待。它还意味着对内部代码可能出现的错误缺乏清晰的理解，以及如何正确地从该状态中恢复。如果你捕获所有错误，就无法知道发生了什么错误，也不知道如何修复它。

这通常被称为“掩盖错误”，应当避免。让程序崩溃，而不是默默失败，或者更糟的是，在更深层次的执行中失败。（想象这是一个事务系统）

通常这些结构用于程序的最外层，并会记录错误的详细信息，以便修复错误，或更具体地处理错误。

## 第88.3节：重新抛出异常

有时你想捕获异常只是为了检查它，例如用于日志记录。检查完后，你希望异常像之前一样继续传播。

在这种情况下，只需使用无参数的`raise`语句。

```
尝试:  
try:  
    5 / 0  
except ZeroDivisionError:  
    print("发生错误")  
    raise
```

不过请记住，调用栈上层的某些代码仍然可以捕获该异常并以某种方式处理它。此时，`done`输出可能会成为麻烦，因为无论异常是否被捕获，它都会发生。因此，抛出一个不同的异常可能是更好的选择，该异常包含你对情况的说明以及原始异常：

```
尝试:  
try:  
    5 / 0  
except ZeroDivisionError as e:  
    raise ZeroDivisionError("发生错误", e)
```

但这样做的缺点是异常追踪会被缩减到这个`raise`语句，而无参数的`raise`则保留了原始异常追踪。

在Python 3中，你可以使用`raise-from`语法来保留原始堆栈：

```
raise ZeroDivisionError("发生错误") from e
```

## 第88.4节：捕获多个异常

有几种方法可以捕获多个异常。

第一种方法是创建一个包含你希望以相同方式捕获和处理的异常类型的元组。这个示例将使代码忽略`KeyError`和`AttributeError`异常。

```
尝试:  
d = {}  
a = d[1]  
b = d.non_existing_field  
except (KeyError, AttributeError) as e:  
    print("已捕获KeyError或AttributeError异常。")
```

In most cases it's bad practice. It might catch more than intended, such as `SystemExit`, `KeyboardInterrupt` and `MemoryError` - each of which should generally be handled differently than usual system or logic errors. It also means there's no clear understanding for what the internal code may do wrong and how to recover properly from that condition. If you're catching every error, you won't know what error occurred or how to fix it.

This is more commonly referred to as 'bug masking' and should be avoided. Let your program crash instead of silently failing or even worse, failing at deeper level of execution. (Imagine it's a transactional system)

Usually these constructs are used at the very outer level of the program, and will log the details of the error so that the bug can be fixed, or the error can be handled more specifically.

## Section 88.3: Re-raising exceptions

Sometimes you want to catch an exception just to inspect it, e.g. for logging purposes. After the inspection, you want the exception to continue propagating as it did before.

In this case, simply use the `raise` statement with no parameters.

```
try:  
    5 / 0  
except ZeroDivisionError:  
    print("Got an error")  
    raise
```

Keep in mind, though, that someone further up in the caller stack can still catch the exception and handle it somehow. The `done` output could be a nuisance in this case because it will happen in any case (caught or not caught). So it might be a better idea to raise a different exception, containing your comment about the situation as well as the original exception:

```
try:  
    5 / 0  
except ZeroDivisionError as e:  
    raise ZeroDivisionError("Got an error", e)
```

But this has the drawback of reducing the exception trace to exactly this `raise` while the `raise` without argument retains the original exception trace.

In Python 3 you can keep the original stack by using the `raise-from` syntax:

```
raise ZeroDivisionError("Got an error") from e
```

## Section 88.4: Catching multiple exceptions

There are a few ways to [catch multiple exceptions](#).

The first is by creating a tuple of the exception types you wish to catch and handle in the same manner. This example will cause the code to ignore `KeyError` and `AttributeError` exceptions.

```
try:  
    d = {}  
    a = d[1]  
    b = d.non_existing_field  
except (KeyError, AttributeError) as e:  
    print("A KeyError or an AttributeError exception has been caught.")
```

如果你希望以不同方式处理不同的异常，可以为每种类型提供单独的异常块。在这个示例中，我们仍然捕获`KeyError`和`AttributeError`，但以不同的方式处理这些异常。

尝试：

```
d = {}  
a = d[1]  
b = d.non_existing_field  
except KeyError as e:  
    print("发生了KeyError。异常信息:", e)  
except AttributeError as e:  
    print("发生了 AttributeError 异常。异常信息:", e)
```

## 第 88.5 节：异常层次结构

异常处理是基于异常层次结构进行的，该层次结构由异常类的继承结构决定。

例如，`IOError` 和 `OSError` 都是 `EnvironmentError` 的子类。捕获 `IOError` 的代码不会捕获 `OSError`。但是，捕获 `EnvironmentError` 的代码会同时捕获 `IOError` 和 `OSError`。

内置异常的层次结构：

Python 2.x 版本 ≥ 2.3

```
BaseException  
+-- SystemExit  
+-- KeyboardInterrupt  
+-- GeneratorExit  
+-- Exception  
+-- StopIteration  
+-- StandardError  
| +-- BufferError  
| +-- ArithmeticError  
| | +-- FloatingPointError  
| | +-- OverflowError  
| | +-- ZeroDivisionError  
| +-- AssertionError  
| +-- AttributeError  
| +-- EnvironmentError  
| | +-- IOError  
| | +-- OSErr  
| | +-- WindowsError (Windows)  
| | +-- VMSError (VMS)  
| +-- EOFError  
| +-- ImportError  
| +-- LookupError  
| | +-- IndexError  
| | +-- KeyError  
| +-- MemoryError  
| +-- NameError  
| | +-- UnboundLocalError  
| +-- ReferenceError  
| +-- RuntimeError  
| | +-- NotImplementedError  
| +-- SyntaxError  
| | +-- IndentationError  
| | +-- TabError  
| +-- SystemError  
| +-- TypeError  
| +-- ValueError
```

If you wish to handle different exceptions in different ways, you can provide a separate exception block for each type. In this example, we still catch the `KeyError` and `AttributeError`, but handle the exceptions in different manners.

try:

```
d = {}  
a = d[1]  
b = d.non_existing_field  
except KeyError as e:  
    print("A KeyError has occurred. Exception message:", e)  
except AttributeError as e:  
    print("An AttributeError has occurred. Exception message:", e)
```

## Section 88.5: Exception Hierarchy

Exception handling occurs based on an exception hierarchy, determined by the inheritance structure of the exception classes.

For example, `IOError` and `OSError` are both subclasses of `EnvironmentError`. Code that catches an `IOError` will not catch an `OSError`. However, code that catches an `EnvironmentError` will catch both `IOErrors` and `OSErrors`.

The hierarchy of built-in exceptions:

Python 2.x 版本 ≥ 2.3

```
BaseException  
+-- SystemExit  
+-- KeyboardInterrupt  
+-- GeneratorExit  
+-- Exception  
+-- StopIteration  
+-- StandardError  
| +-- BufferError  
| +-- ArithmeticError  
| | +-- FloatingPointError  
| | +-- OverflowError  
| | +-- ZeroDivisionError  
| +-- AssertionError  
| +-- AttributeError  
| +-- EnvironmentError  
| | +-- IOError  
| | +-- OSErr  
| | +-- WindowsError (Windows)  
| | +-- VMSError (VMS)  
| +-- EOFError  
| +-- ImportError  
| +-- LookupError  
| | +-- IndexError  
| | +-- KeyError  
| +-- MemoryError  
| +-- NameError  
| | +-- UnboundLocalError  
| +-- ReferenceError  
| +-- RuntimeError  
| | +-- NotImplementedError  
| +-- SyntaxError  
| | +-- IndentationError  
| | +-- TabError  
| +-- SystemError  
| +-- TypeError  
| +-- ValueError
```

```
| +-+ UnicodeError  
| +-+ UnicodeDecodeError  
| +-+ UnicodeEncodeError  
| +-+ UnicodeTranslateError  
+-+ Warning  
+-+ DeprecationWarning  
+-+ PendingDeprecationWarning  
+-+ RuntimeWarning  
+-+ 语法警告 (SyntaxWarning)  
+-+ 用户警告 (UserWarning)  
+-+ 未来警告 (FutureWarning)  
+-+ 导入警告 (ImportWarning)  
+-+ Unicode警告 (UnicodeWarning)  
+-+ 字节警告 (BytesWarning)
```

Python 3.x 版本 ≥ 3.0

```
BaseException (基类异常)  
+-+ SystemExit (系统退出)  
+-+ KeyboardInterrupt (键盘中断)  
+-+ GeneratorExit (生成器退出)  
+-+ Exception (异常)  
+-+ StopIteration (迭代停止)  
+-+ StopAsyncIteration (异步迭代停止)  
+-+ ArithmeticError (算术错误)  
| +-+ FloatingPointError (浮点错误)  
| +-+ OverflowError (溢出错误)  
| +-+ ZeroDivisionError (除零错误)  
+-+ AssertionError (断言错误)  
+-+ AttributeError (属性错误)  
+-+ BufferError (缓冲区错误)  
+-+ EOFError (文件结束错误)  
+-+ ImportError (导入错误)  
+-+ LookupError (查找错误)  
| +-+ IndexError (索引错误)  
| +-+ KeyError (键错误)  
+-+ MemoryError (内存错误)  
+-+ NameError (名称错误)  
| +-+ UnboundLocalError (未绑定的局部变量错误)  
+-+ OSError (操作系统错误)  
| +-+ BlockingIOError (阻塞IO错误)  
| +-+ ChildProcessError (子进程错误)  
| +-+ ConnectionError (连接错误)  
| | +-+ BrokenPipeError (管道破裂错误)  
| | +-+ ConnectionAbortedError (连接中止错误)  
| | +-+ ConnectionRefusedError (连接拒绝错误)  
| | +-+ ConnectionResetError (连接重置错误)  
| +-+ FileExistsError (文件已存在错误)  
| +-+ FileNotFoundError (文件未找到错误)  
| +-+ InterruptedError (中断错误)  
| +-+ IsADirectoryError (是目录错误)  
| +-+ NotADirectoryError (非目录错误)  
| +-+ PermissionError (权限错误)  
| +-+ ProcessLookupError (进程查找错误)  
| +-+ TimeoutError (超时错误)  
+-+ ReferenceError (引用错误)  
+-+ RuntimeError (运行时错误)  
| +-+ NotImplemented (未实现错误)  
| +-+ RecursionError (递归错误)  
+-+ SyntaxError (语法错误)  
| +-+ IndentationError (缩进错误)  
| +-+ TabError (制表符错误)  
+-+ SystemError (系统错误)  
+-+ TypeError (类型错误)
```

```
| +-+ UnicodeError  
| +-+ UnicodeDecodeError  
| +-+ UnicodeEncodeError  
| +-+ UnicodeTranslateError  
+-+ Warning  
+-+ DeprecationWarning  
+-+ PendingDeprecationWarning  
+-+ RuntimeWarning  
+-+ SyntaxWarning  
+-+ UserWarning  
+-+ FutureWarning  
+-+ ImportWarning  
+-+ UnicodeWarning  
+-+ BytesWarning
```

Python 3.x Version ≥ 3.0

```
BaseException  
+-+ SystemExit  
+-+ KeyboardInterrupt  
+-+ GeneratorExit  
+-+ Exception  
+-+ StopIteration  
+-+ StopAsyncIteration  
+-+ ArithmeticError  
| +-+ FloatingPointError  
| +-+ OverflowError  
| +-+ ZeroDivisionError  
+-+ AssertionError  
+-+ AttributeError  
+-+ BufferError  
+-+ EOFError  
+-+ ImportError  
+-+ LookupError  
| +-+ IndexError  
| +-+ KeyError  
+-+ MemoryError  
+-+ NameError  
| +-+ UnboundLocalError  
+-+ OSError  
| +-+ BlockingIOError  
| +-+ ChildProcessError  
| +-+ ConnectionError  
| | +-+ BrokenPipeError  
| | +-+ ConnectionAbortedError  
| | +-+ ConnectionRefusedError  
| | +-+ ConnectionResetError  
| +-+ FileExistsError  
| +-+ FileNotFoundError  
| +-+ InterruptedError  
| +-+ IsADirectoryError  
| +-+ NotADirectoryError  
| +-+ PermissionError  
| +-+ ProcessLookupError  
| +-+ TimeoutError  
+-+ ReferenceError  
+-+ RuntimeError  
| +-+ NotImplemented  
| +-+ RecursionError  
+-+ SyntaxError  
| +-+ IndentationError  
| +-+ TabError  
+-+ SystemError  
+-+ TypeError
```

```
+-- ValueError
| +- UnicodeError
| +- UnicodeDecodeError
| +- UnicodeEncodeError
| +- UnicodeTranslateError
+- Warning
+- DeprecationWarning
+- PendingDeprecationWarning
+- RuntimeWarning
+- SyntaxWarning
+- UserWarning
+- FutureWarning
+- ImportWarning
+- UnicodeWarning
+- BytesWarning
+- ResourceWarning
```

## 第88.6节：Else

else块中的代码只有在try块中的代码没有引发异常时才会执行。如果你有一些代码不想在抛出异常时运行，但又不希望这些代码抛出的异常被捕获，这种用法非常有用。

例如：

```
尝试:
try:
    data = {1: 'one', 2: 'two'}
    print(data[1])
except KeyError as e:
    print('key not found')
else:
    raise ValueError()
# 输出: one
# 输出: ValueError
```

注意，这种else: 不能与以if开头的elif的else子句结合使用。如果后面有if，它需要保持缩进在该else:下面：

```
尝试:
...
except ...:
...
else:
    if ...:
        ...
    elif ...:
        ...
    else:
        ...
...
```

## 第88.7节：引发异常

如果您的代码遇到它不知道如何处理的情况，例如参数错误，它应该引发适当的异常。

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Did not get an odd number")
```

```
+-- ValueError
| +- UnicodeError
| +- UnicodeDecodeError
| +- UnicodeEncodeError
| +- UnicodeTranslateError
+- Warning
+- DeprecationWarning
+- PendingDeprecationWarning
+- RuntimeWarning
+- SyntaxWarning
+- UserWarning
+- FutureWarning
+- ImportWarning
+- UnicodeWarning
+- BytesWarning
+- ResourceWarning
```

## Section 88.6: Else

Code in an else block will only be run if no exceptions were raised by the code in the try block. This is useful if you have some code you don't want to run if an exception is thrown, but you don't want exceptions thrown by that code to be caught.

For example:

```
try:
    data = {1: 'one', 2: 'two'}
    print(data[1])
except KeyError as e:
    print('key not found')
else:
    raise ValueError()
# Output: one
# Output: ValueError
```

Note that this kind of else: cannot be combined with an if starting the else-clause to an elif. If you have a following if it needs to stay indented below that else::

```
try:
...
except ...:
...
else:
    if ...:
        ...
    elif ...:
        ...
    else:
        ...
...
```

## Section 88.7: Raising Exceptions

If your code encounters a condition it doesn't know how to handle, such as an incorrect parameter, it should raise the appropriate exception.

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Did not get an odd number")
```

```
return odds + 1
```

## 第88.8节：创建自定义异常类型

创建一个继承自Exception的类：

```
class FooException(Exception):
    pass
try:
    raise FooException("insert description here")
except FooException:
    print("引发了一个 FooException。")
```

或者其他异常类型：

```
class NegativeError(ValueError):
    pass

def foo(x):
    # 仅接受正值 x 的函数
    if x < 0:
        raise NegativeError("无法处理负数")
    ... # 函数体的其余部分
try:
    result = foo(int(input("请输入一个正整数: "))) # Python 2.x 中的 raw_input
except NegativeError:
    print("你输入了一个负数!")
else:
    print("结果是 " + str(result))
```

```
return odds + 1
```

## Section 88.8: Creating custom exception types

Create a class inheriting from `Exception`:

```
class FooException(Exception):
    pass
try:
    raise FooException("insert description here")
except FooException:
    print("A FooException was raised.")
```

or another exception type:

```
class NegativeError(ValueError):
    pass

def foo(x):
    # function that only accepts positive values of x
    if x < 0:
        raise NegativeError("Cannot process negative numbers")
    ... # rest of function body
try:
    result = foo(int(input("Enter a positive integer: "))) # raw_input in Python 2.x
except NegativeError:
    print("You entered a negative number!")
else:
    print("The result was " + str(result))
```

## 第88.9节：异常处理的实际示例

### 用户输入

假设你想让用户通过`input`输入一个数字。你想确保输入的是数字。你可以使用`try/except`来实现：

```
Python 3.x 版本 ≥ 3.0
while True:
    尝试:
    nb = int(input('请输入一个数字: '))
    break
except ValueError:
    print('这不是一个数字, 请重试。')
```

注意：Python 2.x 中会使用`raw_input`；函数`input`在 Python 2.x 中存在，但语义不同。在上述示例中，`input`也会接受诸如`2 + 2`之类的表达式，这些表达式会被计算为数字。

如果输入无法转换为整数，则会引发`ValueError`。你可以用`except`捕获它。如果没有引发异常，`break`会跳出循环。循环结束后，`nb`中包含一个整数。

### 字典

假设你正在遍历一个连续整数的列表，比如`range(n)`，并且你有一个字典列表`d`，里面包含了当遇到某些特定整数时要执行的操作信息，比如跳过接下来的`d[i]`个。

```
Python 3.x 版本 ≥ 3.0
while True:
    尝试:
    nb = int(input('Enter a number: '))
    break
except ValueError:
    print('This is not a number, try again.')
```

Note: Python 2.x would use `raw_input` instead; the function `input` exists in Python 2.x but has different semantics. In the above example, `input` would also accept expressions such as `2 + 2` which evaluate to a number.

If the input could not be converted to an integer, a `ValueError` is raised. You can catch it with `except`. If no exception is raised, `break` jumps out of the loop. After the loop, `nb` contains an integer.

### Dictionaries

Imagine you are iterating over a list of consecutive integers, like `range(n)`, and you have a list of dictionaries `d` that contains information about things to do when you encounter some particular integers, say `skip the d[i] next ones`.

```
d = [{7: 3}, {25: 9}, {38: 5}]

for i in range(len(d)):
    do_stuff(i)
    尝试:
    dic = d[i]
        i += dic[i]
    except KeyError:
        i += 1
```

当你尝试从字典中获取一个不存在的键对应的值时，会引发一个KeyError异常。

## 第88.10节：异常也是对象

异常只是继承自内置BaseException的普通Python对象。Python脚本可以使用raise语句中断执行，使Python打印该点的调用栈跟踪以及异常实例的表示。例如：

```
>>> def failing_function():
...     raise ValueError('示例错误！')
>>> failing_function()
追踪 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <模块>
    文件 "<stdin>", 第 2 行, 在 failing_function
ValueError: 示例错误!
```

这表示我们的 failing\_function() 抛出了一个带有消息 '示例错误！' 的 ValueError，该函数是在解释器中执行的。

调用代码可以选择处理调用可能抛出的任何类型的异常：

```
>>> try:
...     failing_function()
... except ValueError:
...     print('错误已处理')
错误已处理
```

你可以通过在异常处理代码的 except... 部分赋值来获取异常对象：

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('捕获异常', repr(e))
捕获异常 ValueError('示例错误!',)
```

内置Python异常及其描述的完整列表可以在Python文档中找到：<https://docs.python.org/3.5/library/exceptions.html>。以下是按层级排列的完整列表：  
异常层级结构。

## 第88.11节：使用finally运行清理代码

有时，无论发生什么异常，你都希望某些操作得以执行，例如，如果你必须清理一些资源。

try语句中的finally块无论是否引发异常都会执行。

```
d = [{7: 3}, {25: 9}, {38: 5}]

for i in range(len(d)):
    do_stuff(i)
    尝试:
    dic = d[i]
        i += dic[i]
    except KeyError:
        i += 1
```

A KeyError will be raised when you try to get a value from a dictionary for a key that doesn't exist.

## Section 88.10: Exceptions are Objects too

Exceptions are just regular Python objects that inherit from the built-in BaseException. A Python script can use the raise statement to interrupt execution, causing Python to print a stack trace of the call stack at that point and a representation of the exception instance. For example:

```
>>> def failing_function():
...     raise ValueError('Example error!')
>>> failing_function()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in failing_function
ValueError: Example error!
```

which says that a ValueError with the message 'Example error!' was raised by our failing\_function(), which was executed in the interpreter.

Calling code can choose to handle any and all types of exception that a call can raise:

```
>>> try:
...     failing_function()
... except ValueError:
...     print('Handled the error')
Handled the error
```

You can get hold of the exception objects by assigning them in the except... part of the exception handling code:

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('Caught exception', repr(e))
Caught exception ValueError('Example error!',)
```

A complete list of built-in Python exceptions along with their descriptions can be found in the Python Documentation: <https://docs.python.org/3.5/library/exceptions.html>. And here is the full list arranged hierarchically: Exception Hierarchy.

## Section 88.11: Running clean-up code with finally

Sometimes, you may want something to occur regardless of whatever exception happened, for example, if you have to clean up some resources.

The finally block of a try clause will happen regardless of whether any exceptions were raised.

```
resource = allocate_some_expensive_resource()
尝试:
do_stuff(resource)
except SomeException as e:
    log_error(e)
    raise # 重新抛出错误
finally:
    free_expensive_resource(resource)
```

这种模式通常更适合使用上下文管理器（使用with语句）来处理。

## 第88.12节：使用 raise from 链接异常

在处理异常的过程中，你可能想要抛出另一个异常。例如，如果在读取文件时遇到 `IOError`，你可能想抛出一个特定于应用程序的错误，以便向你的库的用户展示。

Python 3.x 版本 ≥ 3.0

你可以链接异常以显示异常处理的过程：

```
>>> try:
...     5/ 0
... except ZeroDivisionError as e:
...     raise ValueError("除法失败") from e
```

```
追踪 (最近一次调用最后) :
文件 "<stdin>", 第 2 行, 在 <module>
ZeroDivisionError: 除以零
```

上述异常是以下异常的直接原因：

```
追踪 (最近一次调用最后) :
文件 "<stdin>", 第 4 行, 在 <module>
ValueError: 除法失败
```

```
resource = allocate_some_expensive_resource()
try:
    do_stuff(resource)
except SomeException as e:
    log_error(e)
    raise # re-raise the error
finally:
    free_expensive_resource(resource)
```

This pattern is often better handled with context managers (using the `with` statement).

## Section 88.12: Chain exceptions with raise from

In the process of handling an exception, you may want to raise another exception. For example, if you get an `IOError` while reading from a file, you may want to raise an application-specific error to present to the users of your library, instead.

Python 3.x 版本 ≥ 3.0

You can chain exceptions to show how the handling of exceptions proceeded:

```
>>> try:
...     5 / 0
... except ZeroDivisionError as e:
...     raise ValueError("Division failed") from e
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: Division failed
```

# 第89章：引发自定义错误/异常

Python 有许多内置异常，当程序出现错误时会强制输出错误信息。

但是，有时你可能需要创建满足特定需求的自定义异常。

在 Python 中，用户可以通过创建新类来定义此类异常。该异常类必须直接或间接继承自 `Exception` 类。大多数内置异常也是从该类派生的。

## 第89.1节：自定义异常

这里，我们创建了一个名为 `CustomError` 的用户定义异常，它继承自 `Exception` 类。这个新异常可以像其他异常一样使用 `raise` 语句引发，并可附带可选的错误信息。

```
class CustomError(Exception):
    通过

x = 1

if x == 1:
    raise CustomError('这是自定义错误')
```

输出：

```
追踪（最近一次调用最后）：
文件 "error_custom.py", 第8行, 位于
raise CustomError('这是自定义错误')
__main__.CustomError: 这是自定义错误
```

## 第89.2节：捕获自定义异常

本例展示了如何捕获自定义异常

```
class CustomError(Exception):
    通过

尝试:
    raise CustomError('你能捕获我吗？')
except CustomError as e:
    print ('捕获到自定义错误：{}'.format(e))
except Exception as e:
    print ('通用异常：{}'.format(e))
```

输出：

```
捕获到自定义错误：你能捕获我吗？
```

# Chapter 89: Raise Custom Errors / Exceptions

Python has many built-in exceptions which force your program to output an error when something in it goes wrong.

However, sometimes you may need to create custom exceptions that serve your purpose.

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from `Exception` class. Most of the built-in exceptions are also derived from this class.

## Section 89.1: Custom Exception

Here, we have created a user-defined exception called `CustomError` which is derived from the `Exception` class. This new exception can be raised, like other exceptions, using the `raise` statement with an optional error message.

```
class CustomError(Exception):
    pass

x = 1

if x == 1:
    raise CustomError('This is custom error')
```

Output:

```
Traceback (most recent call last):
File "error_custom.py", line 8, in
raise CustomError('This is custom error')
__main__.CustomError: This is custom error
```

## Section 89.2: Catch custom Exception

This example shows how to catch custom Exception

```
class CustomError(Exception):
    pass

try:
    raise CustomError('Can you catch me ?')
except CustomError as e:
    print ('Catched CustomError :{}'.format(e))
except Exception as e:
    print ('Generic exception: {}'.format(e))
```

Output:

```
Catched CustomError :Can you catch me ?
```

# 视频：Python数据科学与机器学习训练营

学习如何使用NumPy、Pandas、Seaborn、Matplotlib、Plotly、Scikit-Learn、机器学习、Tensorflow等！



- ✓ 使用Python进行数据科学和机器学习
- ✓ 使用Spark进行大数据分析
- ✓ 实现机器学习算法
- ✓ 学习使用NumPy进行数值数据处理
- ✓ 学习使用Pandas进行数据分析
- ✓ 学习使用Matplotlib进行Python绘图
- ✓ 学习使用Seaborn进行统计图表绘制
- ✓ 使用Plotly进行交互式动态可视化
- ✓ 使用SciKit-Learn完成机器学习任务
- ✓ K均值聚类
- ✓ 逻辑回归
- ✓ 线性回归
- ✓ 随机森林和决策树
- ✓ 神经网络
- ✓ 支持向量机今天观看→

# VIDEO: Python for Data Science and Machine Learning Bootcamp

Learn how to use NumPy, Pandas, Seaborn, Matplotlib , Plotly, Scikit-Learn , Machine Learning, Tensorflow, and more!



- ✓ Use Python for Data Science and Machine Learning
- ✓ Use Spark for Big Data Analysis
- ✓ Implement Machine Learning Algorithms
- ✓ Learn to use NumPy for Numerical Data
- ✓ Learn to use Pandas for Data Analysis
- ✓ Learn to use Matplotlib for Python Plotting
- ✓ Learn to use Seaborn for statistical plots
- ✓ Use Plotly for interactive dynamic visualizations
- ✓ Use SciKit-Learn for Machine Learning Tasks
- ✓ K-Means Clustering
- ✓ Logistic Regression
- ✓ Linear Regression
- ✓ Random Forest and Decision Trees
- ✓ Neural Networks
- ✓ Support Vector Machines

**Watch Today →**

# 第90章：英联邦例外情况

在Stack Overflow中，我们经常看到重复的问题讨论相同的错误："ImportError: No module named???????"，SyntaxError：无效的语法或NameError：名称'???'未定义。本文旨在减少这些重复，并提供一些可链接的文档。

## 第90.1节：其他错误

### 断言错误

assert语句几乎存在于所有编程语言中。当你执行：

assert 条件

或者：

assert 条件, 消息

它等同于：

```
if __debug__:  
    if not 条件: raise AssertionError(消息)
```

断言可以包含一个可选的消息，并且在调试完成后可以禁用它们。

注意：内置变量 debug 在正常情况下为 True，优化请求时（命令行选项 -O）为 False。对 debug 的赋值是非法的。内置变量的值在解释器启动时确定。

### KeyboardInterrupt

当用户按下中断键时引发的错误，通常是

**Ctrl** + **C** 或 **删除**。

ZeroDivisionError（零除错误）

你尝试计算 1/0，这是未定义的。请参见此示例以查找一个数的除数：

Python 2.x 版本 ≥ 2.0 版本 ≤ 2.7

```
div = float(raw_input("除数为: "))  
for x in xrange(div+1): #包括数字本身和零  
    if div/x == div//x:  
        print x, "是"的除数, div
```

Python 3.x 版本 ≥ 3.0

```
div = int(input("的除数: "))  
for x in range(div+1): #包括数字本身和零  
    if div/x == div//x:  
        print(x, "是"的除数, div)
```

它会引发ZeroDivisionError，因为for循环将该值赋给了x。正确的写法应该是：

Python 2.x 版本 ≥ 2.0 版本 ≤ 2.7

```
div = float(raw_input("除数为: "))  
for x in xrange(1,div+1): #包括数字本身但不包括零  
    if div/x == div//x:
```

# Chapter 90: Commonwealth Exceptions

Here in Stack Overflow we often see duplicates talking about the same errors: "ImportError: No module named '???????'，SyntaxError：invalid syntax or NameError: name '???' is not defined. This is an effort to reduce them and to have some documentation to link to.

## Section 90.1: Other Errors

### AssertionError

The **assert** statement exists in almost every programming language. When you do:

**assert** condition

or:

**assert** condition, message

It's equivalent to this:

```
if __debug__:  
    if not condition: raise AssertionError(message)
```

Assertions can include an optional message, and you can disable them when you're done debugging.

**Note:** the built-in variable **debug** is True under normal circumstances, False when optimization is requested (command line option -O). Assignments to **debug** are illegal. The value for the built-in variable is determined when the interpreter starts.

### KeyboardInterrupt

Error raised when the user presses the interrupt key, normally **Ctrl** + **C** or **del**.

### ZeroDivisionError

You tried to calculate 1/0 which is undefined. See this example to find the divisors of a number:

Python 2.x Version ≥ 2.0 Version ≤ 2.7

```
div = float(raw_input("Divisors of: "))  
for x in xrange(div+1): #includes the number itself and zero  
    if div/x == div//x:  
        print x, "is a divisor of", div
```

Python 3.x Version ≥ 3.0

```
div = int(input("Divisors of: "))  
for x in range(div+1): #includes the number itself and zero  
    if div/x == div//x:  
        print(x, "is a divisor of", div)
```

It raises **ZeroDivisionError** because the **for** loop assigns that value to x. Instead it should be:

Python 2.x Version ≥ 2.0 Version ≤ 2.7

```
div = float(raw_input("Divisors of: "))  
for x in xrange(1,div+1): #includes the number itself but not zero  
    if div/x == div//x:
```

```
print x, "是"的除数, div
```

Python 3.x 版本 ≥ 3.0

```
div = int(input("的除数: "))
for x in range(1,div+1): #包括数字本身但不包括零
    if div/x == div//x:
        print(x, "是"的除数, div)
```

## 第90.2节：NameError：名称“？”未定义

当你尝试使用未初始化的变量、方法或函数（至少之前未初始化）时，会引发该错误。换句话说，当请求的局部或全局名称未找到时，会引发此错误。可能是你拼写错误了对象名称，或者忘记了import某些内容。也可能它在另一个作用域中。我们将通过单独的示例来讲解这些情况。

### 代码中根本没有定义它

你可能忘记初始化它了，尤其是如果它是一个常量的话

```
foo # 这个变量未定义
bar() # 这个函数未定义
```

### 也许它稍后会被定义：

```
baz()
```

```
def baz():
    通过
```

### 或者它没有被导入：

```
#需要导入 math
```

```
def sqrt():
    x = float(input("值: "))
    return math.sqrt(x)
```

### Python 作用域与 LEGB 规则：

所谓的 LEGB 规则讲述了 Python 的作用域。其名称基于不同的作用域，按相应优先级排序：

局部 (Local) → 闭包 (Enclosed) → 全局 (Global) → 内置 (Built-in)。

- 局部 (Local)：在函数中未声明为全局或未赋值的变量。
- 闭包 (Enclosing)：定义在被另一个函数包裹的函数中的变量。
- 全局 (Global)：声明为全局的变量，或在文件顶层赋值的变量。
- 内置 (Built-in)：预先分配在内置名称模块中的变量。

举个例子：

```
for i in range(4):
    d = i * 2
    print(d)
```

```
print x, "is a divisor of", div
```

Python 3.x Version ≥ 3.0

```
div = int(input("Divisors of: "))
for x in range(1,div+1): #includes the number itself but not zero
    if div/x == div//x:
        print(x, "is a divisor of", div)
```

## Section 90.2: NameError: name '???' is not defined

Is raised when you tried to use a variable, method or function that is not initialized (at least not before). In other words, it is raised when a requested local or global name is not found. It's possible that you misspelt the name of the object or forgot to import something. Also maybe it's in another scope. We'll cover those with separate examples.

### It's simply not defined nowhere in the code

It's possible that you forgot to initialize it, especially if it is a constant

```
foo # This variable is not defined
bar() # This function is not defined
```

### Maybe it's defined later:

```
baz()
```

```
def baz():
    pass
```

### Or it wasn't imported:

```
#needs import math
```

```
def sqrt():
    x = float(input("Value: "))
    return math.sqrt(x)
```

### Python scopes and the LEGB Rule:

The so-called LEGB Rule talks about the Python scopes. Its name is based on the different scopes, ordered by the correspondent priorities:

Local → Enclosed → Global → Built-in.

- Local: Variables not declared global or assigned in a function.
- Enclosing: Variables defined in a function that is wrapped inside another function.
- Global: Variables declared global, or assigned at the top-level of a file.
- Built-in: Variables preassigned in the built-in names module.

As an example:

```
for i in range(4):
    d = i * 2
    print(d)
```

d 是可访问的，因为 `for` 循环不会标记一个新的作用域，但如果它标记了，我们将会遇到错误，其行为将类似于：

```
def noaccess():
    for i in range(4):
        d = i * 2
noaccess()
print(d)
```

Python 报错 NameError：名称 'd' 未定义

## 第90.3节：类型错误

当某个对象的类型应当不同而导致的异常

**TypeError: [定义/方法] 接受 ? 个位置参数，但给定了 ? 个**

函数或方法被调用时传入的参数比它能接受的多（或少）。

### 示例

如果给出更多参数：

```
def foo(a): return a
foo(a,b,c,d) #并且 a,b,c,d 已定义
```

如果给出更少参数：

```
def foo(a,b,c,d): return a += b + c + d
foo(a) #并且 a 已定义
```

注意：如果你想使用未知数量的参数，可以使用 `*args` 或 `**kwargs`。参见 `*args` 和 `**kwargs`

**TypeError: 不支持的操作数类型 [operand] : '???' 和 '??'**

某些类型不能一起操作，具体取决于操作符。

### 示例

例如：`+` 用于连接和相加，但你不能对两种类型都使用它。例如，尝试通过连接（+ing）`'set1'` 和 `'tuple1'` 来创建一个 `set` 会报错。代码：

```
set1, tuple1 = {1, 2}, (3, 4)
a = set1 + tuple1
```

某些类型（例如：`int` 和 `string`）都使用 `+` 但用途不同：

```
b = 400 + 'foo'
```

或者它们甚至可能根本不会被用于任何东西：

```
c = ["a", "b"] - [1, 2]
```

但例如你可以将一个 `float` 加到一个 `int` 上：

d 是可访问的，因为 `for` 循环不会标记一个新的作用域，但如果它标记了，我们将会遇到错误，其行为将类似于：

```
def noaccess():
    for i in range(4):
        d = i * 2
noaccess()
print(d)
```

Python says NameError: name 'd' is not defined

## Section 90.3: TypeErrors

These exceptions are caused when the type of some object should be different

**TypeError: [definition/method] takes ? positional arguments but ? was given**

A function or method was called with more (or less) arguments than the ones it can accept.

### Example

If more arguments are given:

```
def foo(a): return a
foo(a, b, c, d) #And a, b, c, d are defined
```

If less arguments are given:

```
def foo(a, b, c, d): return a += b + c + d
foo(a) #And a is defined
```

**Note:** if you want use an unknown number of arguments, you can use `*args` or `**kwargs`. See `*args` and `**kwargs`

**TypeError: unsupported operand type(s) for [operand]: '???' and '??'**

Some types cannot be operated together, depending on the operand.

### Example

For example: `+` is used to concatenate and add, but you can't use any of them for both types. For instance, trying to make a `set` by concatenating (+ing) `'set1'` and `'tuple1'` gives the error. Code:

```
set1, tuple1 = {1, 2}, (3, 4)
a = set1 + tuple1
```

Some types (eg: `int` and `string`) use both `+` but for different things:

```
b = 400 + 'foo'
```

Or they may not be even used for anything:

```
c = ["a", "b"] - [1, 2]
```

But you can for example add a `float` to an `int`:

```
d = 1 + 1.0
```

### TypeError: '???' 对象不可迭代/不可下标访问：

对于一个对象来说，要可迭代，它必须能接受从零开始的连续索引，直到索引不再有效，并抛出一个`IndexError`（更技术性地说：它必须有一个`__iter__`方法返回一个`__iterator__`，或者定义一个`__getitem__`方法来实现上述功能）。

#### 示例

这里我们说bar是数字1的第零个元素。荒谬：

```
foo = 1
bar = foo[0]
```

这是一个更离散的版本：在这个例子中，`for`试图将x设置为`amount[0]`，即一个可迭代对象的第一个元素，但它做不到，因为`amount`是一个int：

```
amount = 10
for x in amount: print(x)
```

### TypeError: '???' 对象不可调用

你定义了一个变量，之后却像调用函数或方法那样调用它

#### 示例

```
foo = "notAFunction"
foo()
```

## 第90.4节：良好代码中的语法错误

绝大多数情况下，指向无关紧要行的`SyntaxError`意味着前一行存在问题（在此示例中，是缺少括号）：

```
def my_print():
    x = (1 + 1
        print(x)
```

返回

```
文件 "<input>", 第 3 行
    print(x)
    ^
语法错误: 无效的语法
```

此问题最常见的原因是括号/方括号不匹配，如示例所示。

Python 3 中 `print` 语句有一个主要注意事项：

```
Python 3.x 版本 ≥ 3.0
>>> print "hello world"
文件 "<stdin>", 第 1 行
    print "hello world"
    ^
```

```
d = 1 + 1.0
```

### TypeError: '???' object is not iterable/subscriptable:

For an object to be iterable it can take sequential indexes starting from zero until the indexes are no longer valid and a `IndexError` is raised (More technically: it has to have an `__iter__` method which returns an `__iterator__`, or which defines a `__getitem__` method that does what was previously mentioned).

#### Example

Here we are saying that bar is the zeroth item of 1. Nonsense:

```
foo = 1
bar = foo[0]
```

This is a more discrete version: In this example `for` tries to set x to `amount[0]`, the first item in an iterable but it can't because `amount` is an int:

```
amount = 10
for x in amount: print(x)
```

### TypeError: '???' object is not callable

You are defining a variable and calling it later (like what you do with a function or method)

#### Example

```
foo = "notAFunction"
foo()
```

## Section 90.4: Syntax Error on good code

The gross majority of the time a `SyntaxError` which points to an uninteresting line means there is an issue on the line before it (in this example, it's a missing parenthesis):

```
def my_print():
    x = (1 + 1
        print(x)
```

Returns

```
File "<input>", line 3
    print(x)
    ^
SyntaxError: invalid syntax
```

The most common reason for this issue is mismatched parentheses/brackets, as the example shows.

There is one major caveat for `print` statements in Python 3:

```
Python 3.x 版本 ≥ 3.0
>>> print "hello world"
File "<stdin>", line 1
    print "hello world"
    ^
```

因为 `print` 语句被替换了 `print()` 函数, 所以你应该写成 :

```
print("hello world") # 注意这对 Py2 和 Py3 都有效
```

## 第 90.5 节 : 缩进错误 (或缩进语法错误)

在大多数其他语言中, 缩进不是强制性的, 但在Python (以及其他语言: 早期版本的FORTRAN、Makefiles、Whitespace (晦涩语言) 等) 中情况并非如此, 如果你来自其他语言, 或者从示例中复制代码到自己的代码, 或者你是新手, 这可能会让人感到困惑。

### IndentationError/SyntaxError: 意外的缩进

当缩进级别无故增加时, 会引发此异常。

#### 示例

这里没有理由增加缩进级别 :

Python 2.x 版本  $\geq$  2.0 版本  $\leq$  2.7

```
print "这行代码是正确的"
print "这行代码不正确"
```

Python 3.x 版本  $\geq$  3.0

```
print("这行代码是正确的")
print("这行代码不正确")
```

这里有两个错误: 最后一个错误以及缩进不匹配任何缩进级别。然而只显示了一个错误:

Python 2.x 版本  $\geq$  2.0 版本  $\leq$  2.7

```
print "这行代码是正确的"
print "这行代码不正确"
```

Python 3.x 版本  $\geq$  3.0

```
print("这行代码是正确的")
print("这行代码不正确")
```

### IndentationError/SyntaxError: 取消缩进不匹配任何外层缩进级别

看起来你没有完全取消缩进。

#### 示例

Python 2.x 版本  $\geq$  2.0 版本  $\leq$  2.7

```
def foo():
    print "这应该是 foo() 的一部分"
    print "错误!"
print "这不是 foo() 的一部分"
```

Python 3.x 版本  $\geq$  3.0

```
print("这行代码是正确的")
print("这行代码不正确")
```

### IndentationError: 预期有缩进块

Because [the `print` statement was replaced with the `print\(\)` function](#), so you want:

```
print("hello world") # Note this is valid for both Py2 & Py3
```

## Section 90.5: IndentationErrors (or indentation SyntaxErrors)

In most other languages indentation is not compulsory, but in Python (and other languages: early versions of FORTRAN, Makefiles, Whitespace (esoteric language), etc.) that is not the case, what can be confusing if you come from another language, if you were copying code from an example to your own, or simply if you are new.

### IndentationError/SyntaxError: unexpected indent

This exception is raised when the indentation level increases with no reason.

#### Example

There is no reason to increase the level here:

Python 2.x Version  $\geq$  2.0 Version  $\leq$  2.7

```
print "This line is ok"
print "This line isn't ok"
```

Python 3.x Version  $\geq$  3.0

```
print("This line is ok")
print("This line isn't ok")
```

Here there are two errors: the last one and that the indentation does not match any indentation level. However just one is shown:

Python 2.x Version  $\geq$  2.0 Version  $\leq$  2.7

```
print "This line is ok"
print "This line isn't ok"
```

Python 3.x Version  $\geq$  3.0

```
print("This line is ok")
print("This line isn't ok")
```

### IndentationError/SyntaxError: unindent does not match any outer indentation level

Appears you didn't unindent completely.

#### Example

Python 2.x Version  $\geq$  2.0 Version  $\leq$  2.7

```
def foo():
    print "This should be part of foo()"
    print "ERROR!"
print "This is not a part of foo()"
```

Python 3.x Version  $\geq$  3.0

```
print("This line is ok")
print("This line isn't ok")
```

### IndentationError: expected an indented block

冒号后（然后换行）缩进级别必须增加。当没有发生这种情况时，会引发此错误。

#### 示例

```
if ok:  
doStuff()
```

注意：使用关键字 `pass`（什么都不做）来仅仅放置一个 `if`、`else`、`except`、`class`、`method` 或 `definition`，但不说明调用/条件为真时会发生什么（稍后再做，或者在 `except` 的情况下什么都不做）：

```
def foo():  
    通过
```

#### IndentationError: 缩进中制表符和空格使用不一致

#### 示例

```
def foo():  
    if ok:  
        return "Two != Four != Tab"  
    return "我不在乎，我想做什么就做什么"
```

#### 如何避免此错误

不要使用制表符。Python 的风格指南 PEP8 不建议使用制表符。

1. 将你的编辑器设置为使用 4 个空格进行缩进。
2. 进行搜索替换，将所有制表符替换为 4 个空格。
3. 确保你的编辑器设置为将制表符显示为 8 个空格，这样你可以轻松发现错误并修正它。

如果你想了解更多，请参见这个问题。

After a colon (and then a new line) the indentation level has to increase. This error is raised when that didn't happen.

#### Example

```
if ok:  
doStuff()
```

**Note:** Use the keyword `pass` (that makes absolutely nothing) to just put an `if`, `else`, `except`, `class`, `method` or `definition` but not say what will happen if called/condition is true (but do it later, or in the case of `except`: just do nothing):

```
def foo():  
    pass
```

#### IndentationError: inconsistent use of tabs and spaces in indentation

#### Example

```
def foo():  
    if ok:  
        return "Two != Four != Tab"  
    return "i don't care i do whatever i want"
```

#### How to avoid this error

Don't use tabs. It is discouraged by PEP8, the style guide for Python.

1. Set your editor to use 4 **spaces** for indentation.
2. Make a search and replace to replace all tabs with 4 spaces.
3. Make sure your editor is set to **display** tabs as 8 spaces, so that you can realize easily that error and fix it.

See [this](#) question if you want to learn more.

# 第 91 章：urllib

## 第 91.1 节：HTTP GET

Python 2.x 版本 ≤ 2.7

**Python 2**

```
import urllib
response = urllib.urlopen('http://stackoverflow.com/documentation/')
```

使用 `urllib.urlopen()` 会返回一个响应对象，可以像处理文件一样处理它。

```
print response.code
# 输出: 200
```

`response.code` 表示HTTP返回值。200表示成功，404表示未找到，等等。

```
print response.read()
'<!DOCTYPE html>|r<html>|r<head>|r|r<title>Documentation - Stack. 等'
```

`response.read()` 和 `response.readlines()` 可用于读取请求返回的实际HTML文件。

这些方法的操作类似于 `file.read*`

Python 3.x 版本 ≥ 3.0

**Python 3**

```
import urllib.request
```

```
print(urllib.request.urlopen("http://stackoverflow.com/documentation/"))
# 输出: <http.client.HTTPResponse at 0x7f37a97e3b00>
```

```
response = urllib.request.urlopen("http://stackoverflow.com/documentation/")
```

```
print(response.code)
# 输出: 200
print(response.read())
# 输出: b'<!DOCTYPE html>|r<html>|r<head>|r|r<title>Documentation - Stack Overflow</title>'
```

该模块已更新为 Python 3.x，但用法基本相同。`urllib.request.urlopen` 将返回类似的类文件对象。

## 第 91.2 节：HTTP POST

要 POST 数据，将编码后的查询参数作为数据传递给 `urlopen()`

Python 2.x 版本 ≤ 2.7

**Python 2**

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.urlencode(query_parms)
response = urllib.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# 输出: 200
response.read()
# 输出: '<!DOCTYPE html>|r<html>|r<head>|r|r<title>登录 - Stack Overflow'
```

# Chapter 91: urllib

## Section 91.1: HTTP GET

Python 2.x Version ≤ 2.7

**Python 2**

```
import urllib
response = urllib.urlopen('http://stackoverflow.com/documentation/')
```

Using `urllib.urlopen()` will return a response object, which can be handled similar to a file.

```
print response.code
# Prints: 200
```

The `response.code` represents the http return value. 200 is OK, 404 is NotFound, etc.

```
print response.read()
'<!DOCTYPE html>|r\n<html>|r\n<head>|r\n|r\n<title>Documentation - Stack. etc'
```

`response.read()` and `response.readlines()` can be used to read the actual html file returned from the request.

These methods operate similarly to `file.read*`

Python 3.x Version ≥ 3.0

**Python 3**

```
import urllib.request
```

```
print(urllib.request.urlopen("http://stackoverflow.com/documentation/"))
# Prints: <http.client.HTTPResponse at 0x7f37a97e3b00>
```

```
response = urllib.request.urlopen("http://stackoverflow.com/documentation/")
```

```
print(response.code)
# Prints: 200
print(response.read())
# Prints: b'<!DOCTYPE html>|r\n<html>|r\n<head>|r\n|r\n<title>Documentation - Stack Overflow</title>'
```

The module has been updated for Python 3.x, but use cases remain basically the same. `urllib.request.urlopen` will return a similar file-like object.

## Section 91.2: HTTP POST

To POST data pass the encoded query arguments as data to `urlopen()`

Python 2.x Version ≤ 2.7

**Python 2**

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.urlencode(query_parms)
response = urllib.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# Output: 200
response.read()
# Output: '<!DOCTYPE html>|r\n<html>|r\n<head>|r\n|r\n<title>Log In - Stack Overflow'
```

Python 3.x 版本 ≥ 3.0

**Python 3**

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.parse.urlencode(query_parms).encode('utf-8')
response = urllib.request.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# 输出: 200
response.read()
# 输出: b'<!DOCTYPE html>|r<html>...等'
```

## 第91.3节：根据内容类型编码解码接收的字节

接收到的字节必须使用正确的字符编码进行解码，才能被解释为文本：

Python 3.x 版本 ≥ 3.0

```
import urllib.request

response = urllib.request.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().get_content_charset()
html = data.decode(encoding)

Python 2.x 版本 ≤ 2.7
import urllib2
response = urllib2.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().getencoding()
html = data.decode(encoding)
```

Python 3.x Version ≥ 3.0

**Python 3**

```
import urllib
query_parms = {'username':'stackoverflow', 'password':'me.me'}
encoded_parms = urllib.parse.urlencode(query_parms).encode('utf-8')
response = urllib.request.urlopen("https://stackoverflow.com/users/login", encoded_parms)
response.code
# Output: 200
response.read()
# Output: b'<!DOCTYPE html>\r\n<html>....etc'
```

## Section 91.3: Decode received bytes according to content type encoding

The received bytes have to be decoded with the correct character encoding to be interpreted as text:

Python 3.x Version ≥ 3.0

```
import urllib.request

response = urllib.request.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().get_content_charset()
html = data.decode(encoding)

Python 2.x Version ≤ 2.7
import urllib2
response = urllib2.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().getencoding()
html = data.decode(encoding)
```

# 第92章：使用Python进行网页爬取

网页爬取是一种自动化、程序化的过程，通过该过程可以不断地从网页上“爬取”数据。网页爬取也被称为屏幕抓取或网页采集，可以从任何公开访问的网页即时获取数据。在某些网站上，网页爬取可能是非法的。

## 第92.1节：使用Scrapy框架进行爬取

首先，您需要创建一个新的 Scrapy 项目。进入您想存放代码的目录并运行：

```
scrapy startproject projectName
```

要进行爬取，我们需要一个爬虫。爬虫定义了如何爬取某个特定网站。这里是一个爬虫的代码，它跟踪链接到StackOverflow上投票最高的问题，并从每个页面爬取一些数据 ([source](#))：

```
import scrapy

class StackOverflowSpider(scrapy.Spider):
    name = 'stackoverflow' # 每个爬虫都有一个唯一的名称    start_urls = [
        'http://stackoverflow.com/questions?sort=votes'] # 解析从一组特定的URL开始

    def parse(self, response): # 对于该生成器产生的每个请求，其响应会被发送到
        parse_question
        for href in response.css('.question-summary h3 a::attr(href)'): # 使用CSS选择器查找问题的URL，进行一些爬取
            操作
            full_url = response.urljoin(href.extract())
            yield scrapy.Request(full_url, callback=self.parse_question)

    def parse_question(self, response):
        yield {
            'title': response.css('h1 a::text').extract_first(),
            'votes': response.css('.question .vote-count-post::text').extract_first(),
            'body': response.css('.question .post-text').extract_first(),
            'tags': response.css('.question .post-tag::text').extract(),
            'link': response.url,
        }
```

将你的爬虫类保存在projectName\spiders目录中。在本例中是projectName\spiders\stackoverflow\_spider.py。

现在你可以使用你的爬虫了。例如，尝试在项目目录下运行：

```
scrapy crawl stackoverflow
```

## 第92.2节：使用Selenium WebDriver进行爬取

有些网站不喜欢被爬取。在这些情况下，你可能需要模拟真实用户使用浏览器的操作。Selenium可以启动并控制一个网页浏览器。

```
from selenium import webdriver

browser = webdriver.Firefox() # 启动Firefox浏览器

browser.get('http://stackoverflow.com/questions?sort=votes') # 加载网址
```

# Chapter 92: Web scraping with Python

[Web scraping](#) is an automated, programmatic process through which data can be constantly 'scraped' off webpages. Also known as screen scraping or web harvesting, web scraping can provide instant data from any publicly accessible webpage. On some websites, web scraping may be illegal.

## Section 92.1: Scraping using the Scrapy framework

First you have to set up a new Scrapy project. Enter a directory where you'd like to store your code and run:

```
scrapy startproject projectName
```

To scrape we need a spider. Spiders define how a certain site will be scraped. Here's the code for a spider that follows the links to the top voted questions on StackOverflow and scrapes some data from each page ([source](#)):

```
import scrapy

class StackOverflowSpider(scrapy.Spider):
    name = 'stackoverflow' # each spider has a unique name
    start_urls = ['http://stackoverflow.com/questions?sort=votes'] # the parsing starts from a
    specific set of urls

    def parse(self, response): # for each request this generator yields, its response is sent to
        parse_question
        for href in response.css('.question-summary h3 a::attr(href)'): # do some scraping stuff
            using css selectors to find question urls
                full_url = response.urljoin(href.extract())
                yield scrapy.Request(full_url, callback=self.parse_question)

    def parse_question(self, response):
        yield {
            'title': response.css('h1 a::text').extract_first(),
            'votes': response.css('.question .vote-count-post::text').extract_first(),
            'body': response.css('.question .post-text').extract_first(),
            'tags': response.css('.question .post-tag::text').extract(),
            'link': response.url,
        }
```

Save your spider classes in the projectName\spiders directory. In this case - projectName\spiders\stackoverflow\_spider.py.

Now you can use your spider. For example, try running (in the project's directory):

```
scrapy crawl stackoverflow
```

## Section 92.2: Scraping using Selenium WebDriver

Some websites don't like to be scraped. In these cases you may need to simulate a real user working with a browser. Selenium launches and controls a web browser.

```
from selenium import webdriver

browser = webdriver.Firefox() # launch Firefox browser

browser.get('http://stackoverflow.com/questions?sort=votes') # load url
```

```

title = browser.find_element_by_css_selector('h1').text # 页面标题 (第一个 h1 元素)
questions = browser.find_elements_by_css_selector('.question-summary') # 问题列表

for question in questions: # 遍历问题
    question_title = question.find_element_by_css_selector('.summary h3 a').text
    question_excerpt = question.find_element_by_css_selector('.summary .excerpt').text
    question_vote = question.find_element_by_css_selector('.stats .vote .votes .vote-count-post').text

    print "%s%s%s votes-----" % (question_title, question_excerpt, question_vote)

```

Selenium 可以做更多操作。它可以修改浏览器的 cookie，填写表单，模拟鼠标点击，截取网页截图，并运行自定义的 JavaScript。

## 第 92.3 节：使用 requests 和 lxml 抓取一些数据的基本示例

```

# 兼容 Python 2.
from __future__ import print_function

import lxml.html
import requests

def main():
    r = requests.get("https://httpbin.org")
    html_source = r.text
    root_element = lxml.html.fromstring(html_source)
    # 注意 root_element.xpath() 返回的是一个*列表*。
    # XPath 指定了我们想要的元素路径。
    page_title = root_element.xpath('/html/head/title/text()')[0]
    print(page_title)

if __name__ == '__main__':
    main()

```

## 第92.4节：使用 requests 维护网页爬取会话

维护一个[网页爬取会话](#)以保持cookie和其他参数是个好主意。此外，这还能带来性能提升，因为requests.Session会重用与主机的底层TCP连接：

```

import requests

with requests.Session() as session:
    # 通过 session 发出的所有请求现在都带有 User-Agent 头
    session.headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36'}

    # 设置 cookies
    session.get('http://httpbin.org/cookies/set?key=value')

    # 获取 cookies
    response = session.get('http://httpbin.org/cookies')
    print(response.text)

```

```

title = browser.find_element_by_css_selector('h1').text # page title (first h1 element)
questions = browser.find_elements_by_css_selector('.question-summary') # question list

for question in questions: # iterate over questions
    question_title = question.find_element_by_css_selector('.summary h3 a').text
    question_excerpt = question.find_element_by_css_selector('.summary .excerpt').text
    question_vote = question.find_element_by_css_selector('.stats .vote .votes .vote-count-post').text

    print "%s\n%s\n%s votes\n-----\n" % (question_title, question_excerpt, question_vote)

```

Selenium can do much more. It can modify browser's cookies, fill in forms, simulate mouse clicks, take screenshots of web pages, and run custom JavaScript.

## Section 92.3: Basic example of using requests and lxml to scrape some data

```

# For Python 2 compatibility.
from __future__ import print_function

import lxml.html
import requests

def main():
    r = requests.get("https://httpbin.org")
    html_source = r.text
    root_element = lxml.html.fromstring(html_source)
    # Note root_element.xpath() gives a *list* of results.
    # XPath specifies a path to the element we want.
    page_title = root_element.xpath('/html/head/title/text()')[0]
    print(page_title)

if __name__ == '__main__':
    main()

```

## Section 92.4: Maintaining web-scraping session with requests

It is a good idea to maintain a [web-scraping session](#) to persist the cookies and other parameters. Additionally, it can result into a *performance improvement* because requests.Session reuses the underlying TCP connection to a host:

```

import requests

with requests.Session() as session:
    # all requests through session now have User-Agent header set
    session.headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36'}

    # set cookies
    session.get('http://httpbin.org/cookies/set?key=value')

    # get cookies
    response = session.get('http://httpbin.org/cookies')
    print(response.text)

```

## 第92.5节：使用 BeautifulSoup4 进行爬取

```
from bs4 import BeautifulSoup
import requests

# 使用 requests 模块获取页面
res = requests.get('https://www.codechef.com/problems/easy')

# 创建一个 BeautifulSoup 对象
page = BeautifulSoup(res.text, 'lxml') # text 字段包含页面的源代码

# 现在使用 CSS 选择器获取包含问题列表的表格
datatable_tags = page.select('table.dataTable') # 问题位于 <table> 标签中,
                                                # 类名为 "dataTable"

# 我们从列表中提取第一个标签, 因为这是我们想要的
datatable = datatable_tags[0]
# 现在因为我们想要问题名称, 它们包含在 <b> 标签中,
# 这些标签直接嵌套在 <a> 标签下
prob_tags = datatable.select('a > b')
prob_names = [tag.getText().strip() for tag in prob_tags]

print prob_names
```

## 第 92.6 节：使用 urllib.request 简单下载网页内容

标准库模块 `urllib.request` 可用于下载网页内容：

```
from urllib.request import urlopen

response = urlopen('http://stackoverflow.com/questions?sort=votes')
data = response.read()

# 接收到的字节通常应根据响应的字符集进行解码
encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

Python 2 中也有类似的模块。

## 第92.7节：修改 Scrapy 用户代理

有时默认的 Scrapy 用户代理 (`"Scrapy/VERSION (+http://scrapy.org)"`) 会被主机屏蔽。要更改默认用户代理，请打开 `settings.py`，取消注释并编辑以下行为你想要的内容。

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

例如

```
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/51.0.2704.103 Safari/537.36'
```

## 第92.8节：使用 curl 进行爬取

导入：

```
from subprocess import Popen, PIPE
from lxml import etree
```

## Section 92.5: Scraping using BeautifulSoup4

```
from bs4 import BeautifulSoup
import requests

# Use the requests module to obtain a page
res = requests.get('https://www.codechef.com/problems/easy')

# Create a BeautifulSoup object
page = BeautifulSoup(res.text, 'lxml') # the text field contains the source of the page

# Now use a CSS selector in order to get the table containing the list of problems
datatable_tags = page.select('table.dataTable') # The problems are in the <table> tag,
                                                # with class "dataTable"

# We extract the first tag from the list, since that's what we desire
datatable = datatable_tags[0]
# Now since we want problem names, they are contained in <b> tags, which are
# directly nested under <a> tags
prob_tags = datatable.select('a > b')
prob_names = [tag.getText().strip() for tag in prob_tags]

print prob_names
```

## Section 92.6: Simple web content download with urllib.request

The standard library module `urllib.request` can be used to download web content:

```
from urllib.request import urlopen

response = urlopen('http://stackoverflow.com/questions?sort=votes')
data = response.read()

# The received bytes should usually be decoded according the response's character set
encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

A similar module is also available in Python 2.

## Section 92.7: Modify Scrapy user agent

Sometimes the default Scrapy user agent (`"Scrapy/VERSION (+http://scrapy.org)"`) is blocked by the host. To change the default user agent open `settings.py`, uncomment and edit the following line to whatever you want.

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

For example

```
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/51.0.2704.103 Safari/537.36'
```

## Section 92.8: Scraping with curl

imports:

```
from subprocess import Popen, PIPE
from lxml import etree
```

```
from io import StringIO
```

下载中：

```
user_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/55.0.2883.95 Safari/537.36'
url = 'http://stackoverflow.com'
get = Popen(['curl', '-s', '-A', user_agent, url], stdout=PIPE)
result = get.stdout.read().decode('utf8')
```

-s: 静默下载

-A: 用户代理标志

解析中：

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

```
from io import StringIO
```

Downloading:

```
user_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/55.0.2883.95 Safari/537.36'
url = 'http://stackoverflow.com'
get = Popen(['curl', '-s', '-A', user_agent, url], stdout=PIPE)
result = get.stdout.read().decode('utf8')
```

-s: silent download

-A: user agent flag

Parsing:

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

# 第93章：HTML解析

## 第93.1节：在BeautifulSoup中使用CSS选择器

BeautifulSoup对CSS选择器的支持有限，但涵盖了大多数常用选择器。使用select()方法查找多个元素，使用select\_one()方法查找单个元素。

基本示例：

```
from bs4 import BeautifulSoup

data = """
<ul>
<li class="item">item1</li>
<li class="item">item2</li>
<li class="item">item3</li>
</ul>
"""

soup = BeautifulSoup(data, "html.parser")

for item in soup.select("li.item"):
    print(item.get_text())
```

输出：

```
项目1
项目2
项目3
```

## 第93.2节：PyQuery

pyquery是一个类似jquery的Python库。它对css选择器有很好的支持。

```
from pyquery import PyQuery

html = """
<h1>销售</h1>
<table id="table">
<tr>
<td>Lorem</td>
<td>46</td>
</tr>
<tr>
<td>Ipsum</td>
<td>12</td>
</tr>
<tr>
<td>Dolor</td>
<td>27</td>
</tr>
<tr>
<td>Sit</td>
<td>90</td>
</tr>
</table>
"""
```

# Chapter 93: HTML Parsing

## Section 93.1: Using CSS selectors in BeautifulSoup

BeautifulSoup has a [limited support for CSS selectors](#), but covers most commonly used ones. Use **SELECT()** method to find multiple elements and **select\_one()** to find a single element.

Basic example:

```
from bs4 import BeautifulSoup

data = """
<ul>
<li class="item">item1</li>
<li class="item">item2</li>
<li class="item">item3</li>
</ul>
"""

soup = BeautifulSoup(data, "html.parser")

for item in soup.select("li.item"):
    print(item.get_text())
```

Prints:

```
item1
item2
item3
```

## Section 93.2: PyQuery

pyquery is a jquery-like library for python. It has very well support for css selectors.

```
from pyquery import PyQuery

html = """
<h1>Sales</h1>
<table id="table">
<tr>
<td>Lorem</td>
<td>46</td>
</tr>
<tr>
<td>Ipsum</td>
<td>12</td>
</tr>
<tr>
<td>Dolor</td>
<td>27</td>
</tr>
<tr>
<td>Sit</td>
<td>90</td>
</tr>
</table>
"""
```

```

doc = PyQuery(html)

title = doc('h1').text()

print title

table_data = []

rows = doc('#table > tr')
for row in rows:
    name = PyQuery(row).find('td').eq(0).text()
    value = PyQuery(row).find('td').eq(1).text()

    print "%s %s" % (name, value)

```

## 第93.3节：在BeautifulSoup中定位元素后的文本

假设你有以下HTML：

```

<div>
    <label>姓名:</label>
    约翰·史密斯

```

您需要在label元素之后定位文本“John Smith”。

在这种情况下，您可以通过文本定位label元素，然后使用.next\_sibling属性：

```

from bs4 import BeautifulSoup

data = """
<div>
    <label>姓名 :</label>
    约翰·史密斯
"""

soup = BeautifulSoup(data, "html.parser")

label = soup.find("label", text="Name:")
print(label.next_sibling.strip())

```

打印 约翰·史密斯。

```

doc = PyQuery(html)

title = doc('h1').text()

print title

table_data = []

rows = doc('#table > tr')
for row in rows:
    name = PyQuery(row).find('td').eq(0).text()
    value = PyQuery(row).find('td').eq(1).text()

    print "%s\t %s" % (name, value)

```

## Section 93.3: Locate a text after an element in BeautifulSoup

Imagine you have the following HTML:

```

<div>
    <label>Name:</label>
    John Smith
</div>

```

And you need to locate the text "John Smith" after the label element.

In this case, you can locate the label element by text and then use [.next\\_sibling property](#):

```

from bs4 import BeautifulSoup

data = """
<div>
    <label>Name:</label>
    John Smith
</div>
"""

soup = BeautifulSoup(data, "html.parser")

label = soup.find("label", text="Name:")
print(label.next_sibling.strip())

```

Prints John Smith.

# 第94章：操作XML

## 第94.1节：使用ElementTree进行打开和读取

导入 ElementTree 对象，打开相关的 .xml 文件并获取根标签：

```
import xml.etree.ElementTree as ET  
tree = ET.parse("yourXMLfile.xml")  
root = tree.getroot()
```

有几种方法可以遍历树。第一种是通过迭代：

```
for child in root:  
    print(child.tag, child.attrib)
```

或者你可以像访问列表一样引用特定位置：

```
print(root[0][1].text)
```

要按名称搜索特定标签，使用 .find 或 .findall：

```
print(root.findall("myTag"))  
print(root[0].find("myOtherTag"))
```

## 第94.2节：创建和构建 XML 文档

导入 Element Tree 模块

```
import xml.etree.ElementTree as ET
```

Element() 函数用于创建 XML 元素

```
p=ET.Element('parent')
```

SubElement() 函数用于为给定元素创建子元素

```
c = ET.SubElement(p, 'child1')
```

dump() 函数用于输出 XML 元素。

```
ET.dump(p)  
# 输出将如下所示  
#<parent><child1 /></parent>
```

如果想保存到文件，使用 ElementTree() 函数创建 XML 树，使用 write() 方法保存到文件

```
tree = ET.ElementTree(p)  
tree.write("output.xml")
```

Comment() 函数用于在 XML 文件中插入注释。

```
comment = ET.Comment('user comment')  
p.append(comment) #该注释将被添加到父元素中
```

# Chapter 94: Manipulating XML

## Section 94.1: Opening and reading using an ElementTree

Import the ElementTree object, open the relevant .xml file and get the root tag:

```
import xml.etree.ElementTree as ET  
tree = ET.parse("yourXMLfile.xml")  
root = tree.getroot()
```

There are a few ways to search through the tree. First is by iteration:

```
for child in root:  
    print(child.tag, child.attrib)
```

Otherwise you can reference specific locations like a list:

```
print(root[0][1].text)
```

To search for specific tags by name, use the .find or .findall:

```
print(root.findall("myTag"))  
print(root[0].find("myOtherTag"))
```

## Section 94.2: Create and Build XML Documents

Import Element Tree module

```
import xml.etree.ElementTree as ET
```

Element() function is used to create XML elements

```
p=ET.Element('parent')
```

SubElement() function used to create sub-elements to a give element

```
c = ET.SubElement(p, 'child1')
```

dump() function is used to dump xml elements.

```
ET.dump(p)  
# Output will be like this  
#<parent><child1 /></parent>
```

If you want to save to a file create a xml tree with ElementTree() function and to save to a file use write() method

```
tree = ET.ElementTree(p)  
tree.write("output.xml")
```

Comment() function is used to insert comments in xml file.

```
comment = ET.Comment('user comment')  
p.append(comment) #this comment will be appended to parent element
```

## 第 94.3 节：修改 XML 文件

导入 Element Tree 模块并打开 XML 文件，获取一个 XML 元素

```
import xml.etree.ElementTree as ET
tree = ET.parse('sample.xml')
root=tree.getroot()
element = root[0] #获取根元素的第一个子元素
```

可以通过更改字段、添加和修改属性、添加和删除子元素来操作 Element 对象

```
element.set('attribute_name', 'attribute_value') #设置xml元素的属性
element.text="string_text"
```

如果你想删除一个元素，使用 Element.remove() 方法

```
root.remove(element)
```

ElementTree.write() 方法用于将 xml 对象输出到 xml 文件。

```
tree.write('output.xml')
```

## 第94.4节：使用XPath搜索XML

从版本2.7开始，ElementTree 对XPath查询有了更好的支持。XPath是一种语法，允许你像使用SQL搜索数据库一样导航XML。find和findAll函数都支持 XPath。下面的xml将用于本例

```
<Catalog>
  <Books>
    <Book id="1" price="7.95">
      <Title>安卓会梦见电子羊吗？</Title>
      <Author>菲利普·K·迪克</Author>
    </Book>
    <Book id="5" price="5.95">
      <Title>魔法的颜色</Title>
      <Author>特里·普拉切特</Author>
    </Book>
    <Book id="7" price="6.95">
      <Title>世界之眼</Title>
      <Author>罗伯特·乔丹</Author>
    </Book>
  </Books>
</Catalog>
```

搜索所有书籍：

```
import xml.etree.cElementTree as ET
tree = ET.parse('sample.xml')
tree.findall('Books/Book')
```

搜索标题为 '魔法的颜色' 的书籍：

```
tree.find("Books/Book[Title='魔法的颜色']")
# 在比较的右侧总是使用 ''
```

## Section 94.3: Modifying an XML File

Import Element Tree module and open xml file, get an xml element

```
import xml.etree.ElementTree as ET
tree = ET.parse('sample.xml')
root=tree.getroot()
element = root[0] #get first child of root element
```

Element object can be manipulated by changing its fields, adding and modifying attributes, adding and removing children

```
element.set('attribute_name', 'attribute_value') #set the attribute to xml element
element.text="string_text"
```

If you want to remove an element use Element.remove() method

```
root.remove(element)
```

ElementTree.write() method used to output xml object to xml files.

```
tree.write('output.xml')
```

## Section 94.4: Searching the XML with XPath

Starting with version 2.7 ElementTree has a better support for XPath queries. XPath is a syntax to enable you to navigate through an xml like SQL is used to search through a database. Both find and findAll functions support XPath. The xml below will be used for this example

```
<Catalog>
  <Books>
    <Book id="1" price="7.95">
      <Title>Do Androids Dream of Electric Sheep?</Title>
      <Author>Philip K. Dick</Author>
    </Book>
    <Book id="5" price="5.95">
      <Title>The Colour of Magic</Title>
      <Author>Terry Pratchett</Author>
    </Book>
    <Book id="7" price="6.95">
      <Title>The Eye of The World</Title>
      <Author>Robert Jordan</Author>
    </Book>
  </Books>
</Catalog>
```

Searching for all books:

```
import xml.etree.cElementTree as ET
tree = ET.parse('sample.xml')
tree.findall('Books/Book')
```

Searching for the book with title = 'The Colour of Magic':

```
tree.find("Books/Book[Title='The Colour of Magic']")
# always use '' in the right side of the comparison
```

查找id为5的书：

```
tree.find("Books/Book[@id='5']")
# 使用xml属性搜索时，属性名前必须加'@'
```

查找第二本书：

```
tree.find("Books/Book[2]")
# 索引从1开始，而不是0
```

查找最后一本书：

```
tree.find("Books/Book[last()]")
# 'last'是ElementTree中唯一允许的xpath函数
```

搜索所有作者：

```
tree.findall("./Author")
# 使用 // 搜索时必须使用相对路径
```

## 第94.5节：使用 iterparse（增量解析）打开和读取大型XML文件

有时我们不想加载整个XML文件来获取所需信息。在这些情况下，能够增量加载相关部分并在完成后删除它们非常有用。使用 iterparse 函数可以在解析 XML 时编辑存储的元素树。

导入 ElementTree 对象：

```
import xml.etree.ElementTree as ET
```

打开.xml文件并遍历所有元素：

```
for event, elem in ET.iterparse("yourXMLfile.xml"):
... 执行某些操作 ...
```

或者，我们可以只查找特定事件，例如开始/结束标签或命名空间。如果省略此选项（如上所示），则只返回“end”事件：

```
events=("start", "end", "start-ns", "end-ns")
for event, elem in ET.iterparse("yourXMLfile.xml", events=events):
... 执行某些操作 ...
```

下面是一个完整的示例，展示了当我们处理完元素后，如何清除内存中的元素树：

```
for event, elem in ET.iterparse("yourXMLfile.xml", events=("start", "end")):
    if elem.tag == "record_tag" and event == "end":
        print elem.text
        elem.clear()
    ... do something else ...
```

Searching for the book with id = 5:

```
tree.find("Books/Book[@id='5']")
# searches with xml attributes must have '@' before the name
```

Search for the second book:

```
tree.find("Books/Book[2]")
# indexes starts at 1, not 0
```

Search for the last book:

```
tree.find("Books/Book[last()]")
# 'last' is the only xpath function allowed in ElementTree
```

Search for all authors:

```
tree.findall("./Author")
# searches with // must use a relative path
```

## Section 94.5: Opening and reading large XML files using iterparse (incremental parsing)

Sometimes we don't want to load the entire XML file in order to get the information we need. In these instances, being able to incrementally load the relevant sections and then delete them when we are finished is useful. With the iterparse function you can edit the element tree that is stored while parsing the XML.

Import the ElementTree object:

```
import xml.etree.ElementTree as ET
```

Open the .xml file and iterate over all the elements:

```
for event, elem in ET.iterparse("yourXMLfile.xml"):
    ... do something ...
```

Alternatively, we can only look for specific events, such as start/end tags or namespaces. If this option is omitted (as above), only "end" events are returned:

```
events=("start", "end", "start-ns", "end-ns")
for event, elem in ET.iterparse("yourXMLfile.xml", events=events):
    ... do something ...
```

Here is the complete example showing how to clear elements from the in-memory tree when we are finished with them:

```
for event, elem in ET.iterparse("yourXMLfile.xml", events=("start", "end")):
    if elem.tag == "record_tag" and event == "end":
        print elem.text
        elem.clear()
    ... do something else ...
```

# 视频：机器学习 A-Z：动手 Python数据科学

学习如何从两位数据科学专家那里用Python创建机器学习算法。包含代码模板。



- ✓ 精通Python上的机器学习
- ✓ 对多种机器学习模型有良好的直觉
- ✓ 做出准确的预测
- ✓ 进行强有力的数据分析
- ✓ 制作稳健的机器学习模型
- ✓ 为您的业务创造强大的附加价值
- ✓ 将机器学习用于个人目的
- ✓ 处理强化学习、自然语言处理和深度学习等特定主题
- ✓ 掌握降维等高级技术
- ✓ 了解针对每种问题应选择哪种机器学习模型
- ✓ 构建一支强大的机器学习模型军团，并知道如何组合它们以解决任何问题

今天观看 →

# VIDEO: Machine Learning A-Z: Hands-On Python In Data Science

Learn to create Machine Learning Algorithms in Python from two Data Science experts. Code templates included.



- ✓ Master Machine Learning on Python
- ✓ Have a great intuition of many Machine Learning models
- ✓ Make accurate predictions
- ✓ Make powerful analysis
- ✓ Make robust Machine Learning models
- ✓ Create strong added value to your business
- ✓ Use Machine Learning for personal purpose
- ✓ Handle specific topics like Reinforcement Learning, NLP and Deep Learning
- ✓ Handle advanced techniques like Dimensionality Reduction
- ✓ Know which Machine Learning model to choose for each type of problem
- ✓ Build an army of powerful Machine Learning models and know how to combine them to solve any problem

Watch Today →

# 第95章：Python Requests Post

Python Requests 模块在 HTTP POST 方法及其对应 Requests 函数中的文档说明

## 第95.1节：简单的POST

```
from requests import post

foo = post('http://httpbin.org/post', data = {'key':'value'})
```

将执行一个简单的 HTTP POST 操作。发送的数据格式多样，但键值对是最常见的。

### 请求头

请求头可以查看：

```
print(foo.headers)
```

一个示例响应：

```
{'Content-Length': '439', 'X-Processed-Time': '0.000802993774414', 'X-Powered-By': 'Flask',
'Server': 'meinheld/0.6.1', 'Connection': 'keep-alive', 'Via': '1.1 vegur', 'Access-Control-Allow-
Credentials': 'true', 'Date': 'Sun, 21 May 2017 20:56:05 GMT', 'Access-Control-Allow-Origin': '*',
'Content-Type': 'application/json'}
```

标题也可以在发布前准备：

```
headers = {'Cache-Control':'max-age=0',
'Upgrade-Insecure-Requests':'1',
'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/54.0.2840.99 Safari/537.36',
'Content-Type':'application/x-www-form-urlencoded',
'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',
'Referer':'https://www.groupon.com/signup',
'Accept-Encoding':'gzip, deflate, br',
'Accept-Language':'es-ES,es;q=0.8'
}

foo = post('http://httpbin.org/post', headers=headers, data = {'key':'value'})
```

### 编码

编码可以以类似的方式设置和查看：

```
print(foo.encoding)

'utf-8'

foo.encoding = 'ISO-8859-1'
```

### SSL 验证

Requests 默认会验证域名的 SSL 证书。此行为可以被覆盖：

# Chapter 95: Python Requests Post

Documentation for the Python Requests module in the context of the HTTP POST method and its corresponding Requests function

## Section 95.1: Simple Post

```
from requests import post

foo = post('http://httpbin.org/post', data = {'key':'value'})
```

Will perform a simple HTTP POST operation. Posted data can be inmost formats, however key value pairs are most prevalent.

### Headers

Headers can be viewed:

```
print(foo.headers)
```

An example response:

```
{'Content-Length': '439', 'X-Processed-Time': '0.000802993774414', 'X-Powered-By': 'Flask',
'Server': 'meinheld/0.6.1', 'Connection': 'keep-alive', 'Via': '1.1 vegur', 'Access-Control-Allow-
Credentials': 'true', 'Date': 'Sun, 21 May 2017 20:56:05 GMT', 'Access-Control-Allow-Origin': '*',
'Content-Type': 'application/json'}
```

Headers can also be prepared before post:

```
headers = {'Cache-Control':'max-age=0',
'Upgrade-Insecure-Requests':'1',
'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/54.0.2840.99 Safari/537.36',
'Content-Type':'application/x-www-form-urlencoded',
'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',
'Referer':'https://www.groupon.com/signup',
'Accept-Encoding':'gzip, deflate, br',
'Accept-Language':'es-ES,es;q=0.8'
}

foo = post('http://httpbin.org/post', headers=headers, data = {'key':'value'})
```

### Encoding

Encoding can be set and viewed in much the same way:

```
print(foo.encoding)

'utf-8'

foo.encoding = 'ISO-8859-1'
```

### SSL Verification

Requests by default validates SSL certificates of domains. This can be overridden:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, verify=False)
```

## 重定向

任何重定向都会被跟随（例如 http 到 https），这也可以被更改：

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, allow_redirects=False)
```

如果 post 操作被重定向，可以访问此值：

```
print(foo.url)
```

可以查看完整的重定向历史：

```
print(foo.history)
```

## 第95.2节：表单编码数据

```
from requests import post

payload = {'key1' : 'value1',
           'key2' : 'value2'
         }

foo = post('http://httpbin.org/post', data=payload)
```

要通过post操作传递表单编码数据，数据必须以字典形式构造并作为data参数提供。

如果数据不想被表单编码，只需将字符串或整数传递给data参数。

将字典传递给json参数，Requests会自动格式化数据：

```
from requests import post

payload = {'key1' : 'value1', 'key2' : 'value2'}

foo = post('http://httpbin.org/post', json=payload)
```

## 第95.3节：文件上传

使用Requests模块，只需提供文件句柄，而不是获取的内容.read()：

```
from requests import post

files = {'file' : open('data.txt', 'rb')}

foo = post('http://http.org/post', files=files)
```

文件名、content\_type 和头信息也可以设置：

```
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel', {'Expires': '0'})}
```

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, verify=False)
```

## Redirection

Any redirection will be followed (e.g. http to https) this can also be changed:

```
foo = post('http://httpbin.org/post', data = {'key':'value'}, allow_redirects=False)
```

If the post operation has been redirected, this value can be accessed:

```
print(foo.url)
```

A full history of redirects can be viewed:

```
print(foo.history)
```

## Section 95.2: Form Encoded Data

```
from requests import post

payload = {'key1' : 'value1',
           'key2' : 'value2'
         }

foo = post('http://httpbin.org/post', data=payload)
```

To pass form encoded data with the post operation, data must be structured as dictionary and supplied as the data parameter.

If the data does not want to be form encoded, simply pass a string, or integer to the data parameter.

Supply the dictionary to the json parameter for Requests to format the data automatically:

```
from requests import post

payload = {'key1' : 'value1', 'key2' : 'value2'}

foo = post('http://httpbin.org/post', json=payload)
```

## Section 95.3: File Upload

With the Requests module, it's only necessary to provide a file handle as opposed to the contents retrieved with.read():

```
from requests import post

files = {'file' : open('data.txt', 'rb')}

foo = post('http://http.org/post', files=files)
```

Filename, content\_type and headers can also be set:

```
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel', {'Expires': '0'})}
```

```
foo = requests.post('http://httpbin.org/post', files=files)
```

只要作为files参数提供，字符串也可以作为文件发送。

#### 多个文件

多个文件的提供方式与单个文件类似：

```
multiple_files = [
    ('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),
    ('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))]
foo = post('http://httpbin.org/post', files=multiple_files)
```

## 第95.4节：响应

可以从post操作中查看响应代码：

```
from requests import post
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.status_code)
```

#### 返回的数据

访问返回的数据：

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.text)
```

#### 原始响应

在需要访问底层urllib3 response.HTTResponse对象的情况下，可以通过以下方式实现：

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
res = foo.raw
print(res.read())
```

## 第95.5节：认证

#### 简单HTTP认证

简单HTTP认证可以通过以下方式实现：

```
from requests import post
foo = post('http://natas0.natas.labs.overthewire.org', auth=('natas0', 'natas0'))
```

这在技术上是以下内容的简写：

```
from requests import post
from requests.auth import HTTPBasicAuth
```

```
foo = requests.post('http://httpbin.org/post', files=files)
```

Strings can also be sent as a file, as long they are supplied as the files parameter.

#### Multiple Files

Multiple files can be supplied in much the same way as one file:

```
multiple_files = [
    ('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),
    ('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))]
foo = post('http://httpbin.org/post', files=multiple_files)
```

## Section 95.4: Responses

Response codes can be viewed from a post operation:

```
from requests import post
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.status_code)
```

#### Returned Data

Accessing data that is returned:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
print(foo.text)
```

#### Raw Responses

In the instances where you need to access the underlying urllib3 response.HTTResponse object, this can be done by the following:

```
foo = post('http://httpbin.org/post', data={'data' : 'value'})
res = foo.raw
print(res.read())
```

## Section 95.5: Authentication

#### Simple HTTP Authentication

Simple HTTP Authentication can be achieved with the following:

```
from requests import post
foo = post('http://natas0.natas.labs.overthewire.org', auth=('natas0', 'natas0'))
```

This is technically short hand for the following:

```
from requests import post
from requests.auth import HTTPBasicAuth
```

```
foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPBasicAuth('natas0', 'natas0'))
```

## HTTP摘要认证

HTTP摘要认证的实现方式非常相似，Requests提供了一个不同的对象来处理此类认证：

```
from requests import post
from requests.auth import HTTPDigestAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPDigestAuth('natas0', 'natas0'))
```

## 自定义认证

在某些情况下，内置的认证机制可能不够用，想象以下示例：

服务器配置为如果发送方具有正确的用户代理字符串、某个特定的头部值，并通过HTTP基本认证提供正确的凭据，则接受认证。为实现这一点，应准备一个自定义认证类，继承自AuthBase，AuthBase是Requests认证实现的基类：

```
from requests.auth import AuthBase
from requests.auth import _basic_auth_str
from requests._internal_utils import to_native_string

class CustomAuth(AuthBase):

    def __init__(self, secret_header, user_agent, username, password):
        # 在这里设置任何与认证相关的数据
        self.secret_header = secret_header
        self.user_agent = user_agent
        self.username = username
        self.password = password

    def __call__(self, r):
        # 修改并返回请求
        r.headers['X-Secret'] = self.secret_header
        r.headers['User-Agent'] = self.user_agent
        r.headers['Authorization'] = _basic_auth_str(self.username, self.password)

    return r
```

然后可以使用以下代码：

```
foo = get('http://test.com/admin', auth=CustomAuth('SecretHeader', 'CustomUserAgent', 'user',
'password' ))
```

## 第95.6节：代理

每个请求的POST操作都可以配置使用网络代理

### HTTP/S 代理

```
from requests import post

proxies = {
    'http': 'http://192.168.0.128:3128',
    'https': 'http://192.168.0.127:1080',
}
```

```
foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPBasicAuth('natas0', 'natas0'))
```

## HTTP Digest Authentication

HTTP Digest Authentication is done in a very similar way, Requests provides a different object for this:

```
from requests import post
from requests.auth import HTTPDigestAuth

foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPDigestAuth('natas0', 'natas0'))
```

## Custom Authentication

In some cases the built in authentication mechanisms may not be enough, imagine this example:

A server is configured to accept authentication if the sender has the correct user-agent string, a certain header value and supplies the correct credentials through HTTP Basic Authentication. To achieve this a custom authentication class should be prepared, subclassing AuthBase, which is the base for Requests authentication implementations:

```
from requests.auth import AuthBase
from requests.auth import _basic_auth_str
from requests._internal_utils import to_native_string

class CustomAuth(AuthBase):

    def __init__(self, secret_header, user_agent, username, password):
        # setup any auth-related data here
        self.secret_header = secret_header
        self.user_agent = user_agent
        self.username = username
        self.password = password

    def __call__(self, r):
        # modify and return the request
        r.headers['X-Secret'] = self.secret_header
        r.headers['User-Agent'] = self.user_agent
        r.headers['Authorization'] = _basic_auth_str(self.username, self.password)

    return r
```

This can then be utilized with the following code:

```
foo = get('http://test.com/admin', auth=CustomAuth('SecretHeader', 'CustomUserAgent', 'user',
'password' ))
```

## Section 95.6: Proxies

Each request POST operation can be configured to use network proxies

### HTTP/S Proxies

```
from requests import post

proxies = {
    'http': 'http://192.168.0.128:3128',
    'https': 'http://192.168.0.127:1080',
}
```

```
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

HTTP 基本认证可以通过以下方式提供：

```
proxies = {'http': 'http://user:pass@192.168.0.128:312'}
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

## SOCKS 代理

使用 socks 代理需要第三方依赖requests[socks]，安装后，socks 代理的使用方式与 HTTPBasicAuth 非常相似：

```
proxies = {
'http': 'socks5://user:pass@host:port',
'https': 'socks5://user:pass@host:port'
}
```

```
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

```
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

HTTP Basic Authentication can be provided in this manner:

```
proxies = {'http': 'http://user:pass@192.168.0.128:312'}
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

## SOCKS Proxies

The use of socks proxies requires 3rd party dependencies requests[socks], once installed socks proxies are used in a very similar way to HTTPBasicAuth:

```
proxies = {
'http': 'socks5://user:pass@host:port',
'https': 'socks5://user:pass@host:port'
}
```

```
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

# 第96章：分发

## 第96.1节：py2app

要使用py2app框架，您必须先安装它。方法是打开终端并输入以下命令：

```
sudo easy_install -U py2app
```

您也可以使用pip安装该包，命令为：

```
pip install py2app
```

然后为您的Python脚本创建setup文件：

```
py2applet --make-setup MyApplication.py
```

根据需要编辑setup文件的设置，以下是默认设置：

```
"""
这是由 py2applet 生成的 setup.py 脚本
```

用法：

```
python setup.py py2app
```

```
from setuptools import setup
```

```
APP = ['test.py']
DATA_FILES = []
OPTIONS = {'argv_emulation': True}

setup(
    app=APP,
    data_files=DATA_FILES,
    options={'py2app': OPTIONS},
    setup_requires=['py2app'],
)
```

要添加图标文件（该文件必须是.icns扩展名），或在应用程序中包含图像作为引用，请按如下方式更改选项：

```
DATA_FILES = ['myInsertedImage.jpg']
OPTIONS = {'argv_emulation': True, 'iconfile': 'myCoolIcon.icns'}
```

最后在终端输入以下命令：

```
python setup.py py2app
```

脚本应当运行，您将在 dist 文件夹中找到完成的应用程序。

使用以下选项进行更多自定义：

```
optimize (-O) 优化级别：-O1 对应"python -O", -O2 对应
              "python -OO", 以及 -O0 用于禁用 [默认值：-O0]
```

# Chapter 96: Distribution

## Section 96.1: py2app

To use the py2app framework you must install it first. Do this by opening terminal and entering the following command:

```
sudo easy_install -U py2app
```

You can also pip install the packages as:

```
pip install py2app
```

Then create the setup file for your python script:

```
py2applet --make-setup MyApplication.py
```

Edit the settings of the setup file to your liking, this is the default:

```
"""
This is a setup.py script generated by py2applet
```

Usage:

```
    python setup.py py2app
```

```
"""

from setuptools import setup
```

```
APP = ['test.py']
DATA_FILES = []
OPTIONS = {'argv_emulation': True}
```

```
setup(
    app=APP,
    data_files=DATA_FILES,
    options={'py2app': OPTIONS},
    setup_requires=['py2app'],
)
```

To add an icon file (this file must have a .icns extension), or include images in your application as reference, change your options as shown:

```
DATA_FILES = ['myInsertedImage.jpg']
OPTIONS = {'argv_emulation': True, 'iconfile': 'myCoolIcon.icns'}
```

Finally enter this into terminal:

```
python setup.py py2app
```

The script should run and you will find your finished application in the dist folder.

Use the following options for more customization:

```
optimize (-O)      optimization level: -O1 for "python -O", -O2 for
                  "python -OO", and -O0 to disable [default: -O0]
```

includes (-i)	用逗号分隔的模块列表以包含
packages (-p)	用逗号分隔的包列表以包含
extension	打包扩展名 [默认值：.app 用于应用程序, .plugin 用于插件]
extra-scripts	用逗号分隔的额外脚本列表以包含在应用程序或插件中。

includes (-i)	comma-separated list of modules to include
packages (-p)	comma-separated list of packages to include
extension	Bundle extension [default:.app for app, .plugin for plugin]
extra-scripts	comma-separated list of additional scripts to include in an application or plugin.

## 第96.2节 : cx\_Freeze

从这里安装 cx\_Freeze [here](#)

解压文件夹并在该目录下运行以下命令：

```
python setup.py build
sudo python setup.py install
```

为你的 Python 脚本创建一个新目录，并在同一目录下创建一个"setup.py"文件，内容如下：

```
application_title = "我的应用程序" # 使用你自己的应用程序名称
main_python_file = "my_script.py" # 你的 Python 脚本

import sys

from cx_Freeze import setup, Executable

base = None
if sys.platform == "win32":
base = "Win32GUI"

includes = ["atexit", "re"]

setup(
name = application_title,
version = "0.1",
description = "你的描述",
options = {"build_exe" : {"includes" : includes }},
executables = [Executable(main_python_file, base = base)])
```

现在从终端运行你的 setup.py：

```
python setup.py bdist_mac
```

注意：在 El Capitan 上，需要以 root 身份运行并禁用 SIP 模式。

## Section 96.2: cx\_Freeze

Install cx\_Freeze from [here](#)

Unzip the folder and run these commands from that directory:

```
python setup.py build
sudo python setup.py install
```

Create a new directory for your python script and create a "setup.py" file in the same directory with the following content:

```
application_title = "My Application" # Use your own application name
main_python_file = "my_script.py" # Your python script

import sys

from cx_Freeze import setup, Executable

base = None
if sys.platform == "win32":
base = "Win32GUI"

includes = ["atexit", "re"]

setup(
name = application_title,
version = "0.1",
description = "Your Description",
options = {"build_exe" : {"includes" : includes }},
executables = [Executable(main_python_file, base = base)])
```

Now run your setup.py from terminal:

```
python setup.py bdist_mac
```

**NOTE: On El Capitan this will need to be run as root with SIP mode disabled.**

# 第97章：属性对象

## 第97.1节：使用@property装饰器实现可读写属性

如果你想使用@property来实现自定义的设置和获取行为，可以使用以下模式：

```
class Cash(object):
    def __init__(self, value):
        self.value = value
    @property
    def formatted(self):
        return '${:.2f}'.format(self.value)
    @formatted.setter
    def formatted(self, new):
        self.value = float(new[1:])
```

使用方法如下：

```
>>> wallet = Cash(2.50)
>>> print(wallet.formatted)
$2.50
>>> print(wallet.value)
2.5
>>> wallet.formatted = '$123.45'
>>> print(wallet.formatted)
$123.45
>>> print(wallet.value)
123.45
```

## 第97.2节：使用@property装饰器

@property装饰器可以用来定义类中的方法，使其表现得像属性一样。一个有用的例子是，当需要暴露的信息可能需要初次（耗时）查找，之后则可以简单地检索时。

给定某个模块foobar.py：

```
class Foo(object):
    def __init__(self):
        self.__bar = None

    @property
    def bar(self):
        if self.__bar is None:
            self.__bar = some_expensive_lookup_operation()
        return self.__bar
```

然后

```
>>> from foobar import Foo
>>> foo = Foo()
>>> # 打印(foo.bar) # 由于初始化后 bar 为 None, 这将花费一些时间
42
>>> print(foo.bar) # 由于 bar 现在有值, 这样速度快得多
42
```

# Chapter 97: Property Objects

## Section 97.1: Using the @property decorator for read-write properties

If you want to use @property to implement custom behavior for setting and getting, use this pattern:

```
class Cash(object):
    def __init__(self, value):
        self.value = value
    @property
    def formatted(self):
        return '${:.2f}'.format(self.value)
    @formatted.setter
    def formatted(self, new):
        self.value = float(new[1:]))
```

To use this:

```
>>> wallet = Cash(2.50)
>>> print(wallet.formatted)
$2.50
>>> print(wallet.value)
2.5
>>> wallet.formatted = '$123.45'
>>> print(wallet.formatted)
$123.45
>>> print(wallet.value)
123.45
```

## Section 97.2: Using the @property decorator

The @property decorator can be used to define methods in a class which act like attributes. One example where this can be useful is when exposing information which may require an initial (expensive) lookup and simple retrieval thereafter.

Given some module foobar.py:

```
class Foo(object):
    def __init__(self):
        self.__bar = None

    @property
    def bar(self):
        if self.__bar is None:
            self.__bar = some_expensive_lookup_operation()
        return self.__bar
```

Then

```
>>> from foobar import Foo
>>> foo = Foo()
>>> # 打印(foo.bar) # This will take some time since bar is None after initialization
42
>>> print(foo.bar) # This is much faster since bar has a value now
42
```

## 第97.3节：仅重写属性对象的getter、setter或deleter

当你继承一个带有属性的类时，可以通过引用父类上的属性对象，为一个或多个属性的getter、setter或deleter函数提供新的实现：

```
类 BaseClass(object):
    @property
    定义 foo(self):
        返回 some_calculated_value()

    @foo.setter
    定义 foo(self, value):
        do_something_with_value(value)

class DerivedClass(BaseClass):
    @BaseClass.foo.setter
    定义 foo(self, value):
        do_something_different_with_value(value)
```

你也可以添加一个setter或deleter，即使基类之前没有。

## 第97.4节：不使用装饰器的属性用法

虽然使用装饰器语法（带@）很方便，但也有些隐藏性。你可以直接使用属性，不使用装饰器。以下Python 3.x示例展示了这一点：

```
class A:
    p = 1234
    def getX (self):
        return self._x

    def setX (self, value):
        self._x = value

    def getY (self):
        return self._y

    def setY (self, value):
        self._y = 1000 + value    # 奇怪但可能

    def getY2 (self):
        return self._y

    def setY2 (self, value):
        self._y = value

    def getT (self):
        return self._t

    def setT (self, value):
        self._t = value

    def getU (self):
        return self._u + 10000

    def setU (self, value):
        self._u = value - 5000
```

## Section 97.3: Overriding just a getter, setter or a deleter of a property object

When you inherit from a class with a property, you can provide a new implementation for one or more of the property getter, setter or deleter functions, by referencing the property object *on the parent class*:

```
class BaseClass(object):
    @property
    def foo(self):
        return some_calculated_value()

    @foo.setter
    def foo(self, value):
        do_something_with_value(value)

class DerivedClass(BaseClass):
    @BaseClass.foo.setter
    def foo(self, value):
        do_something_different_with_value(value)
```

You can also add a setter or deleter where there was not one on the base class before.

## Section 97.4: Using properties without decorators

While using decorator syntax (with the @) is convenient, it also a bit concealing. You can use properties directly, without decorators. The following Python 3.x example shows this:

```
class A:
    p = 1234
    def getX (self):
        return self._x

    def setX (self, value):
        self._x = value

    def getY (self):
        return self._y

    def setY (self, value):
        self._y = 1000 + value    # Weird but possible

    def getY2 (self):
        return self._y

    def setY2 (self, value):
        self._y = value

    def getT (self):
        return self._t

    def setT (self, value):
        self._t = value

    def getU (self):
        return self._u + 10000

    def setU (self, value):
        self._u = value - 5000
```

```

x, y, y2 = property (getX, setX), property (getY, setY), property (getY2, setY2)
t = property (getT, setT)
u = property (getU, setU)

A.q = 5678

class B:
    def getZ (self):
        return self.z_

    def setZ (self, value):
        self.z_ = value

z = property (getZ, setZ)

class C:
    def __init__ (self):
        self.offset = 1234

    def getW (self):
        return self.w_ + self.offset

    def setW (self, value):
        self.w_ = value - self.offset

w = property (getW, setW)

a1 = A ()
a2 = A ()

a1.y2 = 1000
a2.y2 = 2000

a1.x = 5
a1.y = 6

a2.x = 7
a2.y = 8

a1.t = 77
a1.u = 88

print (a1.x, a1.y, a1.y2)
print (a2.x, a2.y, a2.y2)
print (a1.p, a2.p, a1.q, a2.q)

print (a1.t, a1.u)

b = B ()
c = C ()

b.z = 100100
c.z = 200200
c.w = 300300

print (a1.x, b.z, c.z, c.w)

c.w = 400400
c.z = 500500
b.z = 600600

```

```

x, y, y2 = property (getX, setX), property (getY, setY), property (getY2, setY2)
t = property (getT, setT)
u = property (getU, setU)

A.q = 5678

class B:
    def getZ (self):
        return self.z_

    def setZ (self, value):
        self.z_ = value

z = property (getZ, setZ)

class C:
    def __init__ (self):
        self.offset = 1234

    def getW (self):
        return self.w_ + self.offset

    def setW (self, value):
        self.w_ = value - self.offset

w = property (getW, setW)

a1 = A ()
a2 = A ()

a1.y2 = 1000
a2.y2 = 2000

a1.x = 5
a1.y = 6

a2.x = 7
a2.y = 8

a1.t = 77
a1.u = 88

print (a1.x, a1.y, a1.y2)
print (a2.x, a2.y, a2.y2)
print (a1.p, a2.p, a1.q, a2.q)

print (a1.t, a1.u)

b = B ()
c = C ()

b.z = 100100
c.z = 200200
c.w = 300300

print (a1.x, b.z, c.z, c.w)

c.w = 400400
c.z = 500500
b.z = 600600

```

```
print (a1.x, b.z, c.z, c.w)
```

```
print (a1.x, b.z, c.z, c.w)
```

# 第98章：重载

## 第98.1节：运算符重载

下面是类中可以重载的运算符，以及所需的方法定义，以及运算符在表达式中使用的示例。

**注意：使用other作为变量名不是强制的，但被认为是常规做法。**

运算符	方法	表达式
+ 加法	<code>_add_(self, other)</code>	<code>a1 + a2</code>
- 减法	<code>_sub_(self, other)</code>	<code>a1 - a2</code>
* 乘法	<code>_mul_(self, other)</code>	<code>a1 * a2</code>
@ 矩阵乘法	<code>_matmul_(self, other)</code>	<code>a1 @ a2 (Python 3.5)</code>
/ 除法	<code>_div_(self, other)</code>	<code>a1 / a2 (仅限Python 2)</code>
/ 除法	<code>_truediv_(self, other)</code>	<code>a1 / a2 (Python 3)</code>
// 整除	<code>_floordiv_(self, other)</code>	<code>a1 // a2</code>
% 取模/余数	<code>_mod_(self, other)</code>	<code>a1 % a2</code>
** 幂运算	<code>_pow_(self, other[, modulo])</code>	<code>a1 ** a2</code>
<< 按位左移	<code>_lshift_(self, other)</code>	<code>a1 &lt;&lt; a2</code>
>> 按位右移	<code>_rshift_(self, other)</code>	<code>a1 &gt;&gt; a2</code>
& 按位与	<code>_and_(self, other)</code>	<code>a1 &amp; a2</code>
^ 按位异或	<code>_xor_(self, other)</code>	<code>a1 ^ a2</code>
(按位或)	<code>_or_(self, other)</code>	<code>a1   a2</code>
- 否定 (算术)	<code>_neg_(self)</code>	<code>-a1</code>
+ 正数	<code>_pos_(self)</code>	<code>+a1</code>
~ 按位取反	<code>_invert_(self)</code>	<code>~a1</code>
< 小于	<code>_lt_(self, other)</code>	<code>a1 &lt; a2</code>
<= 小于或等于	<code>_le_(self, other)</code>	<code>a1 &lt;= a2</code>
== 等于	<code>_eq_(self, other)</code>	<code>a1 == a2</code>
!= 不等于	<code>_ne_(self, other)</code>	<code>a1 != a2</code>
> 大于	<code>_gt_(self, other)</code>	<code>a1 &gt; a2</code>
>= 大于或等于	<code>_ge_(self, other)</code>	<code>a1 &gt;= a2</code>
[index] 索引操作符	<code>_getitem_(self, index)</code>	<code>a1[index]</code>
在 In 运算符	<code>_contains_(self, other)</code>	<code>a2 in a1</code>
(*args, ...) 调用	<code>_call_(self, *args, **kwargs)</code>	<code>a1(*args, **kwargs)</code>

`_pow_` 的可选参数 `modulo` 仅由内置函数 `pow` 使用。

每个对应于二元运算符的方法都有一个对应的“右”方法，名称以`_r`开头，例如`__radd__`：

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, other):
        return self.a + other
    def __radd__(self, other):
        print("radd")
```

# Chapter 98: Overloading

## Section 98.1: Operator overloading

Below are the operators that can be overloaded in classes, along with the method definitions that are required, and an example of the operator in use within an expression.

**N.B. The use of `other` as a variable name is not mandatory, but is considered the norm.**

Operator	Method	Expression
+ Addition	<code>__add__(self, other)</code>	<code>a1 + a2</code>
- Subtraction	<code>__sub__(self, other)</code>	<code>a1 - a2</code>
* Multiplication	<code>__mul__(self, other)</code>	<code>a1 * a2</code>
@ Matrix Multiplication	<code>__matmul__(self, other)</code>	<code>a1 @ a2 (Python 3.5)</code>
/ Division	<code>__div__(self, other)</code>	<code>a1 / a2 (Python 2 only)</code>
/ Division	<code>__truediv__(self, other)</code>	<code>a1 / a2 (Python 3)</code>
// Floor Division	<code>__floordiv__(self, other)</code>	<code>a1 // a2</code>
% Modulo/Remainder	<code>__mod__(self, other)</code>	<code>a1 % a2</code>
** Power	<code>__pow__(self, other[, modulo])</code>	<code>a1 ** a2</code>
<< Bitwise Left Shift	<code>__lshift__(self, other)</code>	<code>a1 &lt;&lt; a2</code>
>> Bitwise Right Shift	<code>__rshift__(self, other)</code>	<code>a1 &gt;&gt; a2</code>
& Bitwise AND	<code>__and__(self, other)</code>	<code>a1 &amp; a2</code>
^ Bitwise XOR	<code>__xor__(self, other)</code>	<code>a1 ^ a2</code>
(Bitwise OR)	<code>__or__(self, other)</code>	<code>a1   a2</code>
- Negation (Arithmetic)	<code>__neg__(self)</code>	<code>-a1</code>
+ Positive	<code>__pos__(self)</code>	<code>+a1</code>
~ Bitwise NOT	<code>__invert__(self)</code>	<code>~a1</code>
< Less than	<code>__lt__(self, other)</code>	<code>a1 &lt; a2</code>
<= Less than or Equal to	<code>__le__(self, other)</code>	<code>a1 &lt;= a2</code>
== Equal to	<code>__eq__(self, other)</code>	<code>a1 == a2</code>
!= Not Equal to	<code>__ne__(self, other)</code>	<code>a1 != a2</code>
> Greater than	<code>__gt__(self, other)</code>	<code>a1 &gt; a2</code>
>= Greater than or Equal to	<code>__ge__(self, other)</code>	<code>a1 &gt;= a2</code>
[index] Index operator	<code>__getitem__(self, index)</code>	<code>a1[index]</code>
in In operator	<code>__contains__(self, other)</code>	<code>a2 in a1</code>
(*args, ...) Calling	<code>__call__(self, *args, **kwargs)</code>	<code>a1(*args, **kwargs)</code>

The optional parameter `modulo` for `__pow__` is only used by the `pow` built-in function.

Each of the methods corresponding to a binary operator has a corresponding “right” method which start with `_r`, for example `__radd__`:

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, other):
        return self.a + other
    def __radd__(self, other):
        print("radd")
```

```

return other + self.a

A(1) + 2 # 输出: 3
2 + A(1) # 打印 radd。输出: 3

```

以及对应的就地版本，从`_i`开始：

```

类 B:
定义 __init__(self, b):
    self.b = b
定义 __iadd__(self, other):
    self.b += other
    print("iadd")
    return self

```

```

b = B(2)
b.b      # 输出: 2
b += 1   # 打印 iadd
b.b      # 输出: 3

```

由于这些方法没有什么特别之处，语言的许多其他部分、标准库的部分，甚至第三方模块都会自行添加魔法方法，比如将对象转换为某种类型的方法或检查对象属性的方法。例如，内置的`str()`函数会调用对象的`__str__`方法（如果存在）。以下列出了一些此类用法。

函数	方法	表达式
转换为int	<code>__int__(self)</code>	<code>int(a1)</code>
绝对值函数	<code>__abs__(self)</code>	<code>abs(a1)</code>
转换为str	<code>__str__(self)</code>	<code>str(a1)</code>
转换为unicode	<code>__unicode__(self)</code>	<code>unicode(a1)</code> (仅限Python 2)
字符串表示	<code>__repr__(self)</code>	<code>repr(a1)</code>
转换为bool	<code>__nonzero__(self)</code>	<code>bool(a1)</code>
字符串格式化	<code>__format__(self, formatstr)</code>	<code>"Hi {:abc}" .format(a1)</code>
哈希	<code>__hash__(self)</code>	<code>hash(a1)</code>
长度	<code>__len__(self)</code>	<code>len(a1)</code>
反转	<code>__reversed__(self)</code>	<code>reversed(a1)</code>
向下取整	<code>__floor__(self)</code>	<code>math.floor(a1)</code>
向上取整	<code>__ceil__(self)</code>	<code>math.ceil(a1)</code>

还有用于上下文管理器的特殊方法`__enter__`和`__exit__`，以及更多其他方法。

## 第98.2节：魔术方法/双下划线方法

Python中的魔法方法（也称为dunder，是double-underscore的缩写）具有与其他语言中的运算符重载类似的作用。它们允许一个类定义当其作为一元或二元运算符表达式中的操作数时的行为。它们也作为某些内置函数调用的实现。

考虑这个二维向量的实现。

```

import math

class Vector(object):
    # 实例化
    def __init__(self, x, y):

```

```

return other + self.a

```

```

A(1) + 2 # Out: 3
2 + A(1) # prints radd. Out: 3

```

as well as a corresponding inplace version, starting with `__i`:

```

class B:
    def __init__(self, b):
        self.b = b
    def __iadd__(self, other):
        self.b += other
        print("iadd")
        return self

```

```

b = B(2)
b.b      # Out: 2
b += 1   # prints iadd
b.b      # Out: 3

```

Since there's nothing special about these methods, many other parts of the language, parts of the standard library, and even third-party modules add magic methods on their own, like methods to cast an object to a type or checking properties of the object. For example, the builtin `str()` function calls the object's `__str__` method, if it exists. Some of these uses are listed below.

Function	Method	Expression
Casting to int	<code>__int__(self)</code>	<code>int(a1)</code>
Absolute function	<code>__abs__(self)</code>	<code>abs(a1)</code>
Casting to str	<code>__str__(self)</code>	<code>str(a1)</code>
Casting to unicode	<code>__unicode__(self)</code>	<code>unicode(a1)</code> (Python 2 only)
String representation	<code>__repr__(self)</code>	<code>repr(a1)</code>
Casting to bool	<code>__nonzero__(self)</code>	<code>bool(a1)</code>
String formatting	<code>__format__(self, formatstr)</code>	<code>"Hi {:abc}" .format(a1)</code>
Hashing	<code>__hash__(self)</code>	<code>hash(a1)</code>
Length	<code>__len__(self)</code>	<code>len(a1)</code>
Reversed	<code>__reversed__(self)</code>	<code>reversed(a1)</code>
Floor	<code>__floor__(self)</code>	<code>math.floor(a1)</code>
Ceiling	<code>__ceil__(self)</code>	<code>math.ceil(a1)</code>

There are also the special methods `__enter__` and `__exit__` for context managers, and many more.

## Section 98.2: Magic/Dunder Methods

Magic (also called dunder as an abbreviation for double-underscore) methods in Python serve a similar purpose to operator overloading in other languages. They allow a class to define its behavior when it is used as an operand in unary or binary operator expressions. They also serve as implementations called by some built-in functions.

Consider this implementation of two-dimensional vectors.

```

import math

class Vector(object):
    # instantiation
    def __init__(self, x, y):

```

```

self.x = x
self.y = y

# 一元取反 (-v)
def __neg__(self):
    return Vector(-self.x, -self.y)

# 加法 (v + u)
def __add__(self, other):
    return Vector(self.x + other.x, self.y + other.y)

# 减法 (v - u)
def __sub__(self, other):
    return self + (-other)

# 相等 (v == u)
def __eq__(self, other):
    return self.x == other.x and self.y == other.y

# 绝对值 (abs(v))
def __abs__(self):
    return math.hypot(self.x, self.y)

# str(v)
def __str__(self):
    return '<{0.x}, {0.y}>'.format(self)

# repr(v)
def __repr__(self):
    return 'Vector({0.x}, {0.y})'.format(self)

```

现在可以自然地在各种表达式中使用Vector类的实例。

```

v = Vector(1, 4)
u = Vector(2, 0)

u + v      # Vector(3, 4)
print(u + v) # "<3, 4>" (隐式字符串转换)
u - v      # Vector(1, -4)
u == v      # False
u + v == v + u # True
abs(u + v) # 5.0

```

## 第98.3节：容器和序列类型

可以模拟容器类型，支持通过键或索引访问值。

考虑这个简单的稀疏列表实现，它只存储非零元素以节省内存。

```

类 sparselist(对象):
    定义__init__(self, 大小):
        self.size = 大小
        self.data = {}

    # l[index]
    定义__getitem__(self, 索引):
        如果 索引 < 0:
            索引 += self.size
        if index >= self.size:
            raise IndexError(index)

```

```

self.x = x
self.y = y

# unary negation (-v)
def __neg__(self):
    return Vector(-self.x, -self.y)

# addition (v + u)
def __add__(self, other):
    return Vector(self.x + other.x, self.y + other.y)

# subtraction (v - u)
def __sub__(self, other):
    return self + (-other)

# equality (v == u)
def __eq__(self, other):
    return self.x == other.x and self.y == other.y

# abs(v)
def __abs__(self):
    return math.hypot(self.x, self.y)

# str(v)
def __str__(self):
    return '<{0.x}, {0.y}>'.format(self)

# repr(v)
def __repr__(self):
    return 'Vector({0.x}, {0.y})'.format(self)

```

Now it is possible to naturally use instances of the Vector class in various expressions.

```

v = Vector(1, 4)
u = Vector(2, 0)

u + v      # Vector(3, 4)
print(u + v) # "<3, 4>" (implicit string conversion)
u - v      # Vector(1, -4)
u == v      # False
u + v == v + u # True
abs(u + v) # 5.0

```

## Section 98.3: Container and sequence types

It is possible to emulate container types, which support accessing values by key or index.

Consider this naive implementation of a sparse list, which stores only its non-zero elements to conserve memory.

```

class sparselist(object):
    定义__init__(self, size):
        self.size = size
        self.data = {}

    # l[index]
    定义__getitem__(self, index):
        if index < 0:
            index += self.size
        if index >= self.size:
            raise IndexError(index)

```

```

尝试:
    return self.data[index]
except KeyError:
    return 0.0

# l[index] = value
def __setitem__(self, index, value):
    self.data[index] = value

# del l[index]
def __delitem__(self, index):
    if index in self.data:
        del self.data[index]

# l 中的值
def __contains__(self, value):
    return value == 0.0 or value in self.data.values()

# l 的长度
def __len__(self):
    return self.size

# 遍历 l 中的值: ...
def __iter__(self):
    return (self[i] for i in range(self.size)) # python2 中使用 xrange

```

然后，我们可以像使用普通列表一样使用稀疏列表。

```

l = sparselist(10 ** 6) # 包含一百万个元素的列表
0 in l                # True
10 in l               # False

l[12345] = 10
10 in l               # True
l[12345]               # 10

for v in l:
    pass # 0, 0, 0, ... 10, 0, 0 ... 0

```

## 第98.4节：可调用类型

```

class adder(object):
    def __init__(self, first):
        self.first = first

    # a(...)
    def __call__(self, second):
        return self.first + second

add2 = adder(2)
add2(1) # 3
add2(2) # 4

```

## 第98.5节：处理未实现的行为

如果你的类没有为提供的参数类型实现特定的重载运算符，它应该返回 `NotImplemented`（注意这是一个[特殊常量](#)，不同于`NotImplementedError`）。这将允许Python 尝试其他方法以使操作生效：

```

try:
    return self.data[index]
except KeyError:
    return 0.0

# l[index] = value
def __setitem__(self, index, value):
    self.data[index] = value

# del l[index]
def __delitem__(self, index):
    if index in self.data:
        del self.data[index]

# value in l
def __contains__(self, value):
    return value == 0.0 or value in self.data.values()

# len(l)
def __len__(self):
    return self.size

# for value in l: ...
def __iter__(self):
    return (self[i] for i in range(self.size)) # use xrange for python2

```

Then, we can use a sparselist much like a regular `list`.

```

l = sparselist(10 ** 6) # list with 1 million elements
0 in l                # True
10 in l               # False

l[12345] = 10
10 in l               # True
l[12345]               # 10

for v in l:
    pass # 0, 0, 0, ... 10, 0, 0 ... 0

```

## Section 98.4: Callable types

```

class adder(object):
    def __init__(self, first):
        self.first = first

    # a(...)
    def __call__(self, second):
        return self.first + second

add2 = adder(2)
add2(1) # 3
add2(2) # 4

```

## Section 98.5: Handling unimplemented behaviour

If your class doesn't implement a specific overloaded operator for the argument types provided, it should `return NotImplemented` (**note** that this is a [special constant](#), not the same as `NotImplementedError`). This will allow Python to fall back to trying other methods to make the operation work:

当`NotImplemented`被返回时，解释器将尝试对另一种类型执行反射操作，或根据运算符尝试其他回退方法。如果所有尝试的操作都返回`NotImplemented`，解释器将引发相应的异常。

例如，给定 `x + y`，如果 `x.__add__(y)` 返回未实现，则尝试 `y.__radd__(x)`。

```
class NotAddable(object):
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
        return NotImplemented

class Addable(NotAddable):
    def __add__(self, other):
        return Addable(self.value + other.value)
    __radd__ = __add__
```

由于这是反射方法，我们必须实现`__add__`和`__radd__`以在所有情况下获得预期的行为；幸运的是，在这个简单的例子中它们做的是同样的事情，我们可以采取捷径。

使用示例：

```
>>> x = NotAddable(1)
>>> y = Addable(2)
>>> x + x
追踪 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <module>
TypeError: 不支持的操作数类型 用于 +: 'NotAddable' 和 'NotAddable'
>>> y + y
<so.Addable 对象 位于 0x1095974d0>
>>> z = x + y
>>> z
<so.Addable object at 0x109597510>
>>> z.value
3
```

When `NotImplemented` is returned, the interpreter will then try the reflected operation on the other type, or some other fallback, depending on the operator. If all attempted operations return `NotImplemented`, the interpreter will raise an appropriate exception.

For example, given `x + y`, if `x.__add__(y)` returns unimplemented, `y.__radd__(x)` is attempted instead.

```
class NotAddable(object):
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
        return NotImplemented

class Addable(NotAddable):
    def __add__(self, other):
        return Addable(self.value + other.value)
    __radd__ = __add__
```

As this is the *reflected* method we have to implement `__add__` and `__radd__` to get the expected behaviour in all cases; fortunately, as they are both doing the same thing in this simple example, we can take a shortcut.

In use:

```
>>> x = NotAddable(1)
>>> y = Addable(2)
>>> x + x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NotAddable' and 'NotAddable'
>>> y + y
<so.Addable object at 0x1095974d0>
>>> z = x + y
>>> z
<so.Addable object at 0x109597510>
>>> z.value
3
```

# 第99章：多态

## 第99.1节：鸭子类型

Python中由于其动态类型系统所支持的鸭子类型形式的无继承多态。这意味着只要类包含相同的方法，Python解释器就不会区分它们，因为调用的检查仅在运行时进行。

```
类 Duck:  
    定义 quack(self):  
        打印("Quaaaaaack!")  
    定义 feathers(self):  
        print("这只鸭子有白色和灰色的羽毛。")
```

```
class Person:  
    定义 quack(self):  
        print("这个人模仿鸭子。")  
    def feathers(self):  
        print("这个人从地上捡起一根羽毛并展示它。")  
    def name(self):  
        print("约翰·史密斯")
```

```
def in_the_forest(obj):  
    obj.quack()  
    obj.feathers()
```

```
唐纳德 = 鸭子()  
约翰 = 人()  
在森林中(唐纳德)  
在森林中(约翰)
```

输出是：

```
呱呱呱！  
这只鸭子有白色和灰色的羽毛。  
这个人模仿鸭子。  
这个人从地上捡起一根羽毛并展示它。
```

## 第99.2节：基本多态性

多态性是指能够对一个对象执行操作而不考虑其类型的能力。这通常通过创建一个基类，并让两个或多个子类实现具有相同签名的方法来实现。

任何其他操作这些对象的函数或方法都可以调用相同的方法，而不需要先进行类型检查，无论它操作的是哪种类型的对象。在面向对象术语中，当类X继承类Y时，Y称为超类或基类，X称为子类或派生类。

```
class Shape:
```

```
    这是一个父类，旨在被其他类继承
```

```
    定义 calculate_area(self):
```

```
        该方法旨在由子类重写。  
        如果子类未实现该方法但被调用，将引发NotImplemented异常。
```

# Chapter 99: Polymorphism

## Section 99.1: Duck Typing

Polymorphism without inheritance in the form of duck typing as available in Python due to its dynamic typing system. This means that as long as the classes contain the same methods the Python interpreter does not distinguish between them, as the only checking of the calls occurs at run-time.

```
class Duck:  
    定义 quack(self):  
        打印("Quaaaaaack!")  
    定义 feathers(self):  
        print("The duck has white and gray feathers.")
```

```
class Person:  
    定义 quack(self):  
        print("The person imitates a duck.")  
    def feathers(self):  
        print("The person takes a feather from the ground and shows it.")  
    def name(self):  
        print("John Smith")
```

```
def in_the_forest(obj):  
    obj.quack()  
    obj.feathers()
```

```
donald = Duck()  
john = Person()  
in_the_forest(donald)  
in_the_forest(john)
```

The output is:

```
Quaaaaaack!  
The duck has white and gray feathers.  
The person imitates a duck.  
The person takes a feather from the ground and shows it.
```

## Section 99.2: Basic Polymorphism

Polymorphism is the ability to perform an action on an object regardless of its type. This is generally implemented by creating a base class and having two or more subclasses that all implement methods with the same signature. Any other function or method that manipulates these objects can call the same methods regardless of which type of object it is operating on, without needing to do a type check first. In object-oriented terminology when class X extends class Y, then Y is called super class or base class and X is called subclass or derived class.

```
class Shape:
```

```
    This is a parent class that is intended to be inherited by other classes
```

```
    定义 calculate_area(self):
```

```
        This method is intended to be overridden in subclasses.  
        If a subclass doesn't implement it but it is called, NotImplemented will be raised.
```

```

    """
raise NotImplemented

class Square(Shape):
    """
这是 Shape 类的子类, 表示一个正方形
    """
side_length = 2      # 在此示例中, 边长为 2 个单位

def calculate_area(self):
    """
该方法重写了 Shape.calculate_area()。当调用 Square 类型对象的 calculate_area() 方法时, 调用
的将是此方法, 而不是父类的版本。
    """

它执行此形状 (正方形) 所需的计算, 并返回结果。

    """
return self.side_length * 2

```

```

class Triangle(Shape):
    """
这也是 Shape 类的子类, 表示一个三角形
    """
base_length = 4
height = 3

def calculate_area(self):
    """
该方法也重写了 Shape.calculate_area(), 并执行三角形的面积计算, 返回结果。
    """

    """
return 0.5 * self.base_length * self.height

```

```

def get_area(input_obj):
    """
该函数接受一个输入对象, 并调用该对象的
    calculate_area() 方法。注意对象类型未指定,
    它可以是 Square、Triangle 或 Shape 对象。
    """

    print(input_obj.calculate_area())

```

```

# 创建每个类的一个对象
shape_obj = Shape()
square_obj = Square()
triangle_obj = Triangle()

# 现在将每个对象依次传递给 get_area() 函数并查看
# 结果。
get_area(shape_obj)
get_area(square_obj)
get_area(triangle_obj)

```

我们应该看到以下输出：

无
4

```

    """
raise NotImplemented

class Square(Shape):
    """
This is a subclass of the Shape class, and represents a square
    """
side_length = 2      # in this example, the sides are 2 units long

def calculate_area(self):
    """
This method overrides Shape.calculate_area(). When an object of type
Square has its calculate_area() method called, this is the method that
will be called, rather than the parent class' version.

It performs the calculation necessary for this shape, a square, and
returns the result.
    """

return self.side_length * 2

```

```

class Triangle(Shape):
    """
This is also a subclass of the Shape class, and it represents a triangle
    """
base_length = 4
height = 3

def calculate_area(self):
    """
This method also overrides Shape.calculate_area() and performs the area
calculation for a triangle, returning the result.
    """

return 0.5 * self.base_length * self.height

```

```

def get_area(input_obj):
    """
This function accepts an input object, and will call that object's
calculate_area() method. Note that the object type is not specified. It
could be a Square, Triangle, or Shape object.
    """

    print(input_obj.calculate_area())

```

```

# Create one object of each class
shape_obj = Shape()
square_obj = Square()
triangle_obj = Triangle()

# Now pass each object, one at a time, to the get_area() function and see the
# result.
get_area(shape_obj)
get_area(square_obj)
get_area(triangle_obj)

```

We should see this output:

None
4

## 没有多态会发生什么？

没有多态，在对对象执行操作之前，可能需要进行类型检查以确定调用的正确方法。以下计数器示例执行与之前代码相同的任务，但没有使用多态，`get_area()`函数必须做更多的工作。

类 Square：

边长 = 2

```
定义 calculate_square_area(self):
    return self.side_length ** 2
```

class 三角形：

```
base_length = 4
height = 3
```

```
def 计算三角形面积(self):
    return (0.5 * self.base_length) * self.height
```

def get\_area(input\_obj):

```
# 注意这里现在必须进行类型检查。这些类型检查
# 对于更复杂的例子可能会变得非常复杂，导致
# 代码重复且难以维护。
```

```
if type(input_obj).__name__ == "Square":
area = input_obj.calculate_square_area()
```

```
elif type(input_obj).__name__ == "Triangle":
area = input_obj.calculate_triangle_area()
```

print(area)

```
# 创建每个类的一个对象
square_obj = Square()
triangle_obj = Triangle()
```

```
# 现在将每个对象依次传递给 get_area() 函数并查看
# 结果。
```

```
get_area(square_obj)
get_area(triangle_obj)
```

我们应该看到以下输出：

```
4
6.0
```

### 重要提示

请注意，反例中使用的类是“新式”类，并且如果使用的是Python 3，则会隐式继承自`object`类。多态性在Python 2.x和3.x中都能正常工作，但多态性

反例代码如果在Python 2.x解释器中运行会引发异常，因为`type(input_obj).name`会返回“`instance`”而不是类名，如果它们没有显式继承自`object`，导致`area`永远不会被赋值。

## What happens without polymorphism?

Without polymorphism, a type check may be required before performing an action on an object to determine the correct method to call. The following **counter example** performs the same task as the previous code, but without the use of polymorphism, the `get_area()` function has to do more work.

`class Square:`

```
side_length = 2
```

```
def calculate_square_area(self):
    return self.side_length ** 2
```

`class Triangle:`

```
base_length = 4
height = 3
```

```
def calculate_triangle_area(self):
    return (0.5 * self.base_length) * self.height
```

`def get_area(input_obj):`

```
# Notice the type checks that are now necessary here. These type checks
# could get very complicated for a more complex example, resulting in
# duplicate and difficult to maintain code.
```

```
if type(input_obj).__name__ == "Square":
    area = input_obj.calculate_square_area()
```

```
elif type(input_obj).__name__ == "Triangle":
    area = input_obj.calculate_triangle_area()
```

`print(area)`

```
# Create one object of each class
square_obj = Square()
triangle_obj = Triangle()
```

```
# Now pass each object, one at a time, to the get_area() function and see the
# result.
```

```
get_area(square_obj)
get_area(triangle_obj)
```

We should see this output:

```
4
6.0
```

### Important Note

Note that the classes used in the counter example are “new style” classes and implicitly inherit from the `object` class if Python 3 is being used. Polymorphism will work in both Python 2.x and 3.x, but the polymorphism counterexample code will raise an exception if run in a Python 2.x interpreter because `type(input_obj).name` will return “`instance`” instead of the class name if they do not explicitly inherit from `object`, resulting in `area` never being assigned to.

# 视频：机器学习、数据科学和使用 Python 的深度学习

完整的动手机器学习教程，涵盖  
数据科学、Tensorflow、人工智能  
和神经网络



- ✓ 使用Tensorflow和Keras构建人工神经网络
- ✓ 使用深度学习对图像、数据和情感进行分类
- ✓ 使用线性回归、多项式回归和多元回归进行预测
- ✓ 使用Matplotlib和Seaborn进行数据可视化
- ✓ 使用 Apache Spark 的 MLLib 实现大规模机器学习
- ✓ 理解强化学习——以及如何构建一个吃豆人机器人
- ✓ 使用 K-Means 聚类、支持向量机 (SVM) 、KNN、决策树、朴素贝叶斯和主成分分析 (PCA) 对数据进行分类
- ✓ 使用训练/测试和 K 折交叉验证来选择和调整模型
- ✓ 使用基于物品和基于用户的协同过滤构建电影推荐系统

今天观看 →

# VIDEO: Machine Learning, Data Science and Deep Learning with Python

Complete hands-on machine learning tutorial with data science, Tensorflow, artificial intelligence, and neural networks



- ✓ Build artificial neural networks with Tensorflow and Keras
- ✓ Classify images, data, and sentiments using deep learning
- ✓ Make predictions using linear regression, polynomial regression, and multivariate regression
- ✓ Data Visualization with Matplotlib and Seaborn
- ✓ Implement machine learning at massive scale with Apache Spark's MLLib
- ✓ Understand reinforcement learning - and how to build a Pac-Man bot
- ✓ Classify data using K-Means clustering, Support Vector Machines (SVM), KNN, Decision Trees, Naive Bayes, and PCA
- ✓ Use train/test and K-Fold cross validation to choose and tune your models
- ✓ Build a movie recommender system using item-based and user-based collaborative filtering

Watch Today →

# 第100章：方法重写

## 第100.1节：基本方法重写

以下是 Python 中基本重写的示例（为清晰起见，并兼容 Python 2 和 3，使用新式类和带括号的 print）：

```
class Parent(object):
    def introduce(self):
        print("Hello!")

    def print_name(self):
        print("Parent")

class Child(Parent):
    def print_name(self):
        print("Child")

p = Parent()
c = Child()

p.introduce()
p.print_name()

c.introduce()
c.print_name()

$ python basic_override.py
Hello!
父类
你好！
子类
```

当创建子类时，它会继承父类的方法。这意味着父类拥有的任何方法，子类也会拥有。在示例中，introduce是为子类定义的，因为它在父类中定义，尽管在子类的类定义中没有明确声明。

在这个例子中，重写发生在子类定义了自己的print\_name方法时。如果没有声明这个方法，那么c.print\_name()将会打印“父类”。然而，子类重写了父类的print\_name定义，因此现在调用c.print\_name()时，会打印“子类”这个词。

# Chapter 100: Method Overriding

## Section 100.1: Basic method overriding

Here is an example of basic overriding in Python (for the sake of clarity and compatibility with both Python 2 and 3, using new style class and `print` with ()):

```
class Parent(object):
    def introduce(self):
        print("Hello!")

    def print_name(self):
        print("Parent")

class Child(Parent):
    def print_name(self):
        print("Child")

p = Parent()
c = Child()

p.introduce()
p.print_name()

c.introduce()
c.print_name()

$ python basic_override.py
Hello!
Parent
Hello!
Child
```

When the Child class is created, it inherits the methods of the Parent class. This means that any methods that the parent class has, the child class will also have. In the example, the introduce is defined for the Child class because it is defined for Parent, despite not being defined explicitly in the class definition of Child.

In this example, the overriding occurs when Child defines its own print\_name method. If this method was not declared, then `c.print_name()` would have printed "Parent". However, Child has overridden the Parent's definition of print\_name, and so now upon calling `c.print_name()`, the word "Child" is printed.

# 第101章：用户自定义方法

## 第101.1节：创建用户自定义方法对象

当获取类的属性时（可能通过该类的实例），如果该属性是用户自定义的函数对象、未绑定的用户自定义方法对象，或类方法对象，则可能会创建用户自定义方法对象。

```
类 A(object):
    # func : 一个用户自定义的函数对象
    #
    # 请注意, func 在定义时是一个函数对象,
    # 在被检索时是一个未绑定的方法对象。
    def func(self):
        通过

    # classMethod: 一个类方法
    @classmethod
    def classMethod(self):
        通过

class B(object):
    # unboundMeth: 一个未绑定的用户定义方法对象
    #
    # Parent.func 是一个未绑定的用户定义方法对象,
    # 因为它是被检索到的。
    unboundMeth = A.func

a = A()
b = B()

print A.func
# 输出: <unbound method A.func>
print a.func
# 输出: <bound method A.func of <__main__.A object at 0x10e9ab910>>
print B.unboundMeth
# 输出: <unbound method A.func>
print b.unboundMeth
# 输出: <unbound method A.func>
print A.classMethod
# 输出: <bound method type.classMethod of <class '__main__.A'>>
print a.classMethod
# 输出: <bound method type.classMethod of <class '__main__.A'>>
```

当属性是用户定义的方法对象时，只有当检索该属性的类与原始方法对象中存储的类相同或是其派生类时，才会创建一个新的方法对象；否则，将直接使用原始方法对象。

```
# Parent: 原始方法对象中存储的类
class Parent(object):
    # func: 原始方法对象的底层函数
    def func(self):
        通过
func2 = func

# 子类: Parent的派生类
class Child(Parent):
    func = Parent.func
```

# Chapter 101: User-Defined Methods

## Section 101.1: Creating user-defined method objects

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object, an unbound user-defined method object, or a class method object.

```
class A(object):
    # func: A user-defined function object
    #
    # Note that func is a function object when it's defined,
    # and an unbound method object when it's retrieved.
    def func(self):
        pass

    # classMethod: A class method
    @classmethod
    def classMethod(self):
        pass

class B(object):
    # unboundMeth: A unbound user-defined method object
    #
    # Parent.func is an unbound user-defined method object here,
    # because it's retrieved.
    unboundMeth = A.func

a = A()
b = B()

print A.func
# output: <unbound method A.func>
print a.func
# output: <bound method A.func of <__main__.A object at 0x10e9ab910>>
print B.unboundMeth
# output: <unbound method A.func>
print b.unboundMeth
# output: <unbound method A.func>
print A.classMethod
# output: <bound method type.classMethod of <class '__main__.A'>>
print a.classMethod
# output: <bound method type.classMethod of <class '__main__.A'>>
```

When the attribute is a user-defined method object, a new method object is only created if the class from which it is being retrieved is the same as, or a derived class of, the class stored in the original method object; otherwise, the original method object is used as it is.

```
# Parent: The class stored in the original method object
class Parent(object):
    # func: The underlying function of original method object
    def func(self):
        pass
func2 = func

# Child: A derived class of Parent
class Child(Parent):
    func = Parent.func
```

```
# AnotherClass : 一个不同的类，既不是子类也没有被继承
class AnotherClass(object):
    func = Parent.func

    print Parent.func is Parent.func          # False, 创建了新的对象
    print Parent.func2 is Parent.func2         # False, 创建了新的对象
    print Child.func is Child.func            # False, 创建了新的对象
    print AnotherClass.func is AnotherClass.func # True, 使用了原始对象
```

## 第101.2节：海龟示例

下面是一个示例，展示如何使用用户自定义函数，在脚本中轻松调用多次（无限次）。

```
import turtle, time, random #告诉Python我们需要3个不同的模块
turtle.speed(0) #将绘图速度设置为最快
turtle.colormode(255) #特殊的颜色模式
turtle.pensize(4) #绘制线条的粗细def triangle(size): #这是我们自己的
#函数，括号内是我们定义的变量，仅在此函数中使用。该函数绘制一个直角三角形
#开始此函数，我们向前移动，移动的距离由变量 size 决定
#向右转 90 度
#再次向前移动，距离由变量决定
#再次向右转
#闭合三角形。根据毕达哥拉斯定理，我们知道这条线必须是另外两条线（如果它们相等）的 1.5
#倍长while(1): #无限循环
#将绘图点设置为随机的 (x,y) 位置
#随机生成 RGB 颜色
triangle(random.randint(5, 55)) #调用我们的函数，因为它只有一个变量，我们可以直接在括号中传入一个值。传入的值将
#在 5 到 55 之间随机，最终只是改变三角形的大小。
#再次随机生成颜色
```

```
# AnotherClass: A different class, neither subclasses nor subclassed
class AnotherClass(object):
    func = Parent.func

    print Parent.func is Parent.func          # False, new object created
    print Parent.func2 is Parent.func2         # False, new object created
    print Child.func is Child.func            # False, new object created
    print AnotherClass.func is AnotherClass.func # True, original object used
```

## Section 101.2: Turtle example

The following is an example of using an user-defined function to be called multiple( $\infty$ ) times in a script with ease.

```
import turtle, time, random #tell python we need 3 different modules
turtle.speed(0) #set draw speed to the fastest
turtle.colormode(255) #special colormode
turtle.pensize(4) #size of the lines that will be drawn
def triangle(size): #This is our own function, in the parenthesis is a variable we have defined that
#will be used in THIS FUNCTION ONLY. This function creates a right triangle
    turtle.forward(size) #to begin this function we go forward, the amount to go forward by is the
#variable size
    turtle.right(90) #turn right by 90 degree
    turtle.forward(size) #go forward, again with variable
    turtle.right(135) #turn right again
    turtle.forward(size * 1.5) #close the triangle. thanks to the Pythagorean theorem we know that
#this line must be 1.5 times longer than the other two(if they are equal)
while(1): #INFINITE LOOP
    turtle.setpos(random.randint(-200, 200), random.randint(-200, 200)) #set the draw point to a
#random (x,y) position
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize the RGB color
    triangle(random.randint(5, 55)) #use our function, because it has only one variable we can
#simply put a value in the parenthesis. The value that will be sent will be random between 5 - 55, end
#the end it really just changes how big the triangle is.
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
#randomize color again
```

# 第102章：类实例的字符串表示：`__str__` 和 `__repr__` 方法

## 第102.1节：动机

所以你刚刚用Python创建了你的第一个类，一个封装扑克牌的小巧类：

```
类 Card：  
    定义 __init__(self, 花色, 点数):  
        self.花色 = 花色  
        self.点数 = 点数
```

在代码的其他地方，你创建了这个类的几个实例：

```
黑桃A = Card('Spades', 1)  
梅花4 = Card('Clubs', 4)  
红心6 = Card('Hearts', 6)
```

你甚至创建了一副牌的列表，用来表示一手牌：

```
我的手牌 = [黑桃A, 梅花4, 红心6]
```

现在，在调试时，你想看看你的手牌长什么样，所以你自然而然地写了：

```
print(my_hand)
```

但是你得到的是一堆乱码：

```
[<__main__.Card 实例位于 0x000000002533788>,  
<__main__.Card 实例位于 0x0000000025B95C8>,  
<__main__.Card 实例位于 0x0000000025FF508>]
```

感到困惑，你尝试只打印一张牌：

```
print(ace_of_spades)
```

结果你又得到了这个奇怪的输出：

```
<__main__.Card 实例位于 0x000000002533788>
```

别担心。我们马上来解决这个问题。

不过首先，理解这里发生了什么很重要。当你写下`print(ace_of_spades)`时，你告诉Python你想打印代码中调用的Card实例`ace_of_spades`的信息。公平地说，它确实做到了。

该输出由两个重要部分组成：对象的[类型](#)和对象的[id](#)。仅第二部分（十六进制数字）就足以唯一标识[print](#)调用时的对象。[\[1\]](#)

实际上发生的情况是，你让Python“用文字表达”那个对象的本质，然后显示给你。一个更明确的版本可能是：

# Chapter 102: String representations of class instances: `__str__` and `__repr__` methods

## Section 102.1: Motivation

So you've just created your first class in Python, a neat little class that encapsulates a playing card:

```
class Card:  
    def __init__(self, suit, pips):  
        self.suit = suit  
        self.pips = pips
```

Elsewhere in your code, you create a few instances of this class:

```
ace_of_spades = Card('Spades', 1)  
four_of_clubs = Card('Clubs', 4)  
six_of_hearts = Card('Hearts', 6)
```

You've even created a list of cards, in order to represent a "hand":

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

Now, during debugging, you want to see what your hand looks like, so you do what comes naturally and write:

```
print(my_hand)
```

But what you get back is a bunch of gibberish:

```
[<__main__.Card instance at 0x000000002533788>,  
<__main__.Card instance at 0x0000000025B95C8>,  
<__main__.Card instance at 0x0000000025FF508>]
```

Confused, you try just printing a single card:

```
print(ace_of_spades)
```

And again, you get this weird output:

```
<__main__.Card instance at 0x000000002533788>
```

Have no fear. We're about to fix this.

First, however, it's important to understand what's going on here. When you wrote `print(ace_of_spades)` you told Python you wanted it to print information about the Card instance your code is calling `ace_of_spades`. And to be fair, it did.

That output is comprised of two important bits: the [type](#) of the object and the object's [id](#). The second part alone (the hexadecimal number) is enough to uniquely identify the object at the time of the [print](#) call.[\[1\]](#)

What really went on was that you asked Python to "put into words" the essence of that object and then display it to you. A more explicit version of the same machinery might be:

```
string_of_card = str(ace_of_spades)
print(string_of_card)
```

在第一行，你尝试将你的Card实例转换为字符串，在第二行你显示它。

## 问题

你遇到的问题是因为，虽然你告诉了Python关于Card类的所有必要信息以便你可以创建卡牌，但你没有告诉它如何将Card实例转换为字符串。

由于它不知道，当你（隐式地）写下str(ace\_of\_spades)时，它给了你看到的内容，即Card实例的通用表示。

## 解决方案（第一部分）

但是我们可以告诉Python我们希望如何将自定义类的实例转换为字符串。我们这样做的方法是使用`_str_`“dunder”（双下划线）或“魔法”方法。

每当你告诉Python从类实例创建字符串时，它会查找类中的`_str_`方法，并调用它。

考虑以下更新后的Card类版本：

```
类 Card :
    定义 __init__(self, 花色, 点数):
        self.花色 = 花色
        self.点数 = 点数

    def __str__(self):
        special_names = {1:'王牌', 11:'杰克', 12:'皇后', 13:'国王'}

        card_name = special_names.get(self.点数, str(self.点数))

        return "%s 的 %s" % (card_name, self.花色)
```

这里，我们现在在Card类上定义了`_str_`方法，该方法在对人头牌进行简单的字典查找后，返回一个按照我们决定格式化的字符串。

（注意这里“returns”是加粗的，强调返回字符串的重要性，而不仅仅是打印它。）

打印它似乎也能工作，但那样的话，当你执行类似  
`str(ace_of_spades)`时，即使主程序中没有调用打印函数，也会打印出牌面。所以明确一点，确保  
`_str_`返回一个字符串。）

`_str_`方法是一个方法，因此第一个参数是`self`，并且它不应接受或传入额外的参数。

回到我们以更用户友好的方式显示牌面的问题，如果我们再次运行：

```
ace_of_spades = Card('黑桃', 1)
print(ace_of_spades)
```

我们将看到我们的输出要好得多：

黑桃A

太棒了，我们完成了，对吧？

```
string_of_card = str(ace_of_spades)
print(string_of_card)
```

In the first line, you try to turn your Card instance into a string, and in the second you display it.

## The Problem

The issue you're encountering arises due to the fact that, while you told Python everything it needed to know about the Card class for you to *create* cards, you *didn't* tell it how you wanted Card instances to be converted to strings.

And since it didn't know, when you (implicitly) wrote `str(ace_of_spades)`, it gave you what you saw, a generic representation of the Card instance.

## The Solution (Part 1)

But we *can* tell Python how we want instances of our custom classes to be converted to strings. And the way we do this is with the `__str__` "dunder" (for double-underscore) or "magic" method.

Whenever you tell Python to create a string from a class instance, it will look for a `__str__` method on the class, and call it.

Consider the following, updated version of our Card class:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

        card_name = special_names.get(self.pips, str(self.pips))

        return "%s of %s" % (card_name, self.suit)
```

Here, we've now defined the `__str__` method on our Card class which, after a simple dictionary lookup for face cards, **returns** a string formatted however we decide.

(Note that "returns" is in bold here, to stress the importance of returning a string, and not simply printing it. Printing it may seem to work, but then you'd have the card printed when you did something like `str(ace_of_spades)`, without even having a print function call in your main program. So to be clear, make sure that `__str__` returns a string.)

The `__str__` method is a method, so the first argument will be `self` and it should neither accept, nor be passed additional arguments.

Returning to our problem of displaying the card in a more user-friendly manner, if we again run:

```
ace_of_spades = Card('Spades', 1)
print(ace_of_spades)
```

We'll see that our output is much better:

Ace of Spades

So great, we're done, right?

不过，为了万无一失，我们再仔细检查一下，确认已经解决了遇到的第一个问题，即打印列表中的Card实例，hand。

所以我们重新检查以下代码：

```
我的手牌 = [黑桃A, 梅花4, 红心6]
print(my_hand)
```

结果，令我们惊讶的是，又出现了那些奇怪的十六进制代码：

```
[<__main__.Card instance at 0x00000000026F95C8>,
<__main__.Card instance at 0x000000000273F4C8>,
<__main__.Card instance at 0x0000000002732E08>]
```

这是怎么回事？我们已经告诉Python如何显示我们的Card实例，为什么它似乎忘记了呢？

## 解决方案（第二部分）

实际上，当Python想要获取列表中项目的字符串表示时，背后的机制有些不同。事实证明，Python对此目的并不关心`_str_`方法。

相反，它会寻找另一种方法，`_repr_`，如果找不到，就会退回到“十六进制的东西”[2]。所以你的意思是我必须写两个方法来做同样的事情？一个是当我想单独打印我的牌时用，另一个是在某种容器中时用？

不，不过我们先来看一下如果我们同时实现了`_str_`和`_repr_`方法，我们的类会是什么样子：

```
类 Card：
special_names = {1:'王牌', 11:'杰克', 12:'皇后', 13:'国王'}

定义 __init__(self, 花色, 点数):
    self.花色 = 花色
    self.点数 = 点数

def __str__(self):
card_name = Card.special_names.get(self.pips, str(self.pips))
return "%s 的 %s (%S)" % (card_name, self.suit)

def __repr__(self):
card_name = Card.special_names.get(self.pips, str(self.pips))
return "%s 的 %s (%R)" % (card_name, self.suit)
```

这里，两个方法`_str_`和`_repr_`的实现完全相同，只是为了区分这两个方法，(S)被添加到`_str_`返回的字符串中，(R)被添加到`_repr_`返回的字符串中。

注意，和我们的`_str_`方法一样，`_repr_`不接受任何参数并返回一个字符串。

现在我们可以看到每种情况对应的是哪个方法：

```
黑桃A = Card('黑桃', 1)
梅花4 = Card('梅花', 4)
红心6 = Card('红心', 6)
```

```
我的手牌 = [黑桃A, 梅花4, 红心6]
```

Well just to cover our bases, let's double check that we've solved the first issue we encountered, printing the list of Card instances, the hand.

So we re-check the following code:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
print(my_hand)
```

And, to our surprise, we get those funny hex codes again:

```
[<__main__.Card instance at 0x00000000026F95C8>,
<__main__.Card instance at 0x000000000273F4C8>,
<__main__.Card instance at 0x0000000002732E08>]
```

What's going on? We told Python how we wanted our Card instances to be displayed, why did it apparently seem to forget?

## The Solution (Part 2)

Well, the behind-the-scenes machinery is a bit different when Python wants to get the string representation of items in a list. It turns out, Python doesn't care about `__str__` for this purpose.

Instead, it looks for a different method, `__repr__`, and if that's not found, it falls back on the "hexadecimal thing".[2]

*So you're saying I have to make two methods to do the same thing? One for when I want to print my card by itself and another when it's in some sort of container?*

No, but first let's look at what our class would be like if we were to implement both `__str__` and `__repr__` methods:

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (%S)" % (card_name, self.suit)

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (%R)" % (card_name, self.suit)
```

Here, the implementation of the two methods `__str__` and `__repr__` are exactly the same, except that, to differentiate between the two methods, (S) is added to strings returned by `__str__` and (R) is added to strings returned by `__repr__`.

Note that just like our `__str__` method, `__repr__` accepts no arguments and returns a string.

We can see now what method is responsible for each case:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs', 4)
six_of_hearts = Card('Hearts', 6)
```

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

```
print(my_hand)          # [黑桃A (R), 梅花4 (R), 红心6 (R)]  
print(ace_of_spades)    # 黑桃A (S)
```

如前所述，当我们将Card实例传递给print时，会调用`__str__`方法；当我们将实例列表传递给print时，会调用`__repr__`方法。

此时值得指出的是，就像我们之前使用`str()`显式创建自定义类实例的字符串一样，我们也可以使用内置函数`repr()`显式创建类的字符串表示。

例如：

```
str_card = str(four_of_clubs)  
print(str_card)           # 梅花4 (S)  
  
repr_card = repr(four_of_clubs)  
print(repr_card)          # 梅花4 (R)
```

此外，如果定义了，我们可以直接调用这些方法（尽管看起来有些不清楚且不必要）：

```
print(four_of_clubs.__str__())      # 梅花4 (S)  
print(four_of_clubs.__repr__())     # 梅花4 (R)
```

关于那些重复的函数.....

Python开发者意识到，如果你想让`str()`和`repr()`返回相同的字符串，你可能需要功能上重复方法——这是没人喜欢的事情。

因此，有一种机制可以消除这种需求。我之前偷偷带你了解过。事实证明，如果一个类实现了`__repr__`方法但没有实现`__str__`方法，当你将该类的实例传递给`str()`（无论是隐式还是显式），Python会回退使用你的`__repr__`实现。

所以，为了明确起见，考虑以下版本的Card类：

```
类 Card:  
special_names = {1:'王牌', 11:'杰克', 12:'皇后', 13:'国王'}  
  
定义 __init__(self, 花色, 点数):  
    self.花色 = 花色  
    self.点数 = 点数  
  
def __repr__(self):  
card_name = Card.special_names.get(self.点数, str(self.点数))  
    return "%s 的 %s" % (card_name, self.花色)
```

注意这个版本仅实现了`__repr__`方法。尽管如此，调用`str()`仍然会得到用户友好的版本：

```
print(six_of_hearts)          # 6 of Hearts (隐式转换)  
print(str(six_of_hearts))     # 6 of Hearts (显式转换)
```

调用`repr()`也会得到相同结果：

```
print([six_of_hearts])        #[6 of Hearts] (隐式转换)
```

```
print(my_hand)          # [Ace of Spades (R), 4 of Clubs (R), 6 of Hearts (R)]  
print(ace_of_spades)    # Ace of Spades (S)
```

As was covered, the `__str__` method was called when we passed our Card instance to `print` and the `__repr__` method was called when we passed a list of our instances to `print`.

At this point it's worth pointing out that just as we can explicitly create a string from a custom class instance using `str()` as we did earlier, we can also explicitly create a **string representation** of our class with a built-in function called `repr()`.

For example:

```
str_card = str(four_of_clubs)  
print(str_card)           # 4 of Clubs (S)  
  
repr_card = repr(four_of_clubs)  
print(repr_card)          # 4 of Clubs (R)
```

And additionally, if defined, we could call the methods directly (although it seems a bit unclear and unnecessary):

```
print(four_of_clubs.__str__())      # 4 of Clubs (S)  
print(four_of_clubs.__repr__())     # 4 of Clubs (R)
```

#### About those duplicated functions...

Python developers realized, in the case you wanted identical strings to be returned from `str()` and `repr()` you might have to functionally-duplicate methods -- something nobody likes.

So instead, there is a mechanism in place to eliminate the need for that. One I snuck you past up to this point. It turns out that if a class implements the `__repr__` method but not the `__str__` method, and you pass an instance of that class to `str()` (whether implicitly or explicitly), Python will fallback on your `__repr__` implementation and use that.

So, to be clear, consider the following version of the Card class:

```
class Card:  
special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}  
  
def __init__(self, suit, pips):  
    self.suit = suit  
    self.pips = pips  
  
def __repr__(self):  
card_name = Card.special_names.get(self.pips, str(self.pips))  
    return "%s of %s" % (card_name, self.suit)
```

Note this version only implements the `__repr__` method. Nonetheless, calls to `str()` result in the user-friendly version:

```
print(six_of_hearts)          # 6 of Hearts (implicit conversion)  
print(str(six_of_hearts))     # 6 of Hearts (explicit conversion)
```

as do calls to `repr()`:

```
print([six_of_hearts])        #[6 of Hearts] (implicit conversion)
```

```
print(repr(six_of_hearts)) # 6 of Hearts (显式转换)
```

## 总结

为了让你的类实例能够以用户友好的方式“展示自己”，你需要考虑至少实现类的`__repr__`方法。如果没记错的话，雷蒙德·赫廷格（Raymond Hettinger）在一次演讲中提到，确保类实现`__repr__`是他在进行Python代码审查时首先关注的事项之一，现在应该明白原因了。与默认提供的简陋且常常不太有用的信息相比，通过一个简单的方法你可以为调试语句、崩溃报告或日志文件添加的信息量是巨大的。

如果你想要不同的表示形式，例如在容器内部时，你需要实现两个方法`__repr__` 和 `__str__` 方法。（下面会详细介绍这两个方法的不同用法）。

## 第102.2节：实现了两种方法，eval-回环风格的 `__repr__()`

```
类 Card:  
special_names = {1:'王牌', 11:'杰克', 12:'皇后', 13:'国王'}
```

```
定义 __init__(self, 花色, 点数):  
    self.花色 = 花色  
    self.点数 = 点数
```

```
# 当实例通过 str() 转换为字符串时调用  
# 示例：  
#     print(card1)  
#     print(str(card1))  
def __str__(self):  
    card_name = Card.special_names.get(self.pips, str(self.pips))  
    return "%s 的 %s" % (card_name, self.suit)
```

```
# 当实例通过 repr() 转换为字符串时调用  
# 示例：  
#     print([card1, card2, card3])  
#     print(repr(card1))  
def __repr__(self):  
    return "Card(%s, %d)" % (self.suit, self.pips)
```

```
print(repr(six_of_hearts)) # 6 of Hearts (explicit conversion)
```

## Summary

In order for you to empower your class instances to "show themselves" in user-friendly ways, you'll want to consider implementing at least your class's `__repr__` method. If memory serves, during a talk Raymond Hettinger said that ensuring classes implement `__repr__` is one of the first things he looks for while doing Python code reviews, and by now it should be clear why. The amount of information you *could* have added to debugging statements, crash reports, or log files with a simple method is overwhelming when compared to the paltry, and often less-than-helpful (type, id) information that is given by default.

If you want *different* representations for when, for example, inside a container, you'll want to implement both `__repr__` and `__str__` methods. (More on how you might use these two methods differently below).

## Section 102.2: Both methods implemented, eval-round-trip style `__repr__()`

```
class Card:  
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}  
  
    def __init__(self, suit, pips):  
        self.suit = suit  
        self.pips = pips  
  
    # Called when instance is converted to a string via str()  
    # Examples:  
    #     print(card1)  
    #     print(str(card1))  
    def __str__(self):  
        card_name = Card.special_names.get(self.pips, str(self.pips))  
        return "%s of %s" % (card_name, self.suit)  
  
    # Called when instance is converted to a string via repr()  
    # Examples:  
    #     print([card1, card2, card3])  
    #     print(repr(card1))  
    def __repr__(self):  
        return "Card(%s, %d)" % (self.suit, self.pips)
```

# 第103章：调试

## 第103.1节：通过IPython和ipdb

如果安装了IPython（或Jupyter），可以使用以下方式调用调试器：

```
import ipdb  
ipdb.set_trace()
```

当执行到这里时，代码将退出并打印：

```
/home/usr/ook.py(3)<module>()  
 1 import ipdb  
 2 ipdb.set_trace()  
----> 3 print("Hello world!")  
  
ipdb>
```

显然，这意味着必须编辑代码。还有一种更简单的方法：

```
from IPython.core import ultratb  
sys.excepthook = ultratb.FormattedTB(mode='Verbose',  
                                      color_scheme='Linux',  
                                      call_pdb=1)
```

如果出现未捕获的异常，这将导致调用调试器。

## 第103.2节：Python调试器：使用\_pdb\_进行逐步调试

Python标准库中包含一个名为pdb的交互式调试库。pdb功能强大，最常用的是能够“逐步执行”程序。

要立即进入逐步调试，请使用：

```
python -m pdb <my_file.py>
```

这将从程序的第一行开始启动调试器。

通常你会想针对代码的特定部分进行调试。为此，我们导入pdb库并使用set\_trace()来中断这个有问题的示例代码的执行流程。

```
import pdb  
  
def divide(a, b):  
    pdb.set_trace()  
    return a/b  
    # 这有什么问题？提示：2 != 3  
  
print divide(1, 2)
```

运行此程序将启动交互式调试器。

```
python foo.py  
> ~/scratch/foo.py(5)divide()
```

# Chapter 103: Debugging

## Section 103.1: Via IPython and ipdb

If IPython (or Jupyter) are installed, the debugger can be invoked using:

```
import ipdb  
ipdb.set_trace()
```

When reached, the code will exit and print:

```
/home/usr/ook.py(3)<module>()  
 1 import ipdb  
 2 ipdb.set_trace()  
----> 3 print("Hello world!")  
  
ipdb>
```

Clearly, this means that one has to edit the code. There is a simpler way:

```
from IPython.core import ultratb  
sys.excepthook = ultratb.FormattedTB(mode='Verbose',  
                                      color_scheme='Linux',  
                                      call_pdb=1)
```

This will cause the debugger to be called if there is an uncaught exception raised.

## Section 103.2: The Python Debugger: Step-through Debugging with \_pdb\_

The Python Standard Library includes an interactive debugging library called `pdb`. `pdb` has extensive capabilities, the most commonly used being the ability to 'step-through' a program.

To immediately enter into step-through debugging use:

```
python -m pdb <my_file.py>
```

This will start the debugger at the first line of the program.

Usually you will want to target a specific section of the code for debugging. To do this we import the `pdb` library and use `set_trace()` to interrupt the flow of this troubled example code.

```
import pdb  
  
def divide(a, b):  
    pdb.set_trace()  
    return a/b  
    # What's wrong with this? Hint: 2 != 3  
  
print divide(1, 2)
```

Running this program will launch the interactive debugger.

```
python foo.py  
> ~/scratch/foo.py(5)divide()
```

```
-> return a/b  
(Pdb)
```

通常此命令在一行中使用，因此可以用单个#字符注释掉

```
import pdf; pdb.set_trace()
```

在(Pdb)提示符下可以输入命令。这些命令可以是调试器命令或Python命令。要打印变量，我们可以使用调试器的p，或Python的print。

```
(Pdb) p a  
1  
p d b 打印 a  
1
```

要查看所有局部变量的列表，请使用

```
locals
```

内置函数

这些都是值得了解的调试命令：

```
b | : 在第 *n* 行或名为 *f* 的函数处设置断点。  
# b 3  
# b divide  
b: 显示所有断点。  
c: 继续执行直到下一个断点。  
s: 单步执行该行（将进入函数）。  
n: 跳过该行（跳过函数）。  
r: 继续执行直到当前函数返回。  
l: 列出该行附近的代码窗口。  
p : 打印名为 *var* 的变量。  
# p x  
q: 退出调试器。  
bt: 打印当前执行调用栈的回溯  
up: 将作用域向上移动到当前函数的调用者  
down: 将作用域向下移动一个调用栈层级  
step: 运行程序直到下一条执行语句，然后将控制权返回给调试器  
  
next: 运行程序直到当前函数的下一条执行语句，然后将控制权返回给调试器  
return: 运行程序直到当前函数返回，然后将控制权返回给调试器  
continue: 继续运行程序直到下一个断点（或再次调用 set_trace）
```

调试器还可以交互式地评估 Python 代码：

```
>>> return a/b  
(Pdb) p a+b  
  
(Pdb) [ str(m) for m in [a,b]]  
['1', '2']  
(Pdb) [ d for d in xrange(5)]  
[0, 1, 2, 3, 4]
```

注意：

```
-> return a/b  
(Pdb)
```

Often this command is used on one line so it can be commented out with a single # character

```
import pdf; pdb.set_trace()
```

At the (Pdb) prompt commands can be entered. These commands can be debugger commands or python. To print variables we can use p from the debugger, or python's print.

```
(Pdb) p a  
1  
(Pdb) print a  
1
```

To see list of all local variables use

```
locals
```

build-in function

These are good debugger commands to know:

```
b | : set breakpoint at line *n* or function named *f*.  
# b 3  
# b divide  
b: show all breakpoints.  
c: continue until the next breakpoint.  
s: step through this line (will enter a function).  
n: step over this line (jumps over a function).  
r: continue until the current function returns.  
l: list a window of code around this line.  
p : print variable named *var*.  
# p x  
q: quit debugger.  
bt: print the traceback of the current execution call stack  
up: move your scope up the function call stack to the caller of the current function  
down: Move your scope back down the function call stack one level  
step: Run the program until the next line of execution in the program, then return control back to the debugger  
next: run the program until the next line of execution in the current function, then return control back to the debugger  
return: run the program until the current function returns, then return control back to the debugger  
continue: continue running the program until the next breakpoint (or set_trace si called again)
```

The debugger can also evaluate python interactively:

```
-> return a/b  
(Pdb) p a+b  
3  
(Pdb) [ str(m) for m in [a,b]]  
['1', '2']  
(Pdb) [ d for d in xrange(5)]  
[0, 1, 2, 3, 4]
```

Note:

如果你的变量名与调试器命令重合，使用感叹号 '!' 放在变量名前，明确指代变量而非调试器命令。例如，通常你可能使用变量名 'c' 作为计数器，并且想在调试器中打印它。简单的 'c' 命令会继续执行直到下一个断点。此时应使用 '!c' 来打印变量的值，如下所示：

```
(Pdb) !c  
4
```

## 第103.3节：远程调试器

有时你需要调试由另一个进程执行的Python代码，这时 rpdb 就派上用场了。

rpdb是pdb的一个封装，它将标准输入和标准输出重定向到一个套接字处理器。默认情况下，它在端口4444上打开调试器

用法：

```
# 在你想调试的Python文件中。  
import rpdb  
rpdb.set_trace()
```

然后你需要在终端运行以下命令以连接到该进程。

```
# 在终端调用以查看输出  
$ nc 127.0.0.1 4444
```

你将会看到pdb提示符

```
> /home/usr/ook.py(3)<module>()  
-&gt; print("Hello world!")  
(Pdb)
```

If any of your variable names coincide with the debugger commands, use an exclamation mark '!' before the var to explicitly refer to the variable and not the debugger command. For example, often it might so happen that you use the variable name 'c' for a counter, and you might want to print it while in the debugger. a simple 'c' command would continue execution till the next breakpoint. Instead use '!c' to print the value of the variable as follows:

```
(Pdb) !c  
4
```

## Section 103.3: Remote debugger

Sometimes you need to debug python code which is executed by another process and in this cases [rpdb](#) comes in handy.

rpdb is a wrapper around pdb that re-routes stdin and stdout to a socket handler. By default it opens the debugger on port 4444

Usage:

```
# In the Python file you want to debug.  
import rpdb  
rpdb.set_trace()
```

And then you need run this in terminal to connect to this process.

```
# Call in a terminal to see the output  
$ nc 127.0.0.1 4444
```

And you will get pdb prompt

```
> /home/usr/ook.py(3)<module>()  
-> print("Hello world!")  
(Pdb)
```

# 第104章：读取和写入CSV

## 第104.1节：使用pandas

从字典或DataFrame写入CSV文件。

```
import pandas as pd

d = {'a': (1, 101), 'b': (2, 202), 'c': (3, 303)}
pd.DataFrame.from_dict(d, orient="index")
df.to_csv("data.csv")
```

将CSV文件读取为DataFrame并转换为字典：

```
df = pd.read_csv("data.csv")
d = df.to_dict()
```

## 第104.2节：写入TSV文件

### Python

```
import csv

with open('/tmp/output.tsv', 'wt') as out_file:
    tsv_writer = csv.writer(out_file, delimiter="\t")
    tsv_writer.writerow(['姓名', '领域'])
    tsv_writer.writerow(['迪杰斯特拉', '计算机科学'])
    tsv_writer.writerow(['谢拉', '数学'])
    tsv_writer.writerow(['奥曼', '经济科学'])
```

### 输出文件

```
$ cat /tmp/output.tsv
```

姓名	领域
迪杰斯特拉	计算机科学
谢拉	数学
奥曼	经济科学

# Chapter 104: Reading and Writing CSV

## Section 104.1: Using pandas

Write a CSV file from a `dict` or a DataFrame.

```
import pandas as pd

d = {'a': (1, 101), 'b': (2, 202), 'c': (3, 303)}
pd.DataFrame.from_dict(d, orient="index")
df.to_csv("data.csv")
```

Read a CSV file as a DataFrame and convert it to a `dict`:

```
df = pd.read_csv("data.csv")
d = df.to_dict()
```

## Section 104.2: Writing a TSV file

### Python

```
import csv

with open('/tmp/output.tsv', 'wt') as out_file:
    tsv_writer = csv.writer(out_file, delimiter='\t')
    tsv_writer.writerow(['name', 'field'])
    tsv_writer.writerow(['Dijkstra', 'Computer Science'])
    tsv_writer.writerow(['Shelah', 'Math'])
    tsv_writer.writerow(['Aumann', 'Economic Sciences'])
```

### Output file

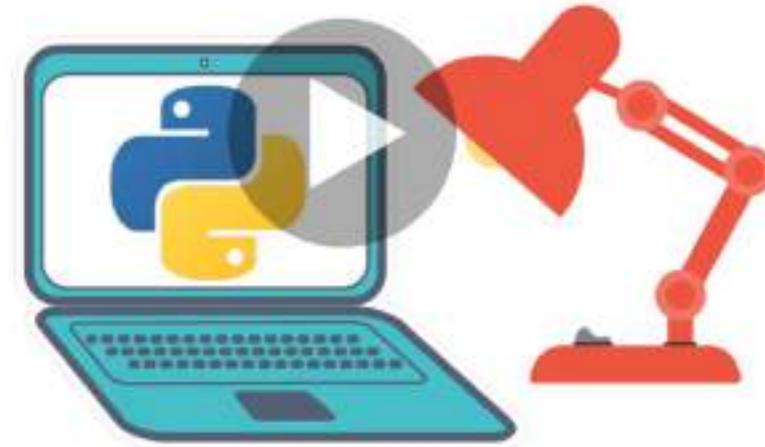
```
$ cat /tmp/output.tsv
```

name	field
Dijkstra	Computer Science
Shelah	Math
Aumann	Economic Sciences

# 视频：完整的Python训练营：从零开始成为Python 3高手

像专业人士一样学习Python！从基础开始，直到创建你自己的应用程序和游戏！

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ 学习专业使用Python，掌握Python 2和Python 3！
- ✓ 用Python创建游戏，比如井字棋和二十一点！
- ✓ 学习高级Python功能，如collections模块和时间戳的使用！
- ✓ 学习使用面向对象编程和类！
- ✓ 理解复杂主题，如装饰器。
- ✓ 理解如何使用Jupyter Notebook并创建.py文件
- ✓ 了解如何在Jupyter Notebook系统中创建图形用户界面（GUI）！
- ✓ 从零开始全面掌握Python！

今天观看 →

# VIDEO: Complete Python Bootcamp: Go from zero to hero in Python 3

Learn Python like a Professional! Start from the basics and go all the way to creating your own applications and games!

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ Learn to use Python professionally, learning both Python 2 and Python 3!
- ✓ Create games with Python, like Tic Tac Toe and Blackjack!
- ✓ Learn advanced Python features, like the collections module and how to work with timestamps!
- ✓ Learn to use Object Oriented Programming with classes!
- ✓ Understand complex topics, like decorators.
- ✓ Understand how to use both the Jupyter Notebook and create .py files
- ✓ Get an understanding of how to create GUIs in the Jupyter Notebook system!
- ✓ Build a complete understanding of Python from the ground up!

Watch Today →

# 第105章：从字符串或列表写入CSV

参数	详细信息
<code>open ("/path/", "mode")</code>	指定CSV文件的路径
<code>open (path, "mode")</code>	指定打开文件的模式（读取、写入等）
<code>csv.writer(file, delimiter)</code>	传入已打开的CSV文件
<code>csv.writer(file, delimiter=' ')</code>	指定分隔符字符或模式

写入.csv文件在大多数方面与写入普通文件类似，且相当简单。我将尽我所能，介绍解决该问题最简单且最高效的方法。

## 第105.1节：基本写入示例

```
import csv

#----- 我们将在此函数中写入CSV -----
def csv_writer(data, path):

    #打开我们传入路径的CSV文件。
    with open(path, "wb") as csv_file:

        writer = csv.writer(csv_file, delimiter=',')
        for line in data:
            writer.writerow(line)

#---- 在这里定义我们的列表，并调用函数 -----

if __name__ == "__main__":
    """
    data = 我们想要写入的列表。
    将其拆分成列表的列表。
    """
    data = ["first_name,last_name,age".split(","),
            "John,Doe,22".split(","),
            "Jane,Doe,31".split(","),
            "Jack,Reacher,27".split(",")]
    """

    # 我们想写入的CSV文件路径。
    path = "output.csv"
    csv_writer(data, path)
```

## 第105.2节：在CSV文件中追加字符串作为换行符

```
def append_to_csv(input_string):
    with open("fileName.csv", "a") as csv_file:
        csv_file.write(input_row + "")
```

# Chapter 105: Writing to CSV from String or List

Parameter	Details
<code>open ("/path/", "mode")</code>	Specify the path to your CSV file
<code>open (path, "mode")</code>	Specify mode to open file in (read, write, etc.)
<code>csv.writer(file, delimiter)</code>	Pass opened CSV file here
<code>csv.writer(file, delimiter=' ')</code>	Specify delimiter character or pattern

Writing to a .csv file is not unlike writing to a regular file in most regards, and is fairly straightforward. I will, to the best of my ability, cover the easiest, and most efficient approach to the problem.

## Section 105.1: Basic Write Example

```
import csv

#----- We will write to CSV in this function -----

def csv_writer(data, path):

    #Open CSV file whose path we passed.
    with open(path, "wb") as csv_file:

        writer = csv.writer(csv_file, delimiter=',')
        for line in data:
            writer.writerow(line)

#---- Define our list here, and call function -----

if __name__ == "__main__":
    """
    data = our list that we want to write.
    Split it so we get a list of lists.
    """
    data = ["first_name,last_name,age".split(","),
            "John,Doe,22".split(","),
            "Jane,Doe,31".split(","),
            "Jack,Reacher,27".split(",")]
    """

    # Path to CSV file we want to write to.
    path = "output.csv"
    csv_writer(data, path)
```

## Section 105.2: Appending a String as a newline in a CSV file

```
def append_to_csv(input_string):
    with open("fileName.csv", "a") as csv_file:
        csv_file.write(input_row + "\n")
```

# 第106章：使用`exec`和`eval`进行动态代码执行

## 参数

	详细信息
expression	表达式代码，作为字符串，或一个 <code>code</code> 对象
对象	语句代码，作为字符串，或一个 <code>code</code> 对象
全局变量	用于全局变量的字典。如果未指定 <code>locals</code> ，则该字典也用于 <code>locals</code> 。如果省略，则使用调用作用域的 <code>globals()</code> 。
locals	用于局部变量的 <code>mapping</code> 对象。如果省略，则使用传入的 <code>globals</code> 对象。若两者均省略，则调用作用域的 <code>globals()</code> 和 <code>locals()</code> 分别用于 <code>globals</code> 和 <code>locals</code> 。

## 第106.1节：使用 `exec`、`eval` 或 `ast.literal_eval` 执行不受信任用户提供的代码

无法安全地使用 `eval` 或 `exec` 来执行来自不受信任用户的代码。即使是 `ast.literal_eval` 也容易导致解析器崩溃。有时可以防范恶意代码执行，但无法排除解析器或词法分析器完全崩溃的可能性。

要评估不受信任用户的代码，您需要使用某些第三方模块，或者可能需要用 Python 编写自己的解析器和虚拟机。

## 第106.2节：使用 `ast.literal_eval` 评估包含 Python 字面量的字符串

如果你有一个包含 Python 字面量（如字符串、浮点数等）的字符串，可以使用 `ast.literal_eval` 来评估其值，而不是使用 `eval`。这样做的额外好处是只允许某些语法。

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

然而，对于不受信任用户提供的代码执行，这并不安全，并且通过精心构造的输入很容易使解释器崩溃

```
>>> import ast
>>> ast.literal_eval('()' * 1000000)
[5]    21358 segmentation fault (core dumped) python3
```

这里，输入是一个字符串()重复一百万次，导致 CPython 解析器崩溃。CPython 开发者不将解析器中的错误视为安全问题。

## 第106.3节：使用 `exec` 评估语句

```
>>> code = """for i in range(5):    print('Hello world!')"""
>>> exec(code)
你好, 世界!
你好, 世界!
你好, 世界!
```

# Chapter 106: Dynamic code execution with `exec` and `eval`

## Argument

`expression` The expression code as a string, or a `code` object

`object` The statement code as a string, or a `code` object

`globals` The dictionary to use for global variables. If `locals` is not specified, this is also used for `locals`. If omitted, the `globals()` of calling scope are used.

`locals` A `mapping` object that is used for local variables. If omitted, the one passed for `globals` is used instead. If both are omitted, then the `globals()` and `locals()` of the calling scope are used for `globals` and `locals` respectively.

## Details

### Section 106.1: Executing code provided by untrusted user using `exec`, `eval`, or `ast.literal_eval`

**It is not possible to use `eval` or `exec` to execute code from untrusted user securely.** Even `ast.literal_eval` is prone to crashes in the parser. It is sometimes possible to guard against malicious code execution, but it doesn't exclude the possibility of outright crashes in the parser or the tokenizer.

To evaluate code by an untrusted user you need to turn to some third-party module, or perhaps write your own parser and your own virtual machine in Python.

### Section 106.2: Evaluating a string containing a Python literal with `ast.literal_eval`

If you have a string that contains Python literals, such as strings, floats etc, you can use `ast.literal_eval` to evaluate its value instead of `eval`. This has the added feature of allowing only certain syntax.

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

**However, this is not secure for execution of code provided by untrusted user, and it is trivial to crash an interpreter with carefully crafted input**

```
>>> import ast
>>> ast.literal_eval('()' * 1000000)
[5]    21358 segmentation fault (core dumped) python3
```

Here, the input is a string of () repeated one million times, which causes a crash in CPython parser. CPython developers do not consider bugs in parser as security issues.

### Section 106.3: Evaluating statements with `exec`

```
>>> code = """for i in range(5):\n    print('Hello world!')"""
>>> exec(code)
Hello world!
Hello world!
Hello world!
```

你好，世界！  
你好，世界！

## 第106.4节：使用eval评估表达式

```
>>> 表达式 = '5 + 3 * a'  
>>> a = 5  
>>> 结果 = eval(表达式)  
>>> 结果  
20
```

## 第106.5节：预编译表达式以多次评估

内置函数compile可用于将表达式预编译为代码对象；然后该代码对象可以传递给eval。这将加快对评估代码的重复执行。compile的第三个参数需要是字符串'eval'。

```
>>> 代码 = compile('a * b + c', '<string>', 'eval')  
>>> 代码  
<代码 对象 <模块> 位于 0x7f0e51a58830, 文件 "<string>", 第 1 行>  
>>> a, b, c = 1, 2, 3  
>>> eval(代码)  
5
```

## 第106.6节：使用自定义全局变量通过eval评估表达式

```
>>> 变量 = {'a': 6, 'b': 7}  
>>> 计算('a * b', 全局变量=变量)  
42
```

另外，使用这个方法代码就不会意外引用外部定义的名称：

```
>>> eval('variables')  
{'a': 6, 'b': 7}  
>>> eval('variables', globals=variables)  
追踪 (最近的调用最后) :  
    文件 "<stdin>", 第 1 行, 在 <模块>  
    文件 "<string>", 第 1 行, 在 <模块>  
NameError: 名称 'variables' 未定义
```

使用defaultdict 例如可以让未定义的变量默认设置为零：

```
>>> from collections import defaultdict  
>>> 变量 = defaultdict(int, {'a': 42})  
>>> eval('a * c', globals=变量) # 注意 'c' 未被明确定义  
0
```

Hello world!  
Hello world!

## Section 106.4: Evaluating an expression with eval

```
>>> expression = '5 + 3 * a'  
>>> a = 5  
>>> result = eval(expression)  
>>> result  
20
```

## Section 106.5: Precompiling an expression to evaluate it multiple times

compile built-in function can be used to precompile an expression to a code object; this code object can then be passed to eval. This will speed up the repeated executions of the evaluated code. The 3rd parameter to compile needs to be the string 'eval'.

```
>>> code = compile('a * b + c', '<string>', 'eval')  
>>> code  
<code object <module> at 0x7f0e51a58830, file "<string>", line 1>  
>>> a, b, c = 1, 2, 3  
>>> eval(code)  
5
```

## Section 106.6: Evaluating an expression with eval using custom globals

```
>>> variables = {'a': 6, 'b': 7}  
>>> eval('a * b', globals=variables)  
42
```

As a plus, with this the code cannot accidentally refer to the names defined outside:

```
>>> eval('variables')  
{'a': 6, 'b': 7}  
>>> eval('variables', globals=variables)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<string>", line 1, in <module>  
NameError: name 'variables' is not defined
```

Using defaultdict allows for example having undefined variables set to zero:

```
>>> from collections import defaultdict  
>>> variables = defaultdict(int, {'a': 42})  
>>> eval('a * c', globals=variables) # note that 'c' is not explicitly defined  
0
```

# 第107章：PyInstaller - 分发Python代码

## 第107.1节：安装与设置

PyInstaller是一个普通的Python包。可以使用pip安装：

```
pip install pyinstaller
```

### Windows上的安装

对于Windows, [pywin32](#)或[pypiwin32](#)是前提条件。后者在使用pip安装pyinstaller时会自动安装。

### Mac OS X上的安装

PyInstaller兼容当前Mac OS X自带的默认Python 2.7。如果要使用更高版本的Python, 或者使用PyQT、Numpy、Matplotlib等主要包, 建议使用MacPorts或Homebrew进行安装。

### 从归档安装

如果没有 pip, 可从PyPI下载压缩包。

要测试开发版本, 请从PyInstaller

[Downloads](#)页面的develop分支下载压缩包。

解压缩包并找到setup.py脚本。以管理员权限执行python setup.py install来安装或升级 PyInstaller。

### 验证安装

安装成功后, 所有平台的系统路径中都应存在pyinstaller命令。

通过在命令行输入pyinstaller --version来验证。这将显示当前的 pyinstaller 版本。

## 第107.2节：使用 Pyinstaller

在最简单的用例中, 只需进入文件所在目录, 输入：

```
pyinstaller myfile.py
```

Pyinstaller 会分析该文件并创建：

- 与myfile.py同一目录下的myfile.spec文件
- 与myfile.py同一目录下的build文件夹
- 与myfile.py同一目录下的dist文件夹
- build文件夹中的日志文件

打包后的应用程序可以在 dist 文件夹中找到

### 选项

PyInstaller 有多个可用选项。完整的选项列表可以在 [here](#) 找到。

打包完成后, 可以通过打开 'dist\myfile\myfile.exe' 来运行您的应用程序。

# Chapter 107: PyInstaller - Distributing Python Code

## Section 107.1: Installation and Setup

Pyinstaller is a normal python package. It can be installed using pip:

```
pip install pyinstaller
```

### Installation in Windows

For Windows, [pywin32](#) or [pypiwin32](#) is a prerequisite. The latter is installed automatically when pyinstaller is installed using pip.

### Installation in Mac OS X

PyInstaller works with the default Python 2.7 provided with current Mac OS X. If later versions of Python are to be used or if any major packages such as PyQt, Numpy, Matplotlib and the like are to be used, it is recommended to install them using either [MacPorts](#) or [Homebrew](#).

### Installing from the archive

If pip is not available, download the compressed archive from [PyPI](#).

To test the development version, download the compressed archive from the *develop* branch of [PyInstaller Downloads](#) page.

Expand the archive and find the setup.py script. Execute python setup.py install with administrator privilege to install or upgrade PyInstaller.

### Verifying the installation

The command pyinstaller should exist on the system path for all platforms after a successful installation.

Verify it by typing pyinstaller --version in the command line. This will print the current version of pyinstaller.

## Section 107.2: Using Pyinstaller

In the simplest use-case, just navigate to the directory your file is in, and type:

```
pyinstaller myfile.py
```

Pyinstaller analyzes the file and creates:

- A **myfile.spec** file in the same directory as myfile.py
- A **build** folder in the same directory as myfile.py
- A **dist** folder in the same directory as myfile.py
- Log files in the **build** folder

The bundled app can be found in the **dist** folder

### Options

There are several options that can be used with pyinstaller. A full list of the options can be found [here](#).

Once bundled your app can be run by opening 'dist\myfile\myfile.exe'.

## 第107.3节：打包到一个文件夹

当 PyInstaller 在不带任何选项的情况下打包 myscript.py 时，默认输出是一个单独的文件夹（名为myscript）, 其中包含一个名为 myscript 的可执行文件（Windows 下为 myscript.exe）以及所有必要的依赖项。

该应用程序可以通过将文件夹压缩成 zip 文件进行分发。

可以使用选项 -D 或 --onedir 明确设置为单文件夹模式

```
pyinstaller myscript.py -D
```

### 优点：

将所有内容打包到单个文件夹的主要优点之一是更容易调试问题。如果有任何模块导入失败，可以通过检查该文件夹来验证。

另一个优点是在更新时体现出来。如果代码有少量更改，但所使用的依赖完全相同，分发者只需发送可执行文件（通常比整个文件夹小）。

### 缺点

这种方法的唯一缺点是用户必须在大量文件中查找可执行文件。

此外，用户可能会删除或修改其他文件，这可能导致应用程序无法正常工作。

## 第107.4节：打包成单个文件

```
pyinstaller myscript.py -F
```

生成单个文件的选项是-F或--onefile。这会将程序打包成单个myscript.exe文件。

单文件可执行文件比单文件夹打包的运行速度更慢，调试也更困难。

## Section 107.3: Bundling to One Folder

When PyInstaller is used without any options to bundle myscript.py , the default output is a single folder (named myscript) containing an executable named myscript (myscript.exe in windows) along with all the necessary dependencies.

The app can be distributed by compressing the folder into a zip file.

One Folder mode can be explicitly set using the option -D or --onedir

```
pyinstaller myscript.py -D
```

### Advantages:

One of the major advantages of bundling to a single folder is that it is easier to debug problems. If any modules fail to import, it can be verified by inspecting the folder.

Another advantage is felt during updates. If there are a few changes in the code but the dependencies used are *exactly* the same, distributors can just ship the executable file (which is typically smaller than the entire folder).

### Disadvantages

The only disadvantage of this method is that the users have to search for the executable among a large number of files.

Also users can delete/modify other files which might lead to the app not being able to work correctly.

## Section 107.4: Bundling to a Single File

```
pyinstaller myscript.py -F
```

The options to generate a single file are -F or --onefile. This bundles the program into a single myscript.exe file.

Single file executable are slower than the one-folder bundle. They are also harder to debug.

# 第108章：使用Python进行数据可视化

## 第108.1节：Seaborn

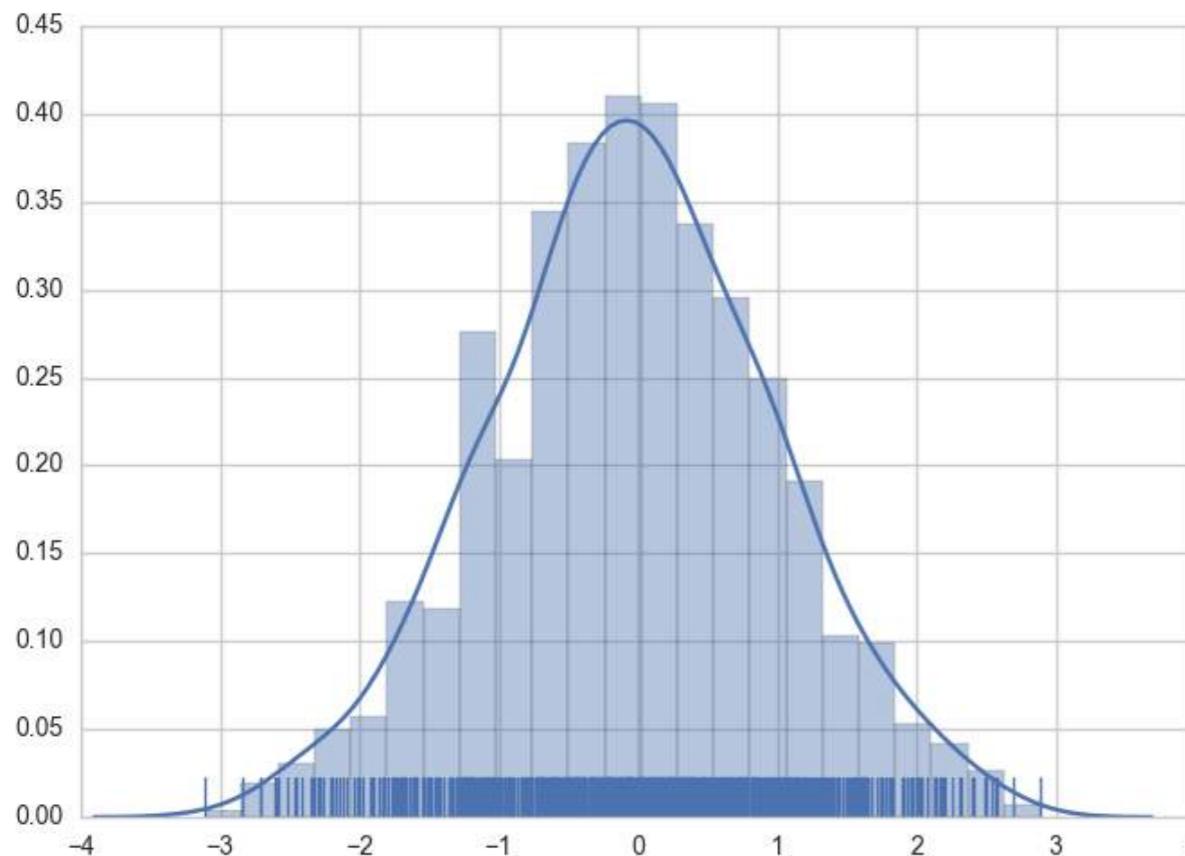
Seaborn 是一个基于Matplotlib的封装，能够轻松创建常见的统计图。支持的图形列表包括单变量和双变量分布图、回归图，以及多种绘制分类变量的方法。Seaborn提供的完整图形列表见其API参考。

在Seaborn中创建图表非常简单，只需调用相应的绘图函数。以下是一个为随机生成数据创建直方图、核密度估计图和rug图的示例。

```
import numpy as np # 使用numpy生成绘图数据
import seaborn as sns # seaborn的常用导入形式

# 生成正态分布数据
data = np.random.randn(1000)

# 绘制带有rugplot和kde图叠加的直方图
sns.distplot(data, kde=True, rug=True)
```



图表的样式也可以通过声明式语法进行控制。

```
# 使用之前导入的库和数据。
# 使用无网格的深色背景。
sns.set_style('dark')
```

# Chapter 108: Data Visualization with Python

## Section 108.1: Seaborn

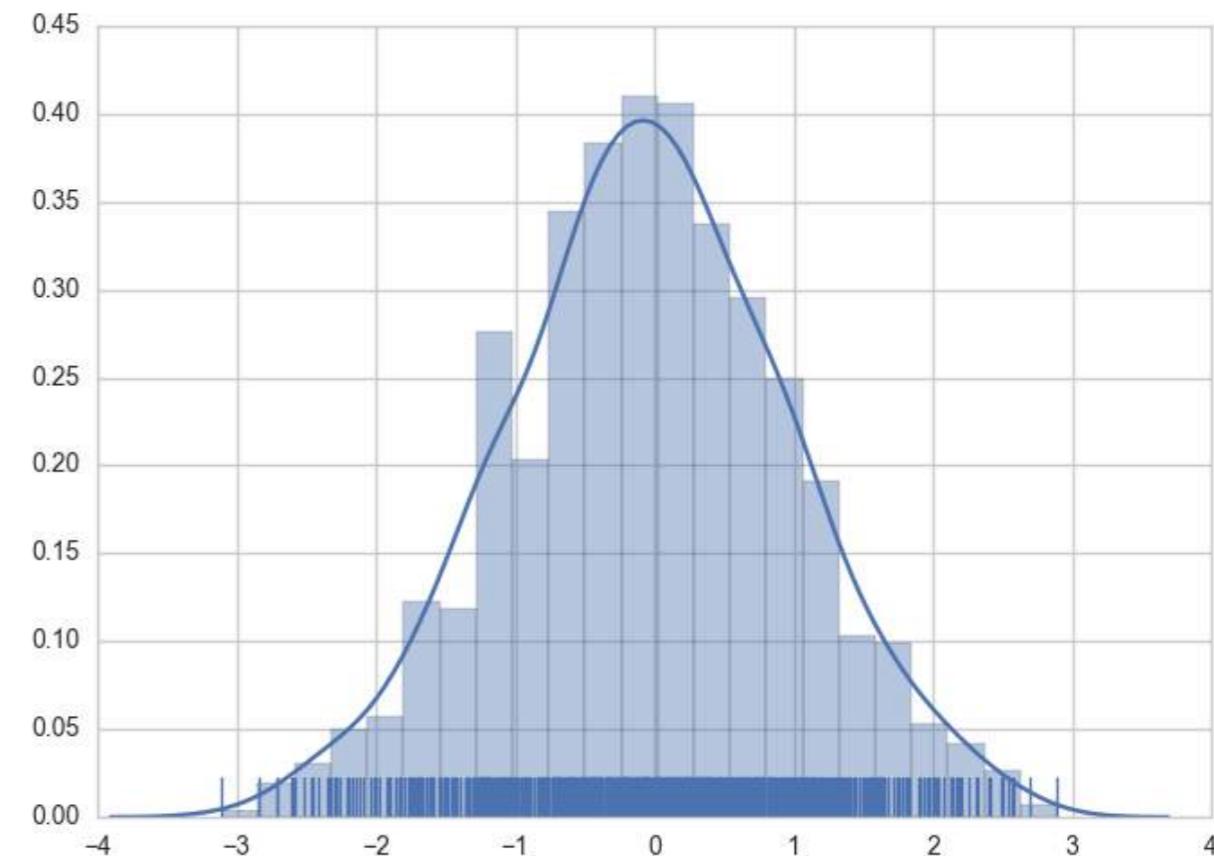
Seaborn is a wrapper around Matplotlib that makes creating common statistical plots easy. The list of supported plots includes univariate and bivariate distribution plots, regression plots, and a number of methods for plotting categorical variables. The full list of plots Seaborn provides is in their [API reference](#).

Creating graphs in Seaborn is as simple as calling the appropriate graphing function. Here is an example of creating a histogram, kernel density estimation, and rug plot for randomly generated data.

```
import numpy as np # numpy used to create data from plotting
import seaborn as sns # common form of importing seaborn

# Generate normally distributed data
data = np.random.randn(1000)

# Plot a histogram with both a rugplot and kde graph superimposed
sns.distplot(data, kde=True, rug=True)
```

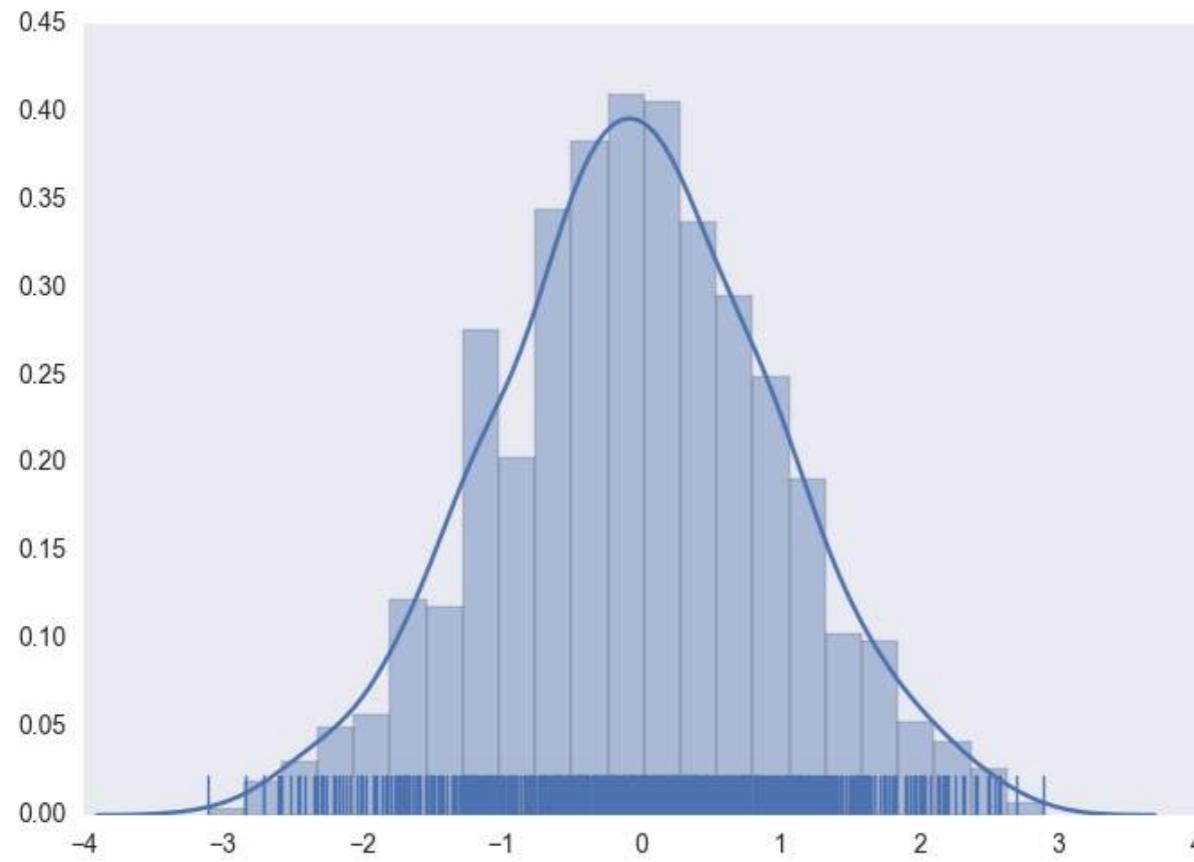


The style of the plot can also be controlled using a declarative syntax.

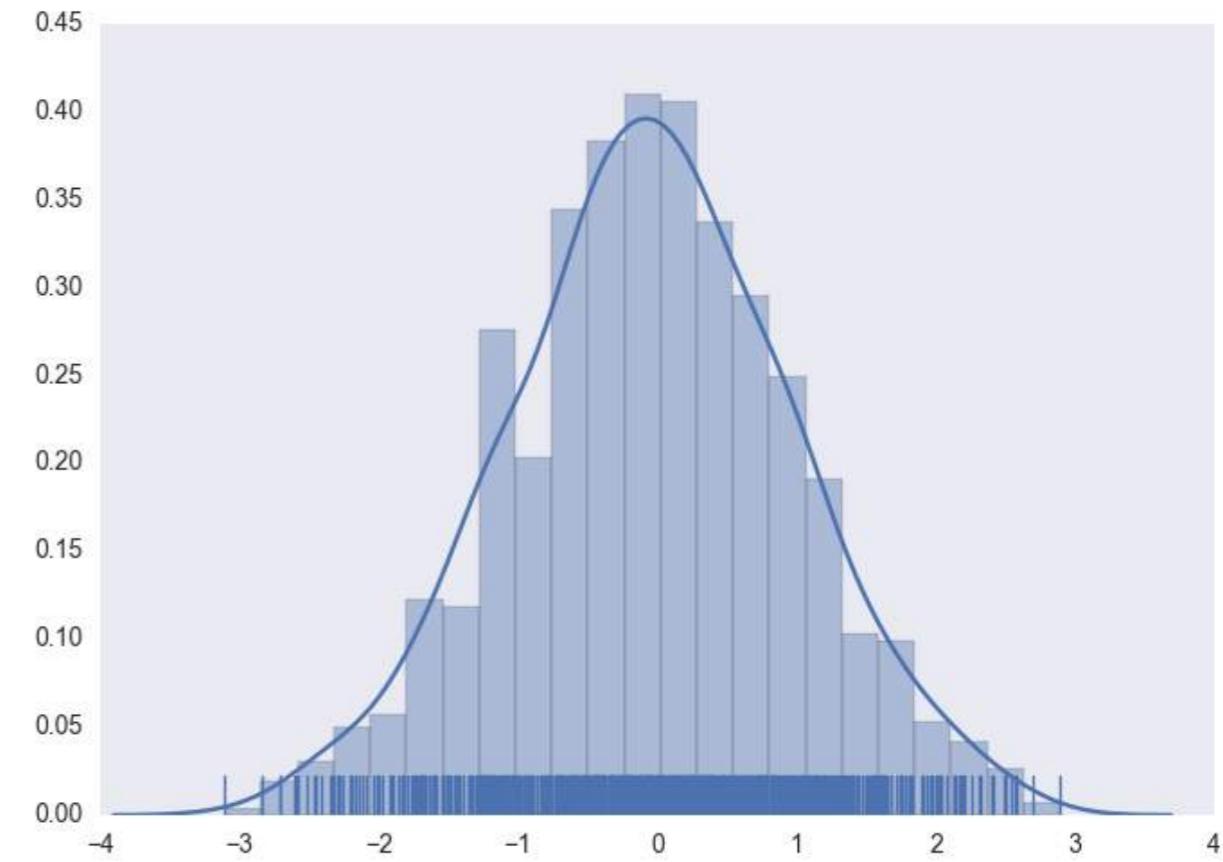
```
# Using previously created imports and data.

# Use a dark background with no grid.
sns.set_style('dark')
```

```
# 重新创建图表  
sns.distplot(data, kde=True, rug=True)
```



```
# Create the plot again  
sns.distplot(data, kde=True, rug=True)
```

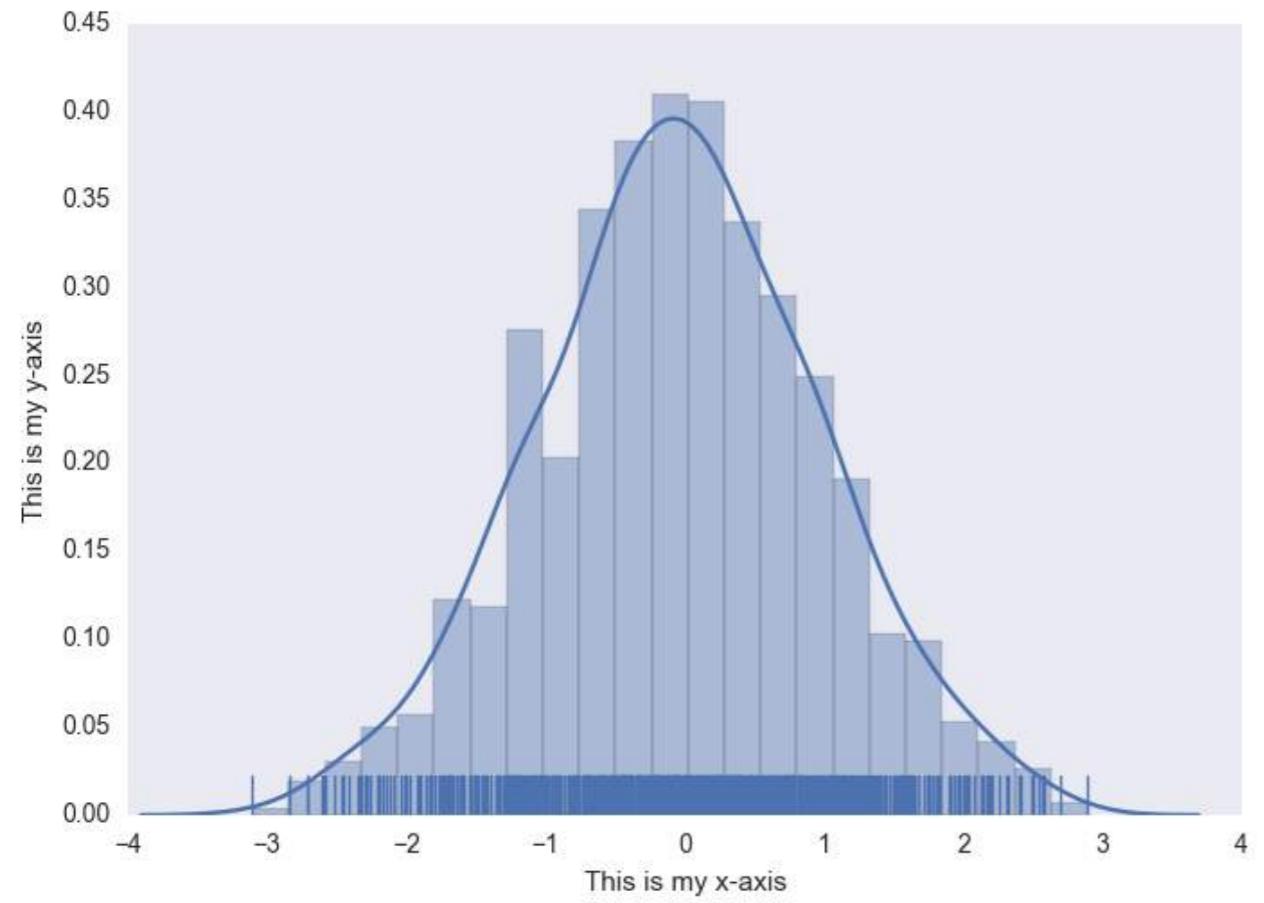


作为额外的好处，普通的matplotlib命令仍然可以应用于Seaborn图表。下面是一个示例，展示如何为我们之前创建的直方图添加坐标轴标题。

```
# 使用之前创建的数据和样式  
  
# 访问matplotlib命令  
import matplotlib.pyplot as plt  
  
# 之前创建的图表。  
sns.distplot(data, kde=True, rug=True)  
# 设置坐标轴标签。  
plt.xlabel('这是我的x轴')  
plt.ylabel('这是我的y轴')
```

As an added bonus, normal matplotlib commands can still be applied to Seaborn plots. Here's an example of adding axis titles to our previously created histogram.

```
# Using previously created data and style  
  
# Access to matplotlib commands  
import matplotlib.pyplot as plt  
  
# Previously created plot.  
sns.distplot(data, kde=True, rug=True)  
# Set the axis labels.  
plt.xlabel('This is my x-axis')  
plt.ylabel('This is my y-axis')
```



## 第108.2节：Matplotlib

[Matplotlib](#) 是一个用于Python的数学绘图库，提供多种不同的绘图功能。

[Matplotlib 文档可以在这里找到，SO 文档也可以在这里查看。](#)

Matplotlib 提供了两种不同的绘图方法，尽管它们在大多数情况下是可以互换的：

- 首先，matplotlib 提供了pyplot接口，这是一个直接且易用的接口，允许以类似 MATLAB 的风格绘制复杂图形。
- 其次，matplotlib 允许用户通过基于对象的系统直接控制不同的方面（坐标轴、线条、刻度等）。这更为复杂，但可以完全控制整个图形。

下面是使用pyplot接口绘制一些生成数据的示例：

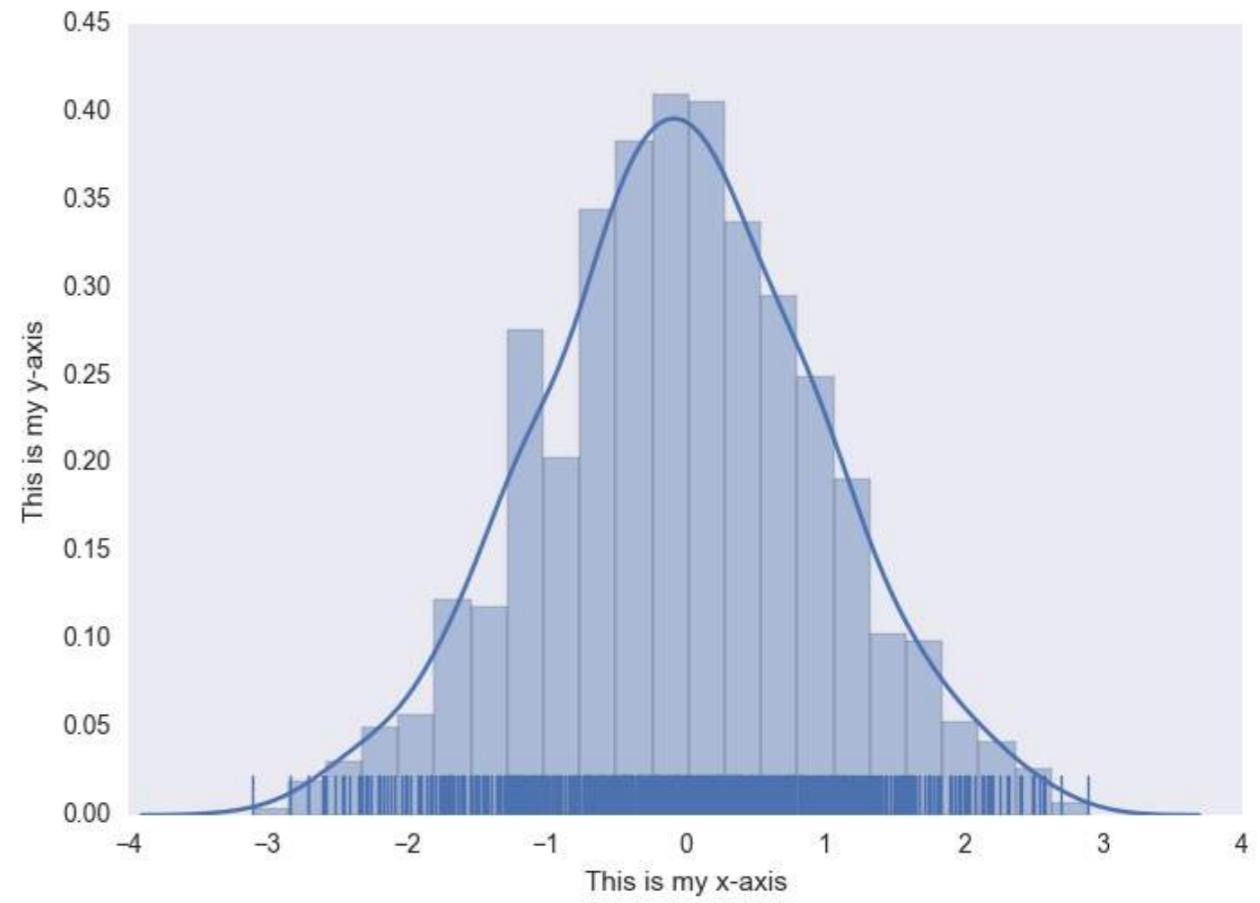
```
import matplotlib.pyplot as plt

# 生成一些用于绘图的数据。
x = [0, 1, 2, 3, 4, 5, 6]
y = [i**2 for i in x]

# 使用一些关键字参数绘制数据 x, y，以控制绘图样式。
# 使用两个不同的绘图命令同时绘制点(散点图)和线(折线图)。

plt.scatter(x, y, c='blue', marker='x', s=100) # 创建蓝色"x"形状、大小为100的标记
plt.plot(x, y, color='red', linewidth=2) # 创建一条红色线，线宽为2。

# 向坐标轴添加一些文本和标题。
plt.xlabel('x data')
```



## Section 108.2: Matplotlib

[Matplotlib](#) is a mathematical plotting library for Python that provides a variety of different plotting functionality.

The [matplotlib documentation can be found here](#), with the SO Docs being available [here](#).

Matplotlib provides two distinct methods for plotting, though they are interchangeable for the most part:

- Firstly, matplotlib provides the pyplot interface, direct and simple-to-use interface that allows plotting of complex graphs in a MATLAB-like style.
- Secondly, matplotlib allows the user to control the different aspects (axes, lines, ticks, etc) directly using an object-based system. This is more difficult but allows complete control over the entire plot.

Below is an example of using the pyplot interface to plot some generated data:

```
import matplotlib.pyplot as plt

# Generate some data for plotting.
x = [0, 1, 2, 3, 4, 5, 6]
y = [i**2 for i in x]

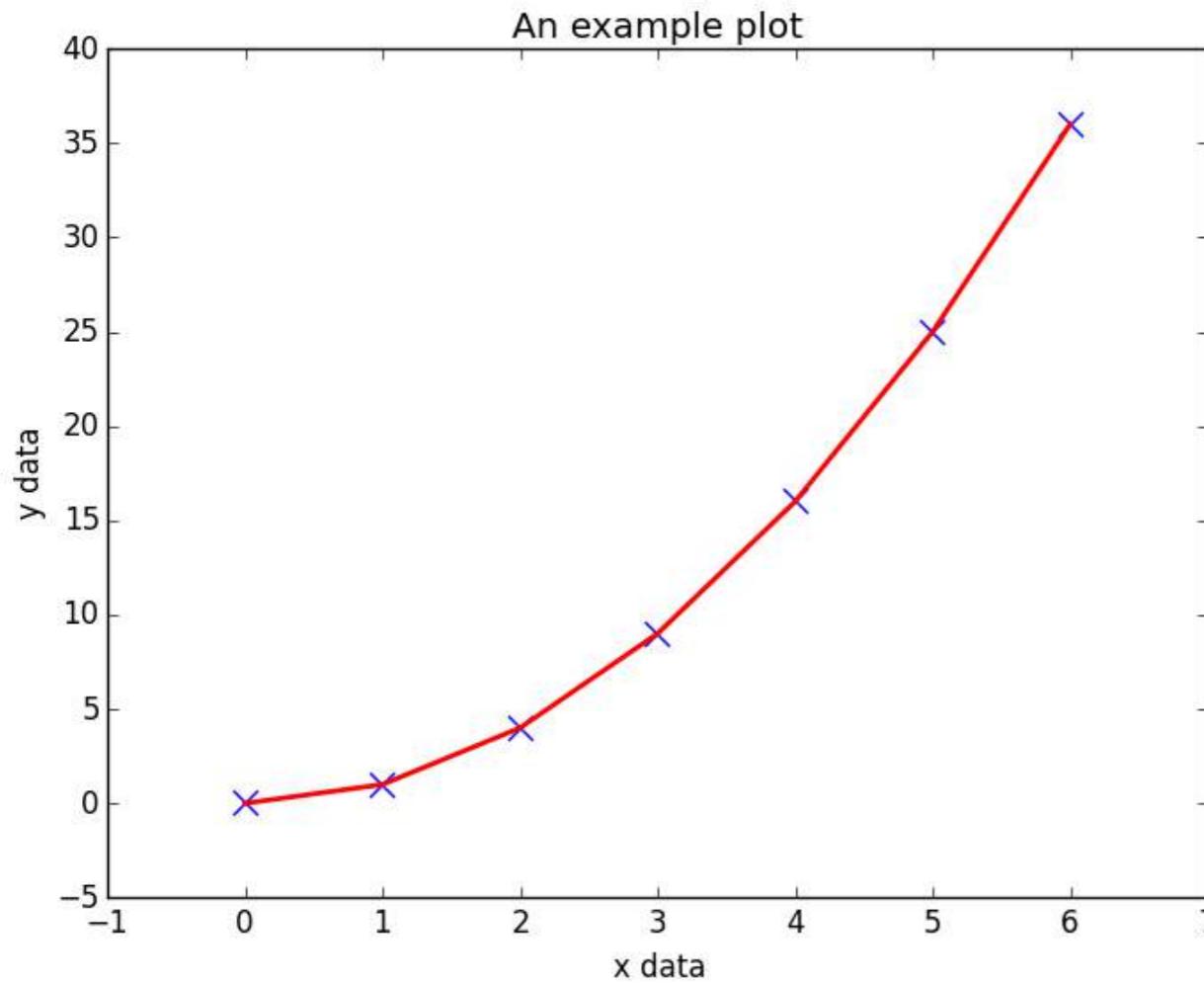
# Plot the data x, y with some keyword arguments that control the plot style.
# Use two different plot commands to plot both points (scatter) and a line (plot).

plt.scatter(x, y, c='blue', marker='x', s=100) # Create blue markers of shape "x" and size 100
plt.plot(x, y, color='red', linewidth=2) # Create a red line with linewidth 2.

# Add some text to the axes and a title.
plt.xlabel('x data')
```

```
plt.ylabel('y data')
plt.title('An example plot')

# 生成图表并显示给用户。
plt.show()
```



注意，`plt.show()`在某些环境中由于以交互模式运行`matplotlib.pyplot`而被认为是有问题的，如果是这样，可以通过传入可选参数`plt.show(block=True)`显式覆盖阻塞行为，以缓解该问题。

### 第108.3节：Plotly

Plotly是一个现代的绘图和数据可视化平台。对于生成各种图表尤其是数据科学方面非常有用，Plotly作为库支持Python、R、JavaScript、Julia和MATLAB。它也可以作为这些语言的网络应用程序使用。

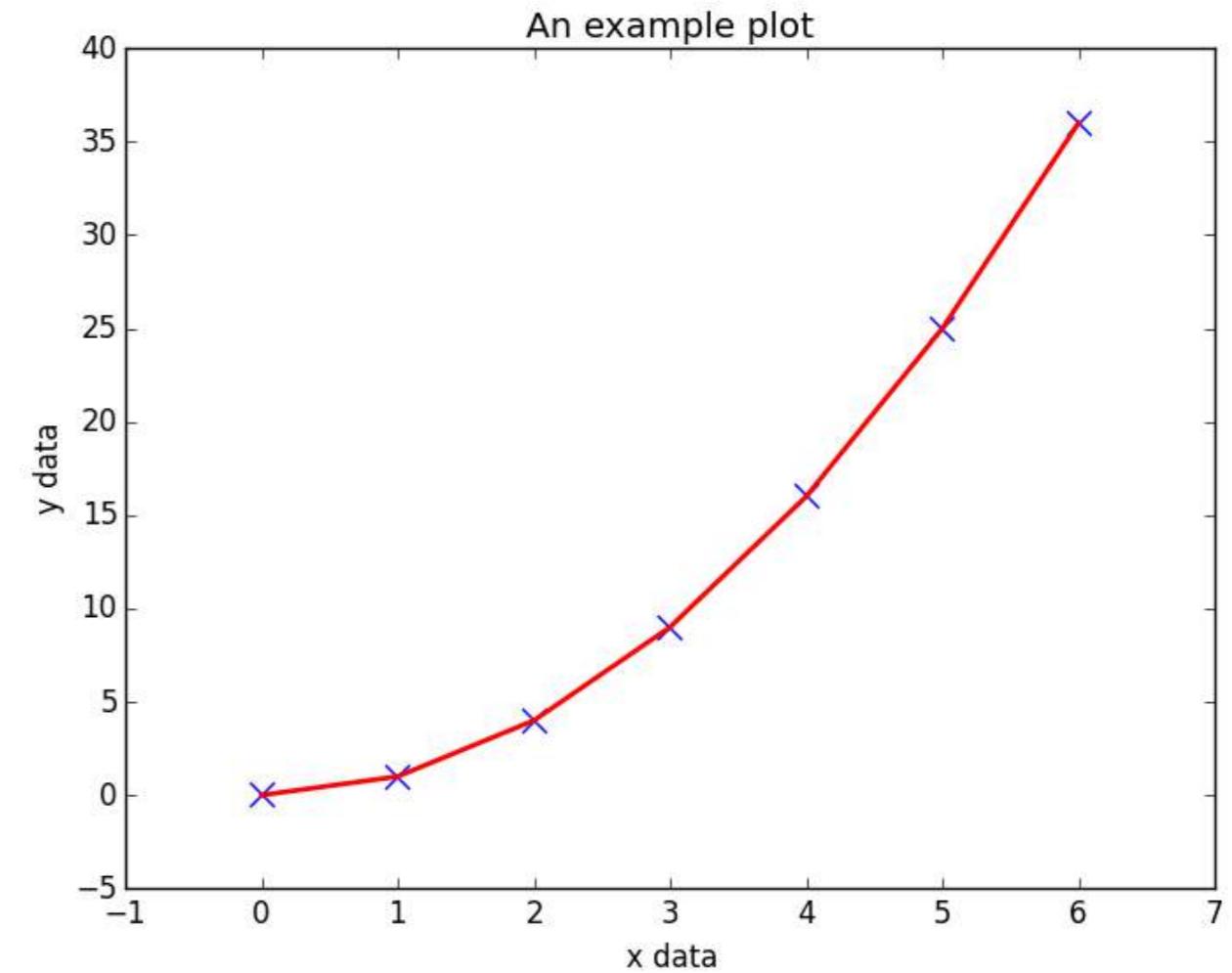
用户可以安装plotly库并在用户认证后离线使用。该库的安装和离线认证说明见此处。此外，也可以在Jupyter Notebooks中绘制图表。

使用此库需要一个带有用户名和密码的账户。这可以提供在云端保存图表和数据的工作空间。

该库的免费版本功能略有局限，设计为每天制作250个图表。付费版本拥有所有功能、无限制的图表下载以及更多的私有数据存储。欲了解更多详情，可访问主页 [here](#)。

```
plt.ylabel('y data')
plt.title('An example plot')

# Generate the plot and show to the user.
plt.show()
```



Note that `plt.show()` is known to be [problematic](#) in some environments due to running `matplotlib.pyplot` in interactive mode, and if so, the blocking behaviour can be overridden explicitly by passing in an optional argument, `plt.show(block=True)`, to alleviate the issue.

### Section 108.3: Plotly

Plotly是一个现代的绘图和数据可视化平台。对于生成各种图表尤其是数据科学方面非常有用，Plotly作为库支持Python、R、JavaScript、Julia和MATLAB。它也可以作为这些语言的网络应用程序使用。

Users can install plotly library and use it offline after user authentication. The installation of this library and offline authentication is given [here](#). Also, the plots can be made in **Jupyter Notebooks** as well.

Usage of this library requires an account with username and password. This gives the workspace to save plots and data on the cloud.

The free version of the library has some slightly limited features and designed for making 250 plots per day. The paid version has all the features, unlimited plot downloads and more private data storage. For more details, one can visit the main page [here](#).

文档示例中的一个样本图表：

```
import plotly.graph_objs as go
import plotly as ply

# 使用 numpy 创建随机数据
import numpy as np

N = 100
random_x = np.linspace(0, 1, N)
random_y0 = np.random.randn(N)+5
random_y1 = np.random.randn(N)
random_y2 = np.random.randn(N)-5

# 创建轨迹
trace0 = go.Scatter(
    x = random_x,
    y = random_y0,
    mode = 'lines',
    name = 'lines'
)
trace1 = go.Scatter(
    x = random_x,
    y = random_y1,
    mode = 'lines+markers',
    name = 'lines+markers'
)
trace2 = go.Scatter(
    x = random_x,
    y = random_y2,
    mode = 'markers',
    name = 'markers'
)
data = [trace0, trace1, trace2]

ply.offline.plot(data, filename='line-mode')
```

A sample plot from the documentation examples:

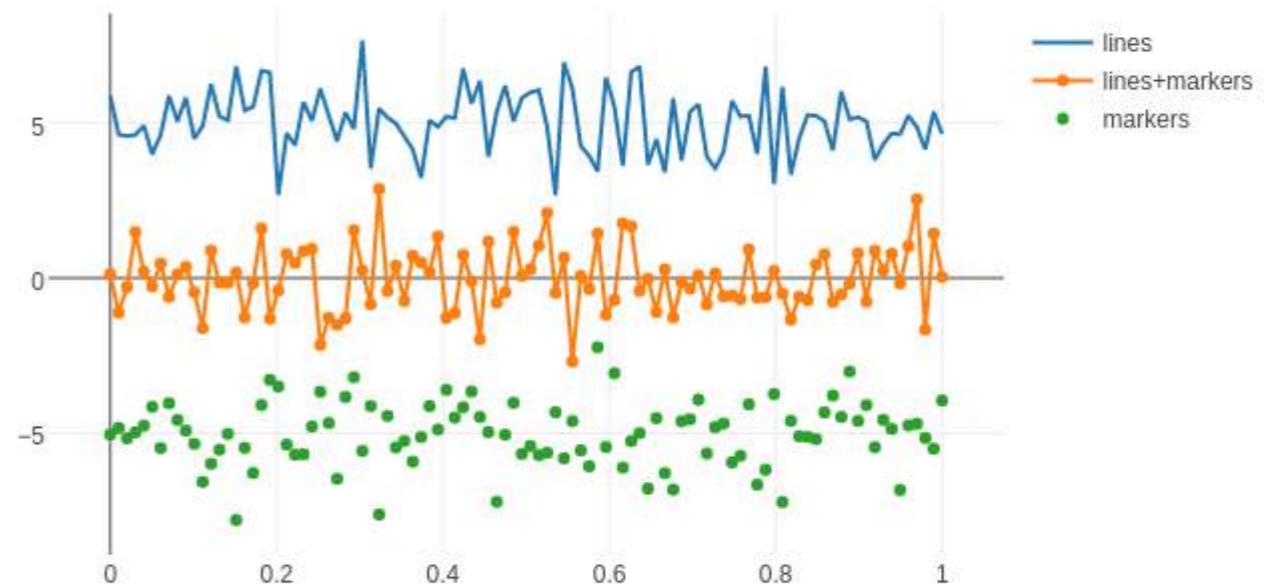
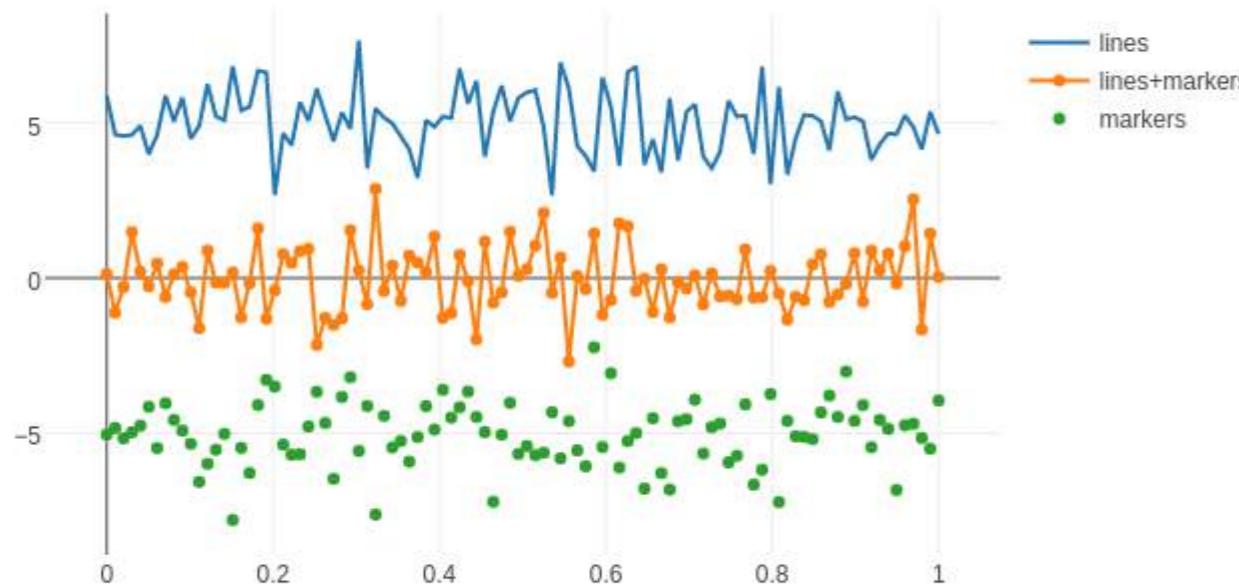
```
import plotly.graph_objs as go
import plotly as ply

# Create random data with numpy
import numpy as np

N = 100
random_x = np.linspace(0, 1, N)
random_y0 = np.random.randn(N)+5
random_y1 = np.random.randn(N)
random_y2 = np.random.randn(N)-5

# Create traces
trace0 = go.Scatter(
    x = random_x,
    y = random_y0,
    mode = 'lines',
    name = 'lines'
)
trace1 = go.Scatter(
    x = random_x,
    y = random_y1,
    mode = 'lines+markers',
    name = 'lines+markers'
)
trace2 = go.Scatter(
    x = random_x,
    y = random_y2,
    mode = 'markers',
    name = 'markers'
)
data = [trace0, trace1, trace2]

ply.offline.plot(data, filename='line-mode')
```



## 第108.4节：MayaVI

[MayaVI](#) 是一个用于科学数据的三维可视化工具。它在底层使用了可视化工具包 (Visualization Tool Kit, 简称 VTK)。利用 VTK 的强大功能，MayaVI 能够生成各种三维图形和图表。它既可以作为独立的软件应用程序使用，也可以作为一个库使用。类似于 Matplotlib，该库提供了面向对象的编程接口，使用户无需了解 VTK 即可创建图表。

**MayaVI 仅支持 Python 2.7x 系列！希望能尽快支持 Python 3-x 系列！**

(尽管在 Python 3 中使用其依赖项时已取得一些成功)

文档可以在 [here](#) 找到。一些画廊示例位于 [here](#)

这是使用 **MayaVI** 从文档中创建的示例图。

```
# 作者: Gael Varoquaux <gael.varoquaux@normalesup.org>
# 版权所有 (c) 2007, Enthought 公司。
# 许可证: BSD 风格。
```

```
从 numpy 导入 sin, cos, mgrid, pi, sqrt
从 mayavi 导入 mlab

mlab.figure(fgcolor=(0, 0, 0), bgcolor=(1, 1, 1))
u, v = mgrid[- 0.035:pi:0.01, - 0.035:pi:0.01]

X = 2 / 3. * (cos(u) * cos(2 * v)
    + sqrt(2) * sin(u) * cos(v)) * cos(u) / (sqrt(2) -
        sin(2 * u) * sin(3 * v))
Y = 2 / 3. * (cos(u) * sin(2 * v) -
    sqrt(2) * sin(u) * sin(v)) * cos(u) / (sqrt(2)
        - sin(2 * u) * sin(3 * v))
```

## Section 108.4: MayaVI

[MayaVI](#) is a 3D visualization tool for scientific data. It uses the Visualization Tool Kit or [VTK](#) under the hood. Using the power of [VTK](#), **MayaVI** is capable of producing a variety of 3-Dimensional plots and figures. It is available as a separate software application and also as a library. Similar to [Matplotlib](#), this library provides an object oriented programming language interface to create plots without having to know about [VTK](#).

**MayaVI is available only in Python 2.7x series! It is hoped to be available in Python 3-x series soon!**

(Although some success is noticed when using its dependencies in Python 3)

Documentation can be found [here](#). Some gallery examples are found [here](#)

Here is a sample plot created using **MayaVI** from the documentation.

```
# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>
# Copyright (c) 2007, Enthought, Inc.
# License: BSD Style.
```

```
from numpy import sin, cos, mgrid, pi, sqrt
from mayavi import mlab

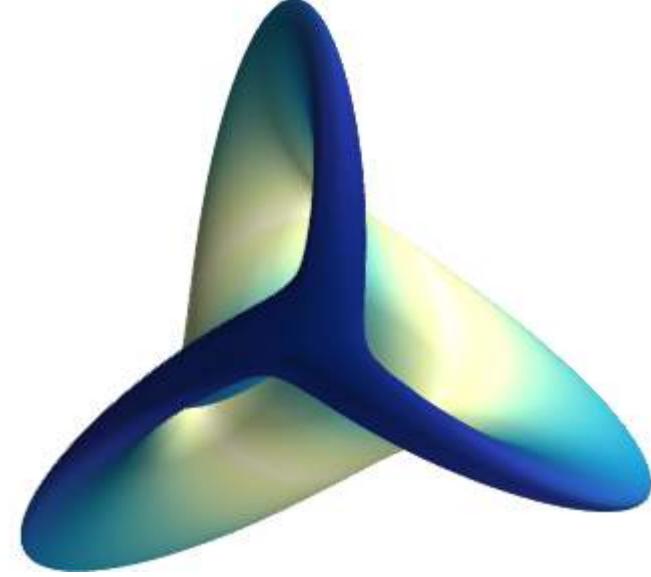
mlab.figure(fgcolor=(0, 0, 0), bgcolor=(1, 1, 1))
u, v = mgrid[- 0.035:pi:0.01, - 0.035:pi:0.01]

X = 2 / 3. * (cos(u) * cos(2 * v)
    + sqrt(2) * sin(u) * cos(v)) * cos(u) / (sqrt(2) -
        sin(2 * u) * sin(3 * v))
Y = 2 / 3. * (cos(u) * sin(2 * v) -
    sqrt(2) * sin(u) * sin(v)) * cos(u) / (sqrt(2)
        - sin(2 * u) * sin(3 * v))
```

```
Z = -sqrt(2) * cos(u) * cos(u) / (sqrt(2) - sin(2 * u) * sin(3 * v))
S = sin(u)

mlab.mesh(X, Y, Z, scalars=S, colormap='YlGnBu', )

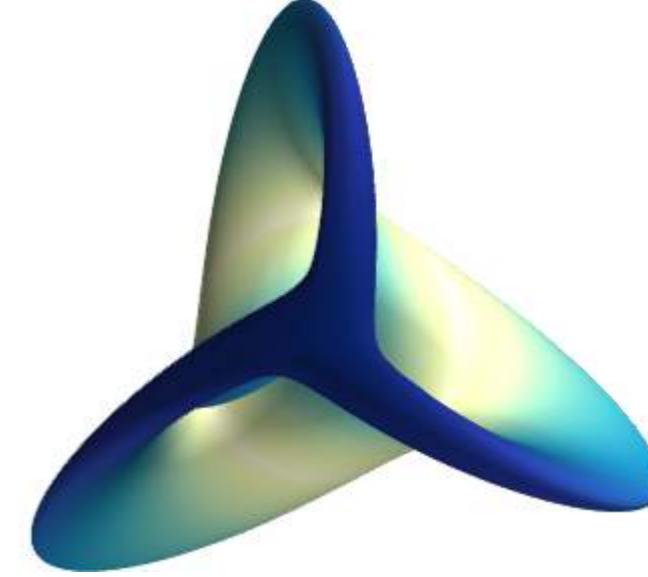
# 从正面看风景很好
mlab.view(.0, - 5.0, 4)
mlab.show()
```



```
Z = -sqrt(2) * cos(u) * cos(u) / (sqrt(2) - sin(2 * u) * sin(3 * v))
S = sin(u)

mlab.mesh(X, Y, Z, scalars=S, colormap='YlGnBu', )

# Nice view from the front
mlab.view(.0, - 5.0, 4)
mlab.show()
```



# 第109章：解释器（命令行控制台）

## 第109.1节：获取一般帮助

如果在控制台中调用`help`函数且不带任何参数，Python会显示一个交互式帮助控制台，您可以在其中了解Python模块、符号、关键字等内容。

```
>>> help()
```

欢迎使用Python 3.4的帮助工具！

如果这是您第一次使用Python，强烈建议您查看网络上的教程：<http://docs.python.org/3.4/tutorial/>。

输入任何模块、关键字或主题的名称，即可获得有关编写Python程序和使用Python模块的帮助。要退出此帮助工具并返回解释器，只需输入“quit”。

要获取可用模块、关键字、符号或主题的列表，请输入“modules”、“keywords”、“symbols”或“topics”。每个模块还附带一行简要说明其功能；要列出名称或说明中包含特定字符串（如“spam”）的模块，请输入“modules spam”。

## 第109.2节：引用上一个表达式

要获取控制台中上一个表达式的结果值，请使用下划线`_`。

```
>>> 2 + 2
4
>>> _
4
>>> _ + 6
10
```

这个魔法下划线变量仅在使用返回值的Python表达式时更新。定义函数或for循环不会改变该值。如果表达式引发异常，该值不会发生变化。

```
>>> "Hello, {0}" .format("World")
'Hello, World'
>>> _
'Hello, World'
>>> def wontchangethings():
...     pass
>>> _
'Hello, World'
>>> 27 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError:除以零错误
>>> _
'Hello, World'
```

请记住，这个魔法变量仅在交互式Python解释器中可用。运行脚本时不会生效。

# Chapter 109: The Interpreter (Command Line Console)

## Section 109.1: Getting general help

If the `help` function is called in the console without any arguments, Python presents an interactive help console, where you can find out about Python modules, symbols, keywords and more.

```
>>> help()
```

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the Internet at <http://docs.python.org/3.4/tutorial/>.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type "modules", "keywords", "symbols", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose name or summary contain a given string such as "spam", type "modules spam".

## Section 109.2: Referring to the last expression

To get the value of the last result from your last expression in the console, use an underscore `_`.

```
>>> 2 + 2
4
>>> _
4
>>> _ + 6
10
```

This magic underscore value is only updated when using a python expression that results in a value. Defining functions or for loops does not change the value. If the expression raises an exception there will be no changes to `_`.

```
>>> "Hello, {0}" .format("World")
'Hello, World'
>>> _
'Hello, World'
>>> def wontchangethings():
...     pass
>>> _
'Hello, World'
>>> 27 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> _
'Hello, World'
```

Remember, this magic variable is only available in the interactive python interpreter. Running scripts will not do this.

## 第109.3节：打开Python控制台

通常可以通过在Windows控制台输入 `py`, 或在其他平台输入 `python` 来打开主要版本的Python控制台。

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, 2015年2月24日, 22:44:40) [MSC v.1600 64 位 (AMD64)] 运行于 win32
输入 "help", "copyright", "credits" 或 "license" 获取更多信息。
>>>
```

如果你有多个版本，默认情况下它们的可执行文件将分别映射为 `python2` 或 `python3`。

这当然取决于Python可执行文件是否在你的PATH环境变量中。

## 第109.4节：PYTHONSTARTUP变量

你可以为 Python 控制台设置一个名为 `PYTHONSTARTUP` 的环境变量。每当你进入 Python 控制台时，这个文件将被执行，从而允许你为控制台添加额外功能，比如自动导入常用模块。

如果 `PYTHONSTARTUP` 变量被设置为包含以下内容的文件位置：

```
print("Welcome!")
```

那么打开 Python 控制台时将会显示以下额外输出：

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, 2015年2月24日, 22:44:40) [MSC v.1600 64 位 (AMD64)] 在 win32
输入 "help"、"copyright"、"credits" 或 "license" 获取更多信息。
欢迎！
>>>
```

## 第109.5节：命令行参数

Python 有多种可以传递给 `py` 的命令行开关。你可以通过执行 `py --help` 来查看，这在 Python 3.4 中会显示如下内容：

### Python 启动器

用法：`py [ 启动器参数 ] [ python 参数 ] 脚本 [ 脚本参数 ]`

启动器参数：

- 2 : 启动最新的 Python 2.x 版本
- 3 : 启动最新的 Python 3.x 版本
- X.Y : 启动指定的 Python 版本
- X.Y-32 : 启动指定的 32 位 Python 版本

以下帮助文本来自 Python：

```
用法: G:\\Python34\\python.exe [选项] ... [-c cmd | -m mod | 文件 | -] [参数] ...
选项和参数 (及对应的环境变量) :
-b : 对 str(bytes_instance)、str(bytarray_instance) 发出警告,
以及比较 bytes/bytarray 与 str。 (-bb: 发出错误)
-B : 导入时不写入 .py[co] 文件；也可通过设置 PYTHONDONTWRITEBYTECODE=x 实现
-c cmd : 以字符串形式传入程序 (终止选项列表)
```

## Section 109.3: Opening the Python console

The console for the primary version of Python can usually be opened by typing `py` into your windows console or `python` on other platforms.

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you have multiple versions, then by default their executables will be mapped to `python2` or `python3` respectively.

This of course depends on the Python executables being in your PATH.

## Section 109.4: The PYTHONSTARTUP variable

You can set an environment variable called `PYTHONSTARTUP` for Python's console. Whenever you enter the Python console, this file will be executed, allowing for you to add extra functionality to the console such as importing commonly-used modules automatically.

If the `PYTHONSTARTUP` variable was set to the location of a file containing this:

```
print("Welcome!")
```

Then opening the Python console would result in this extra output:

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Welcome!
>>>
```

## Section 109.5: Command line arguments

Python has a variety of command-line switches which can be passed to `py`. These can be found by performing `py --help`, which gives this output on Python 3.4:

### Python Launcher

usage: `py [ launcher-arguments ] [ python-arguments ] script [ script-arguments ]`

Launcher arguments:

- 2 : Launch the latest Python 2.x version
- 3 : Launch the latest Python 3.x version
- X.Y : Launch the specified Python version
- X.Y-32: Launch the specified 32bit Python version

The following help text is from Python:

```
usage: G:\\Python34\\python.exe [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b : issue warnings about str(bytes_instance), str(bytarray_instance)
and comparing bytes/bytarray with str. (-bb: issue errors)
-B : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
```

-d : 来自解析器的调试输出；也可使用 PYTHONDEBUG=x  
 -E : 忽略 PYTHON\* 环境变量（例如 PYTHONPATH）  
 -h : 打印此帮助信息并退出（也包括 --help）  
 -i : 运行脚本后进入交互式检查；即使标准输入看起来不是终端，也强制显示提示符；也可通过 PYTHONINSPECT=x 实现  
 -I : 将 Python 与用户环境隔离（隐含 -E 和 -s 选项）  
 -m mod : 以脚本方式运行库模块（终止选项列表）  
 -O : 稍微优化生成的字节码；也可通过 PYTHONOPTIMIZE=x 实现  
 -OO : 在 -O 优化基础上移除文档字符串  
 -q : 交互式启动时不打印版本和版权信息  
 -s : 不将用户站点目录添加到 sys.path；也可通过 PYTHONNOUSERSITE 实现  
 -S : 初始化时不隐式执行 'import site'  
 -u : 无缓冲的二进制标准输出和标准错误，标准输入始终缓冲；  
 也可使用 PYTHONUNBUFFERED=x  
 有关“-u”内部缓冲的详细信息，请参见手册页  
 -v : 详细模式（跟踪导入语句）；也可使用 PYTHONVERBOSE=x  
 可以多次使用以增加详细程度  
 -V : 打印 Python 版本号并退出（也可使用 --version）  
 -W arg : 警告控制；arg 格式为 action:message:category:module:lineno  
 也可使用 PYTHONWARNINGS=arg  
 -x : 跳过源代码的第一行，允许使用非 Unix 形式的 #!cmd  
 -X opt : 设置实现特定的选项  
 file : 从脚本文件读取程序  
 - : 从标准输入读取程序（默认；如果是终端则为交互模式）  
 arg ... : 传递给程序的参数，位于 sys.argv[1:]

#### 其他环境变量：

PYTHONSTARTUP : 交互式启动时执行的文件（无默认值）PYTHONPATH : 以分号 ';' 分隔的目录列表，前缀添加到默认模块搜索路径。结果为 sys.path。

PYTHONHOME : 备用目录（或分号 ';'）。

默认模块搜索路径使用 \\lib。

PYTHONCASEOK : 忽略'import'语句中的大小写（Windows）。

PYTHONIOENCODING : 用于stdin/stdout/stderr的编码[:错误处理]。

PYTHONFAULTHANDLER : 在致命错误时转储Python回溯信息。

PYTHONHASHSEED : 如果该变量设置为'random'，则使用随机值作为str、bytes和datetime对象哈希的种子。它也可以设置为范围[0,4294967295]内的整数，以获得具有可预测种子的哈希值。

-d : debug output from parser; also PYTHONDEBUG=x  
 -E : ignore PYTHON\* environment variables (such as PYTHONPATH)  
 -h : print this help message and exit (also --help)  
 -i : inspect interactively after running script; forces a prompt even if stdin does not appear to be a terminal; also PYTHONINSPECT=x  
 -I : isolate Python from the user's environment (implies -E and -s)  
 -m mod : run library module as a script (terminates option list)  
 -O : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x  
 -OO : remove doc-strings in addition to the -O optimizations  
 -q : don't print version and copyright messages on interactive startup  
 -s : don't add user site directory to sys.path; also PYTHONNOUSERSITE  
 -S : don't imply 'import site' on initialization  
 -u : unbuffered binary stdout and stderr, stdin always buffered;  
 also PYTHONUNBUFFERED=x  
 see man page for details on internal buffering relating to '-u'  
 -v : verbose (trace import statements); also PYTHONVERBOSE=x  
 can be supplied multiple times to increase verbosity  
 -V : print the Python version number and exit (also --version)  
 -W arg : warning control; arg is action:message:category:module:lineno  
 also PYTHONWARNINGS=arg  
 -x : skip first line of source, allowing use of non-Unix forms of #!cmd  
 -X opt : set implementation-specific option  
 file : program read from script file  
 - : program read from stdin (default; interactive mode if a tty)  
 arg ...: arguments passed to program in sys.argv[1:]

#### Other environment variables:

PYTHONSTARTUP: file executed on interactive startup (no default)  
 PYTHONPATH : ';' -separated list of directories prefixed to the default module search path. The result is sys.path.

PYTHONHOME : alternate directory (or ;).

The default module search path uses \\lib.

PYTHONCASEOK : ignore case in 'import' statements (Windows).

PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.

PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.

PYTHONHASHSEED: if this variable is set to 'random', a random value is used to seed the hashes of str, bytes and datetime objects. It can also be set to an integer in the range [0,4294967295] to get hash values with a predictable seed.

## 第109.6节：获取对象的帮助

Python控制台添加了一个新函数，help，可以用来获取函数或对象的信息。

对于函数，help会打印其签名（参数）和文档字符串（如果函数有的话）。

```
>>> help(print)
内置函数 print 的帮助，位于模块 builtins 中：
```

```
print(*args, **kwargs)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)  将值打印到流中，默
```

认打印到 sys.stdout。

可选的关键字参数：

file: 类文件对象（流）；默认是当前的 sys.stdout。

sep: 字符串，插入在值之间，默认是空格。

end: 字符串，附加在最后一个值后，默认是换行符。

flush: 是否强制刷新流。

对于一个对象，help 会列出该对象的文档字符串和该对象拥有的不同成员函数。

## Section 109.6: Getting help about an object

The Python console adds a new function, `help`, which can be used to get information about a function or object.

For a function, `help` prints its signature (arguments) and its docstring, if the function has one.

```
>>> help(print)
Help on built-in function print in module builtins:
```

```
print(*args, **kwargs)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

For an object, `help` lists the object's docstring and the different member functions which the object has.

```
>>> x = 2
>>> help(x)
帮助 int 对象:

class int(object)
|   int(x=0) -> integer
|   int(x, base=10) -> integer
|
|   将数字或字符串转换为整数，如果没有参数则返回0。如果 x 是数字，则返回 x.__int__(). 对于浮点数，此操作向零截断。
|
|   如果 x 不是数字或给定了 base，则 x 必须是表示整数字面量的字符串、字节或字节数组实例，基数为给定的 base。字面量可以以 '+' 或 '-' 开头，并且可以被空白字符包围。基数默认为10。有效基数为0和2至36。
|
|   基数0表示从字符串中解释基数作为整数字面量。
|   >>> int('0b100', base=0)
|   4
|
|   此处定义的方法：
|
|   __abs__(self, /)
|       返回self的绝对值
|
|   __add__(self, value, /)
|       返回self+value...
```

```
>>> x = 2
>>> help(x)
Help on int object:

class int(object)
|   int(x=0) -> integer
|   int(x, base=10) -> integer
|
|   Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.
|
|   If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36.
|   Base 0 means to interpret the base from the string as an integer literal.
|   >>> int('0b100', base=0)
|   4
|
|   Methods defined here:
|
|   __abs__(self, /)
|       abs(self)
|
|   __add__(self, value, /)
|       Return self+value...
```

# 视频：Python数据科学与机器学习训练营

学习如何使用NumPy、Pandas、Seaborn、Matplotlib、Plotly、Scikit-Learn、机器学习、Tensorflow等！



- ✓ 使用Python进行数据科学和机器学习
- ✓ 使用Spark进行大数据分析
- ✓ 实现机器学习算法
- ✓ 学习使用NumPy进行数值数据处理
- ✓ 学习使用Pandas进行数据分析
- ✓ 学习使用Matplotlib进行Python绘图
- ✓ 学习使用Seaborn进行统计图表绘制
- ✓ 使用Plotly进行交互式动态可视化
- ✓ 使用SciKit-Learn完成机器学习任务
- ✓ K均值聚类
- ✓ 逻辑回归
- ✓ 线性回归
- ✓ 随机森林和决策树
- ✓ 神经网络
- ✓ 支持向量机今天观看→

# VIDEO: Python for Data Science and Machine Learning Bootcamp

Learn how to use NumPy, Pandas, Seaborn, Matplotlib , Plotly, Scikit-Learn , Machine Learning, Tensorflow, and more!



- ✓ Use Python for Data Science and Machine Learning
- ✓ Use Spark for Big Data Analysis
- ✓ Implement Machine Learning Algorithms
- ✓ Learn to use NumPy for Numerical Data
- ✓ Learn to use Pandas for Data Analysis
- ✓ Learn to use Matplotlib for Python Plotting
- ✓ Learn to use Seaborn for statistical plots
- ✓ Use Plotly for interactive dynamic visualizations
- ✓ Use SciKit-Learn for Machine Learning Tasks
- ✓ K-Means Clustering
- ✓ Logistic Regression
- ✓ Linear Regression
- ✓ Random Forest and Decision Trees
- ✓ Neural Networks
- ✓ Support Vector Machines

Watch Today →

# 第110章：\*args 和 \*\*kwargs

## 第110.1节：编写函数时使用 \*\*kwargs

你可以通过在参数名前使用双星号来定义一个接受任意数量关键字（命名）参数的函数

\*\* 在参数名前：

```
def print_kwargs(**kwargs):
    print(kwargs)
```

调用该方法时，Python 会构造一个包含所有关键字参数的字典，并在函数体内提供该字典：

```
print_kwargs(a="two", b=3)
# 输出：{"a": "two", "b": 3}
```

注意，函数定义中的 \*\*kwargs 参数必须始终是最后一个参数，并且它只会匹配在之前参数之后传入的参数。

```
def example(a, **kw):
    print kw

example(a=2, b=3, c=4) # => {'b': 3, 'c': 4}
```

在函数体内，kwargs 的操作方式与字典相同；要访问 kwargs 中的单个元素，只需像遍历普通字典一样遍历它们：

```
def print_kwargs(**kwargs):
    for key in kwargs:
        print("key = {0}, value = {1}".format(key, kwargs[key]))
```

现在，调用 print\_kwargs(a="two", b=1) 会显示以下输出：

```
print_kwargs(a = "two", b = 1)
key = a, value = "two"
key = b, value = 1
```

## 第110.2节：编写函数时使用 \*args

编写函数时，可以使用星号 \* 来收集所有位置参数（即未命名参数）到一个元组中：

```
def print_args(farg, *args):
    print("形式参数: %s" % farg)
    for arg in args:
        print("另一个位置参数: %s" % arg)
```

调用方法：

```
print_args(1, "two", 3)
```

在该调用中，farg 将按常规赋值，另外两个参数将按接收顺序传入 args 元组中。

# Chapter 110: \*args and \*\*kwargs

## Section 110.1: Using \*\*kwargs when writing functions

You can define a function that takes an arbitrary number of keyword (named) arguments by using the double star \*\* before a parameter name:

```
def print_kwargs(**kwargs):
    print(kwargs)
```

When calling the method, Python will construct a dictionary of all keyword arguments and make it available in the function body:

```
print_kwargs(a="two", b=3)
# prints: {"a": "two", "b": 3}
```

Note that the \*\*kwargs parameter in the function definition must always be the last parameter, and it will only match the arguments that were passed in after the previous ones.

```
def example(a, **kw):
    print kw

example(a=2, b=3, c=4) # => {'b': 3, 'c': 4}
```

Inside the function body, kwargs is manipulated in the same way as a dictionary; in order to access individual elements in kwargs you just loop through them as you would with a normal dictionary:

```
def print_kwargs(**kwargs):
    for key in kwargs:
        print("key = {0}, value = {1}".format(key, kwargs[key]))
```

Now, calling print\_kwargs(a="two", b=1) shows the following output:

```
print_kwargs(a = "two", b = 1)
key = a, value = "two"
key = b, value = 1
```

## Section 110.2: Using \*args when writing functions

You can use the star \* when writing a function to collect all positional (ie. unnamed) arguments in a tuple:

```
def print_args(farg, *args):
    print("formal arg: %s" % farg)
    for arg in args:
        print("another positional arg: %s" % arg)
```

Calling method:

```
print_args(1, "two", 3)
```

In that call, farg will be assigned as always, and the two others will be fed into the args tuple, in the order they were received.

## 第110.3节：使用字典填充关键字参数值

```
def foobar(foo=None, bar=None):
    return "{}{}".format(foo, bar)

values = {"foo": "foo", "bar": "bar"}

foobar(**values) # "foobar"
```

## 第110.4节：仅限关键字和必须关键字参数

Python 3允许你定义只能通过关键字赋值的函数参数，即使没有默认值。这是通过使用星号 \* 来消费额外的位置参数而不设置关键字参数来实现的。所有位于 \* 之后的参数都是仅限关键字（即非位置）参数。注意，如果仅限关键字参数没有默认值，调用函数时仍然是必需的。

```
def print_args(arg1, *args, keyword_required, keyword_only=True):
    print("第一个位置参数: {}".format(arg1))
    for arg in args:
        print("另一个位置参数: {}".format(arg))
    print("keyword_required 的值: {}".format(keyword_required))
    print("keyword_only 的值: {}".format(keyword_only))

print(1, 2, 3, 4) # TypeError: print_args() 缺少 1 个必需的仅限关键字参数:
'keyword_required'
print(1, 2, 3, keyword_required=4)
# 第一个位置参数: 1
# 另一个位置参数: 2
# 另一个位置参数: 3
# keyword_required 的值: 4
# keyword_only 的值: True
```

## 第110.5节：调用函数时使用\*\*kwargs

你可以使用字典为函数的参数赋值；使用参数名作为字典中的键，参数值绑定到每个键：

```
def test_func(arg1, arg2, arg3): # 常规的三个参数函数
    print("arg1: %s" % arg1)
    print("arg2: %s" % arg2)
    print("arg3: %s" % arg3)

# 注意字典是无序的，所以我们交换arg2和arg3。只有名称才重要。
kwargs = {"arg3": 3, "arg2": "two"}

# 将第一个参数（即arg1）绑定为1，使用kwargs字典绑定其他参数
test_var_args_call(1, **kwargs)
```

## 第110.6节：\*\*kwargs和默认值

使用\*\*kwargs时的默认值

```
def fun(**kwargs):
    print(kwargs.get('value', 0))
```

## Section 110.3: Populating kwarg values with a dictionary

```
def foobar(foo=None, bar=None):
    return "{}{}".format(foo, bar)

values = {"foo": "foo", "bar": "bar"}

foobar(**values) # "foobar"
```

## Section 110.4: Keyword-only and Keyword-required arguments

Python 3 allows you to define function arguments which can only be assigned by keyword, even without default values. This is done by using star \* to consume additional positional parameters without setting the keyword parameters. All arguments after the \* are keyword-only (i.e. non-positional) arguments. Note that if keyword-only arguments aren't given a default, they are still required when calling the function.

```
def print_args(arg1, *args, keyword_required, keyword_only=True):
    print("first positional arg: {}".format(arg1))
    for arg in args:
        print("another positional arg: {}".format(arg))
    print("keyword_required value: {}".format(keyword_required))
    print("keyword_only value: {}".format(keyword_only))

print(1, 2, 3, 4) # TypeError: print_args() missing 1 required keyword-only argument:
'keyword_required'
print(1, 2, 3, keyword_required=4)
# first positional arg: 1
# another positional arg: 2
# another positional arg: 3
# keyword_required value: 4
# keyword_only value: True
```

## Section 110.5: Using \*\*kwargs when calling functions

You can use a dictionary to assign values to the function's parameters; using parameters name as keys in the dictionary and the value of these arguments bound to each key:

```
def test_func(arg1, arg2, arg3): # Usual function with three arguments
    print("arg1: %s" % arg1)
    print("arg2: %s" % arg2)
    print("arg3: %s" % arg3)

# Note that dictionaries are unordered, so we can switch arg2 and arg3. Only the names matter.
kwargs = {"arg3": 3, "arg2": "two"}

# Bind the first argument (ie. arg1) to 1, and use the kwargs dictionary to bind the others
test_var_args_call(1, **kwargs)
```

## Section 110.6: \*\*kwargs and default values

To use default values with \*\*kwargs

```
def fun(**kwargs):
    print(kwargs.get('value', 0))
```

```
fun()  
# print 0  
fun(value=1)  
# print 1
```

## 第110.7节：调用函数时使用 \*args

调用函数时对参数使用\*操作符的效果是解包列表或元组参数

```
def print_args(arg1, arg2):  
    print(str(arg1) + str(arg2))  
  
a = [1, 2]  
b = tuple([3, 4])  
  
print_args(*a)  
# 12  
print_args(*b)  
# 34
```

请注意，带星号的参数的长度需要等于函数参数的数量。

一个常见的 Python 习惯用法是将解包操作符 \* 与 zip 函数一起使用，以逆转其效果：

```
a = [1, 3, 5, 7, 9]  
b = [2, 4, 6, 8, 10]  
  
zipped = zip(a, b)  
# [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]  
  
zip(*zipped)  
# (1, 3, 5, 7, 9), (2, 4, 6, 8, 10)
```

```
fun()  
# print 0  
fun(value=1)  
# print 1
```

## Section 110.7: Using \*args when calling functions

The effect of using the \* operator on an argument when calling a function is that of unpacking the list or a tuple argument

```
def print_args(arg1, arg2):  
    print(str(arg1) + str(arg2))  
  
a = [1, 2]  
b = tuple([3, 4])  
  
print_args(*a)  
# 12  
print_args(*b)  
# 34
```

Note that the length of the starred argument need to be equal to the number of the function's arguments.

A common python idiom is to use the unpacking operator \* with the zip function to reverse its effects:

```
a = [1, 3, 5, 7, 9]  
b = [2, 4, 6, 8, 10]  
  
zipped = zip(a, b)  
# [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]  
  
zip(*zipped)  
# (1, 3, 5, 7, 9), (2, 4, 6, 8, 10)
```

# 第111章：垃圾回收

## 第111.1节：原始对象的重用

一个有趣的现象，可能有助于优化你的应用程序，那就是原始类型实际上在底层也是有引用计数的。我们来看数字；对于所有介于-5到256之间的整数，Python总是重用相同的对象：

```
>>> import sys  
>>> sys.getrefcount(1)  
797  
>>> a = 1  
>>> b = 1  
>>> sys.getrefcount(1)  
799
```

注意引用计数增加，这意味着当 a 和 b 引用时，它们指向的是同一个底层对象。然而，对于较大的数字，Python 实际上并不会重用底层对象：

```
>>> a = 999999999  
>>> sys.getrefcount(999999999)  
3  
>>> b = 999999999  
>>> sys.getrefcount(999999999)  
3
```

因为给 999999999 的引用计数在赋值给 a 和 b 时没有变化，我们可以推断它们指向两个不同的底层对象，尽管它们都被赋值为相同的原始类型。

## 第111.2节：del命令的影响

使用 `del v` 从作用域中移除变量名，或使用 `del v[item]` 或 `del[i:j]` 从集合中移除对象，或使用 `del v.name` 移除属性，或通过任何其他方式移除对对象的引用，并不会触发任何析构函数调用或释放任何内存。对象只有在其引用计数降为零时才会被销毁。

```
>>> import gc  
>>> gc.disable() # 禁用垃圾回收器  
>>> class Track:  
    def __init__(self):  
        print("Initialized")  
    def __del__(self):  
        print("已销毁")  
>>> def bar():  
    return Track()  
>>> t = bar()  
已初始化  
>>> another_t = t # 分配另一个引用  
>>> print("...")  
...  
>>> del t          # 尚未销毁 - another_t 仍然引用它  
>>> del another_t # 最后一个引用消失，对象被销毁  
已销毁
```

# Chapter 111: Garbage Collection

## Section 111.1: Reuse of primitive objects

An interesting thing to note which may help optimize your applications is that primitives are actually also refcounted under the hood. Let's take a look at numbers; for all integers between -5 and 256, Python always reuses the same object:

```
>>> import sys  
>>> sys.getrefcount(1)  
797  
>>> a = 1  
>>> b = 1  
>>> sys.getrefcount(1)  
799
```

Note that the refcount increases, meaning that a and b reference the same underlying object when they refer to the 1 primitive. However, for larger numbers, Python actually doesn't reuse the underlying object:

```
>>> a = 999999999  
>>> sys.getrefcount(999999999)  
3  
>>> b = 999999999  
>>> sys.getrefcount(999999999)  
3
```

Because the refcount for 999999999 does not change when assigning it to a and b we can infer that they refer to two different underlying objects, even though they both are assigned the same primitive.

## Section 111.2: Effects of the del command

Removing a variable name from the scope using `del v`, or removing an object from a collection using `del v[item]` or `del[i:j]`, or removing an attribute using `del v.name`, or any other way of removing references to an object, does not trigger any destructor calls or any memory being freed in and of itself. Objects are only destructed when their reference count reaches zero.

```
>>> import gc  
>>> gc.disable() # disable garbage collector  
>>> class Track:  
    def __init__(self):  
        print("Initialized")  
    def __del__(self):  
        print("Destroyed")  
>>> def bar():  
    return Track()  
>>> t = bar()  
Initialized  
>>> another_t = t # assign another reference  
>>> print("...")  
...  
>>> del t          # not destructed yet - another_t still refers to it  
>>> del another_t # final reference gone, object is destructed  
Destroyed
```

## 第111.3节：引用计数

Python内存管理的绝大部分是通过引用计数来处理的。

每当引用一个对象（例如赋值给变量）时，其引用计数会自动增加。当取消引用（例如变量超出作用域）时，其引用计数会自动减少。

当引用计数达到零时，对象会被**立即销毁**，内存也会被立即释放。因此，在大多数情况下，垃圾回收器甚至不需要使用。

```
>>> import gc; gc.disable() # 禁用垃圾回收器
>>> class Track:
    def __init__(self):
        print("Initialized")
    def __del__(self):
        print("已销毁")
>>> def foo():
    t = Track()
    # 由于不再有任何引用，立即被销毁
    print("---")
t = Track()
# 变量被引用，因此尚未被销毁
print("---")
# 函数退出时变量被销毁
>>> foo()
已初始化
已销毁
---
已初始化
---
已销毁
```

为了进一步演示引用的概念：

```
>>> def bar():
    return Track()
>>> t = bar()
已初始化
>>> another_t = t # 分配另一个引用
>>> print("...")
...
>>> t = None          # 尚未被销毁 - another_t 仍然引用它
>>> another_t = None # 最后一个引用消失，对象被销毁
已销毁
```

## 第111.4节：引用循环的垃圾回收器

只有在存在引用循环时才需要垃圾回收器。引用循环的最简单例子是A引用B，B又引用A，而程序中没有其他任何地方引用A或B。A和B都无法从程序的任何地方访问，因此可以安全地销毁，但它们的引用计数都是1，因此仅靠引用计数算法无法释放它们。

```
>>> import gc; gc.disable() # 禁用垃圾回收器
>>> class Track:
    def __init__(self):
        print("Initialized")
    def __del__(self):
        print("已销毁")
```

## Section 111.3: Reference Counting

The vast majority of Python memory management is handled with reference counting.

Every time an object is referenced (e.g. assigned to a variable), its reference count is automatically increased. When it is dereferenced (e.g. variable goes out of scope), its reference count is automatically decreased.

When the reference count reaches zero, the object is **immediately destroyed** and the memory is immediately freed. Thus for the majority of cases, the garbage collector is not even needed.

```
>>> import gc; gc.disable() # disable garbage collector
>>> class Track:
    def __init__(self):
        print("Initialized")
    def __del__(self):
        print("Destroyed")
>>> def foo():
    t = Track()
    # destroyed immediately since no longer has any references
    print("---")
t = Track()
# variable is referenced, so it's not destroyed yet
print("---")
# variable is destroyed when function exits
>>> foo()
Initialized
Destroyed
---
Initialized
---
Destroyed
```

To demonstrate further the concept of references:

```
>>> def bar():
    return Track()
>>> t = bar()
Initialized
>>> another_t = t # assign another reference
>>> print("...")
...
>>> t = None          # not destroyed yet - another_t still refers to it
>>> another_t = None # final reference gone, object is destroyed
Destroyed
```

## Section 111.4: Garbage Collector for Reference Cycles

The only time the garbage collector is needed is if you have a *reference cycle*. The simplest example of a reference cycle is one in which A refers to B and B refers to A, while nothing else refers to either A or B. Neither A or B are accessible from anywhere in the program, so they can safely be destroyed, yet their reference counts are 1 and so they cannot be freed by the reference counting algorithm alone.

```
>>> import gc; gc.disable() # disable garbage collector
>>> class Track:
    def __init__(self):
        print("Initialized")
    def __del__(self):
        print("Destroyed")
```

```

>>> A = Track()
已初始化
>>> B = Track()
已初始化
>>> A.other = B
>>> B.other = A
>>> del A; del B # 由于引用循环，对象不会被销毁
>>> gc.collect() # 触发垃圾回收
已销毁
已销毁
4

```

引用循环可以任意长。如果 A 指向 B, B 指向 C, C 指向 ... 指向 Z, Z 又指向 A, 那么从 A 到 Z 都不会被回收，直到垃圾回收阶段：

```

>>> objs = [Track() for _ in range(10)]
已初始化
...     objs[i].other = objs[i + 1]
...
>>> objs[-1].other = objs[0] # 完成循环
>>> del objs           # 现在没有人引用 objs - 但仍未销毁
>>> gc.collect()
已销毁
20

```

## 第111.5节：强制释放对象

即使对象的引用计数不为0，也可以在Python 2和3中强制释放对象。

两个版本都使用ctypes模块来实现这一点。

**警告：**这样做会使你的Python环境不稳定，且容易崩溃且没有回溯信息！  
使用此方法也可能引入安全问题（虽然可能性很小）只释放你确定以后再也不会引用的对象。绝对不要再引用。

```

Python 3.x 版本 ≥ 3.0
import ctypes
deallocated = 12345
ctypes.pythonapi._Py_Dealloc(ctypes.py_object(deallocated))

Python 2.x 版本 ≥ 2.3

```

```

>>> A = Track()
Initialized
>>> B = Track()
Initialized
>>> A.other = B
>>> B.other = A
>>> del A; del B # objects are not destructed due to reference cycle
>>> gc.collect() # trigger collection
Destroyed
Destroyed
4

```

A reference cycle can be arbitrary long. If A points to B points to C points to ... points to Z which points to A, then neither A through Z will be collected, until the garbage collection phase:

```

>>> objs = [Track() for _ in range(10)]
Initialized
...
>>> for i in range(len(objs)-1):
...     objs[i].other = objs[i + 1]
...
>>> objs[-1].other = objs[0] # complete the cycle
>>> del objs           # no one can refer to objs now - still not destructed
>>> gc.collect()
Destroyed
20

```

## Section 111.5: Forcefully deallocating objects

You can force deallocate objects even if their refcount isn't 0 in both Python 2 and 3.

Both versions use the ctypes module to do so.

**WARNING:** doing this *will* leave your Python environment unstable and prone to crashing without a traceback!  
Using this method could also introduce security problems (quite unlikely) Only deallocate objects you're sure you'll never reference again. Ever.

```

Python 3.x 版本 ≥ 3.0
import ctypes
deallocated = 12345
ctypes.pythonapi._Py_Dealloc(ctypes.py_object(deallocated))

Python 2.x 版本 ≥ 2.3

```

```
import ctypes, sys  
deallocated = 12345  
(ctypes.c_char * sys.getsizeof(deallocated)).from_address(id(deallocated))[4:] = " * 4
```

运行后，任何对已释放对象的引用都会导致 Python 产生未定义行为或崩溃——且不会有追踪信息。垃圾回收器没有移除该对象，可能是有原因的.....

如果你释放了None，会收到一条特殊信息——致命的 Python 错误：在崩溃前释放了None。

## 第111.6节：查看对象的引用计数

```
>>> import sys  
>>> a = object()  
>>> sys.getrefcount(a)  
2  
>>> b = a  
>>> sys.getrefcount(a)  
3  
>>> del b  
>>> sys.getrefcount(a)  
2
```

## 第111.7节：不要等待垃圾回收来清理

垃圾回收会清理并不意味着你应该等待垃圾回收周期来进行清理。

特别是你不应该等待垃圾回收来关闭文件句柄、数据库连接和打开的网络连接。

例如：

在以下代码中，假设如果 f 是对该文件的最后一个引用，则文件将在下一次垃圾回收周期关闭。

```
>>> f = open("test.txt")  
>>> del f
```

一种更明确的清理方式是调用 f.close()。你甚至可以更优雅地做到这一点，即使用 **with** 语句，也称为 上下文管理器：

```
>>> with open("test.txt") as f:  
...     pass  
...     # 对 f 做一些操作  
>>> # 现在 f 对象仍然存在，但文件已关闭
```

with 语句允许你将代码缩进到打开的文件下方。这使得文件保持打开的时间变得明确且更易于理解。即使在 while 块中抛出异常，它也总是会关闭文件。

## 第111.8节：管理垃圾回收

有两种方法可以影响内存清理的执行时间。一种是影响自动过程执行的频率，另一种是手动触发清理。

```
import ctypes, sys  
deallocated = 12345  
(ctypes.c_char * sys.getsizeof(deallocated)).from_address(id(deallocated))[4:] = '\x00' * 4
```

After running, any reference to the now deallocated object will cause Python to either produce undefined behavior or crash - without a traceback. There was probably a reason why the garbage collector didn't remove that object...

If you deallocate None, you get a special message - Fatal Python error: deallocating None before crashing.

## Section 111.6: Viewing the refcount of an object

```
>>> import sys  
>>> a = object()  
>>> sys.getrefcount(a)  
2  
>>> b = a  
>>> sys.getrefcount(a)  
3  
>>> del b  
>>> sys.getrefcount(a)  
2
```

## Section 111.7: Do not wait for the garbage collection to clean up

The fact that the garbage collection will clean up does not mean that you should wait for the garbage collection cycle to clean up.

In particular you should not wait for garbage collection to close file handles, database connections and open network connections.

for example:

In the following code, you assume that the file will be closed on the next garbage collection cycle, if f was the last reference to the file.

```
>>> f = open("test.txt")  
>>> del f
```

A more explicit way to clean up is to call f.close(). You can do it even more elegant, that is by using the **with** statement, also known as the context manager:

```
>>> with open("test.txt") as f:  
...     pass  
...     # do something with f  
>>> #now the f object still exists, but it is closed
```

The **with** statement allows you to indent your code under the open file. This makes it explicit and easier to see how long a file is kept open. It also always closes a file, even if an exception is raised in the **while** block.

## Section 111.8: Managing garbage collection

There are two approaches for influencing when a memory cleanup is performed. They are influencing how often the automatic process is performed and the other is manually triggering a cleanup.

垃圾收集器可以通过调整收集阈值来控制其运行频率。Python 使用基于代的内存管理系统。新对象被保存在最新的一代——**generation0**，每经过一次存活的收集，对象会被提升到更老的代。

达到最后一代——**generation2**后，对象将不再被提升。

可以使用以下代码片段更改阈值：

```
import gc  
gc.set_threshold(1000, 100, 10) # 这些值仅用于演示目的
```

第一个参数表示收集**generation0**的阈值。每当分配的对象数量超过释放的对象数量1000时，垃圾收集器将被调用。

为了优化过程，较老的代不会在每次运行时都被清理。第二和第三个参数是可选的，控制较老代的清理频率。如果**generation0**被处理了100次而没有清理**generation1**，那么将会处理**generation1**。类似地，只有当**generation1**被清理了10次而没有触及**generation2**时，才会处理**generation2**中的对象。

手动设置阈值有利的一个场景是程序分配了大量小对象但没有释放它们，导致垃圾收集器运行过于频繁（每**generation0\_threshold**个对象分配时）。尽管收集器运行速度相当快，但当处理大量对象时，仍会带来性能问题。无论如何，选择阈值没有一套通用策略，需根据具体使用场景决定。

手动触发垃圾收集可以使用如下代码片段：

```
import gc  
gc.collect()
```

垃圾回收是基于分配和释放的次数自动触发的，而不是基于已使用或可用的内存。因此，在处理大对象时，内存可能会在自动清理触发之前耗尽。这使得手动调用垃圾回收器成为一个很好的使用场景。

虽然这是可能的，但这并不是推荐的做法。避免内存泄漏是最好的选择。无论如何，在大型项目中，检测内存泄漏可能是一项艰巨的任务，手动触发垃圾回收可以作为一种快速解决方案，直到进一步调试完成。

对于长时间运行的程序，垃圾回收可以基于时间触发或基于事件触发。第一个例子是一个在固定请求次数后触发回收的网络服务器。后者的例子是当接收到某种类型请求时触发垃圾回收的网络服务器。

The garbage collector can be manipulated by tuning the collection thresholds which affect the frequency at which the collector runs. Python uses a generation based memory management system. New objects are saved in the newest generation - **generation0** and with each survived collection, objects are promoted to older generations. After reaching the last generation - **generation2**, they are no longer promoted.

The thresholds can be changed using the following snippet:

```
import gc  
gc.set_threshold(1000, 100, 10) # Values are just for demonstration purpose
```

The first argument represents the threshold for collecting **generation0**. Every time the number of **allocations** exceeds the number of **deallocations** by 1000 the garbage collector will be called.

The older generations are not cleaned at each run to optimize the process. The second and third arguments are **optional** and control how frequently the older generations are cleaned. If **generation0** was processed 100 times without cleaning **generation1**, then **generation1** will be processed. Similarly, objects in **generation2** will be processed only when the ones in **generation1** were cleaned 10 times without touching **generation2**.

One instance in which manually setting the thresholds is beneficial is when the program allocates a lot of small objects without deallocating them which leads to the garbage collector running too often (each **generation0\_threshold** object allocations). Even though, the collector is pretty fast, when it runs on huge numbers of objects it poses a performance issue. Anyway, there's no one size fits all strategy for choosing the thresholds and it's use case dependable.

Manually triggering a collection can be done as in the following snippet:

```
import gc  
gc.collect()
```

The garbage collection is automatically triggered based on the number of allocations and deallocations, not on the consumed or available memory. Consequently, when working with big objects, the memory might get depleted before the automated cleanup is triggered. This makes a good use case for manually calling the garbage collector.

Even though it's possible, it's not an encouraged practice. Avoiding memory leaks is the best option. Anyway, in big projects detecting the memory leak can be a tough task and manually triggering a garbage collection can be used as a quick solution until further debugging.

For long-running programs, the garbage collection can be triggered on a time basis or on an event basis. An example for the first one is a web server that triggers a collection after a fixed number of requests. For the later, a web server that triggers a garbage collection when a certain type of request is received.

# 第112章：Pickle数据序列化

参数	详细信息
对象	要存储的对象
文件	将包含对象的打开文件
协议	用于序列化对象的协议（可选参数）
缓冲区	包含序列化对象的字节对象

## 第112.1节：使用Pickle序列化和反序列化对象

pickle模块实现了一种将任意Python对象转换为字节序列的算法。这个过程也称为序列化对象。表示该对象的字节流随后可以被传输或存储，并且之后可以重建以创建具有相同特性的新的对象。

对于最简单的代码，我们使用dump()和load()函数。

### 序列化对象

```
import pickle
```

```
# pickle支持的任意对象集合。
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("字符串", b"字节字符串"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # 使用最高协议对"data"字典进行序列化。
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

### 反序列化对象

```
import pickle
```

```
with open('data.pickle', 'rb') as f:
    # 使用的协议版本会自动检测，因此我们不需要指定它。
    data = pickle.load(f)
```

### 使用pickle和字节对象

也可以使用.dumps和.loads函数对字节对象进行序列化和反序列化，这两个函数等同于.dump和.load。

```
serialized_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
# type(serialized_data) 是 bytes 类型

deserialized_data = pickle.loads(serialized_data)
# deserialized_data == data
```

## 第112.2节：自定义Pickle数据

有些数据无法被pickle。其他数据出于某些原因不应被pickle。

可以在\_\_getstate\_\_方法中定义将被pickle的内容。该方法必须返回可被pickle的内容。

# Chapter 112: Pickle data serialisation

Parameter	Details
object	The object which is to be stored
file	The open file which will contain the object
protocol	The protocol used for pickling the object (optional parameter)
buffer	A bytes object that contains a serialized object

## Section 112.1: Using Pickle to serialize and deserialize an object

The `pickle` module implements an algorithm for turning an arbitrary Python object into a series of bytes. This process is also called **serializing** the object. The byte stream representing the object can then be transmitted or stored, and later reconstructed to create a new object with the same characteristics.

For the simplest code, we use the `dump()` and `load()` functions.

### To serialize the object

```
import pickle
```

```
# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

### To deserialize the object

```
import pickle
```

```
with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

### Using pickle and byte objects

It is also possible to serialize into and deserialize out of byte objects, using the `dumps` and `loads` function, which are equivalent to `dump` and `load`.

```
serialized_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
# type(serialized_data) is bytes

deserialized_data = pickle.loads(serialized_data)
# deserialized_data == data
```

## Section 112.2: Customize Pickled Data

Some data cannot be pickled. Other data should not be pickled for other reasons.

What will be pickled can be defined in `__getstate__` method. This method must return something that is pickleable.

相对应的是`__setstate__`：它将接收`__getstate__`创建的内容，并且必须初始化对象。

```
类 A(object):
    def __init__(self, important_data):
        self.important_data = important_data

        # 添加无法被pickle的数据：
        self.func = lambda: 7

        # 添加不应被序列化的数据，因为它们很快会过期：
        self.is_up_to_date = False

    def __getstate__(self):
        return [self.important_data] # 只需要这个

    def __setstate__(self, state):
        self.important_data = state[0]

        self.func = lambda: 7 # 只是一些硬编码的不可序列化函数

        self.is_up_to_date = False # 即使在序列化之前也是如此
```

现在，可以这样做：

```
>>> a1 = A('非常重要')
>>>
>>> s = pickle.dumps(a1) # 调用 a1.__getstate__()
>>>
>>> a2 = pickle.loads(s) # 调用 a1.__setstate__(['very important'])
>>> a2
<__main__.A 对象 位于 0x0000000002742470>
>>> a2.important_data
'非常重要'
>>> a2.func()
7
```

这里的实现是将一个包含一个值的列表进行pickle : `[self.important_data]`。这只是一个示例，`__getstate__`可以返回任何可被pickle的对象，只要`__setstate__` 知道如何执行相反操作。一个好的替代方案是返回包含所有值的字典：`{'important_data': self.important_data}`。

构造函数未被调用！ 注意在前面的示例中，实例`a2` 是通过`pickle.loads` 创建的，从未调用过`A.__init__`，因此`A.__setstate__` 必须初始化所有`__init__` 本应初始化的内容。

On the opposite side is `__setstate__`: it will receive what `__getstate__` created and has to initialize the object.

```
class A(object):
    def __init__(self, important_data):
        self.important_data = important_data

        # Add data which cannot be pickled:
        self.func = lambda: 7

        # Add data which should never be pickled, because it expires quickly:
        self.is_up_to_date = False

    def __getstate__(self):
        return [self.important_data] # only this is needed

    def __setstate__(self, state):
        self.important_data = state[0]

        self.func = lambda: 7 # just some hard-coded unpicklable function

        self.is_up_to_date = False # even if it was before pickling
```

Now, this can be done:

```
>>> a1 = A('very important')
>>>
>>> s = pickle.dumps(a1) # calls a1.__getstate__()
>>>
>>> a2 = pickle.loads(s) # calls a1.__setstate__(['very important'])
>>> a2
<__main__.A 对象 位于 0x0000000002742470>
>>> a2.important_data
'very important'
>>> a2.func()
7
```

The implementation here pikles a list with one value: `[self.important_data]`. That was just an example, `__getstate__` could have returned anything that is pickleable, as long as `__setstate__` knows how to do the opposite. A good alternative is a dictionary of all values: `{'important_data': self.important_data}`.

**Constructor is not called!** Note that in the previous example instance `a2` was created in `pickle.loads` without ever calling `A.__init__`, so `A.__setstate__` had to initialize everything that `__init__` would have initialized if it were called.

# 第113章：二进制数据

## 第113.1节：将值列表格式化为字节对象

```
from struct import pack  
  
print(pack('I3c', 123, b'a', b'b', b'c')) # b'{abc'
```

## 第113.2节：根据格式字符串解包字节对象

```
from struct import unpack  
  
print(unpack('I3c', b'{abc'))) # (123, b'a', b'b', b'c')
```

## 第113.3节：打包结构体

模块"struct"提供了将Python对象打包为连续字节块或将字节块拆解为Python结构体的功能。

pack函数接受一个格式字符串和一个或多个参数，返回一个二进制字符串。这看起来很像格式化字符串，但输出不是字符串而是一段字节块。

```
import struct  
import sys  
print "本机字节序: ", sys.byteorder  
# 如果未指定字节序，则使用本机字节序  
buffer = struct.pack("ihb", 3, 4, 5)  
print "字节块: ", repr(buffer)  
print "解包后的字节块: ", struct.unpack("ihb", buffer)  
# 最后一个元素作为无符号短整型而非无符号字符 (2字节)  
buffer = struct.pack("ihh", 3, 4, 5)  
print "字节块: ", repr(buffer)
```

输出：

```
本机字节序：小端 字节块：'■■■' 字节块解包：(3, 4, 5) 字节  
块：'■■■'
```

你可以使用网络字节序处理从网络接收的数据，或打包数据以发送到网络。

```
import struct  
# 如果未指定字节序，则使用本机字节序  
buffer = struct.pack("hhh", 3, 4, 5)  
print "字节块本机字节序: ", repr(buffer)  
buffer = struct.pack("!hhh", 3, 4, 5)  
print "字节块网络字节序: ", repr(buffer)
```

输出：

```
字节块本机字节序：'■■■'
```

# Chapter 113: Binary Data

## Section 113.1: Format a list of values into a byte object

```
from struct import pack  
  
print(pack('I3c', 123, b'a', b'b', b'c')) # b'{\x00\x00\x00abc'
```

## Section 113.2: Unpack a byte object according to a format string

```
from struct import unpack  
  
print(unpack('I3c', b'{\x00\x00\x00abc'))) # (123, b'a', b'b', b'c')
```

## Section 113.3: Packing a structure

The module "struct" provides facility to pack python objects as contiguous chunk of bytes or dissemble a chunk of bytes to python structures.

The pack function takes a format string and one or more arguments, and returns a binary string. This looks very much like you are formatting a string except that the output is not a string but a chunk of bytes.

```
import struct  
import sys  
print "Native byteorder: ", sys.byteorder  
# If no byteorder is specified, native byteorder is used  
buffer = struct.pack("ihb", 3, 4, 5)  
print "Byte chunk: ", repr(buffer)  
print "Byte chunk unpacked: ", struct.unpack("ihb", buffer)  
# Last element as unsigned short instead of unsigned char ( 2 Bytes)  
buffer = struct.pack("ihh", 3, 4, 5)  
print "Byte chunk: ", repr(buffer)
```

Output:

```
Native byteorder: little Byte chunk: '\x03\x00\x00\x00\x04\x00\x05' Byte chunk unpacked: (3, 4, 5) Byte  
chunk: '\x03\x00\x00\x00\x04\x00\x05\x00'
```

You could use network byte order with data received from network or pack data to send it to network.

```
import struct  
# If no byteorder is specified, native byteorder is used  
buffer = struct.pack("hhh", 3, 4, 5)  
print "Byte chunk native byte order: ", repr(buffer)  
buffer = struct.pack("!hhh", 3, 4, 5)  
print "Byte chunk network byte order: ", repr(buffer)
```

Output:

```
Byte chunk native byte order: '\x03\x00\x04\x00\x05\x00'
```

字节块网络字节序：'⊗⊗⊗'

你可以通过提供先前创建的缓冲区来避免分配新缓冲区的开销，从而进行优化。

```
import struct
from ctypes import create_string_buffer
bufferVar = create_string_buffer(8)
bufferVar2 = create_string_buffer(8)
# 我们使用已经创建的缓冲区
# 提供格式、缓冲区、偏移量和数据
struct.pack_into("hhh", bufferVar, 0, 3, 4, 5)
print "字节块: ", repr(bufferVar.raw)
struct.pack_into("hhh", bufferVar2, 2, 3, 4, 5)
print "字节块: ", repr(bufferVar2.raw)
```

输出：

字节块: '⊗⊗⊗'

字节块: '⊗⊗⊗'

Byte chunk network byte order: '\x00\x03\x00\x04\x00\x05'

You can optimize by avoiding the overhead of allocating a new buffer by providing a buffer that was created earlier.

```
import struct
from ctypes import create_string_buffer
bufferVar = create_string_buffer(8)
bufferVar2 = create_string_buffer(8)
# We use a buffer that has already been created
# provide format, buffer, offset and data
struct.pack_into("hhh", bufferVar, 0, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar.raw)
struct.pack_into("hhh", bufferVar2, 2, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar2.raw)
```

Output:

Byte chunk: '\x03\x00\x04\x00\x05\x00\x00\x00'

Byte chunk: '\x00\x00\x03\x00\x04\x00\x05\x00'

# 第114章：习语

## 第114.1节：字典键初始化

如果不确定键是否存在，建议使用 `dict.get`方法。它允许你在键未找到时返回默认值。传统方法 `dict[key]`会引发`KeyError`异常。

而不是这样做

```
def add_student():
    尝试:
        students['count'] += 1
    except KeyError:
        students['count'] = 1
```

执行

```
def add_student():
    students['count'] = students.get('count', 0) + 1
```

## 第114.2节：变量交换

要交换两个变量的值，可以使用元组拆包。

```
x = True
y = False
x, y = y, x
x
# False
y
# True
```

## 第114.3节：使用真值测试

Python 会隐式地将任何对象转换为布尔值进行测试，因此应尽可能使用它。

```
# 使用隐式真值测试的好例子
if attr:
    # 执行某些操作

if not attr:
    # 执行某些操作

# 不好的例子，使用具体类型判断
if attr == 1:
    # 执行某些操作

if attr == True:
    # 执行某些操作

if attr != '':
    # 执行某些操作

# 如果你想专门检查 None，使用 'is' 或 'is not'
if attr is None:
    # 执行某些操作
```

# Chapter 114: Idioms

## Section 114.1: Dictionary key initializations

Prefer `dict.get` method if you are not sure if the key is present. It allows you to return a default value if key is not found. The traditional method `dict[key]` would raise a `KeyError` exception.

Rather than doing

```
def add_student():
    尝试:
        students['count'] += 1
    except KeyError:
        students['count'] = 1
```

Do

```
def add_student():
    students['count'] = students.get('count', 0) + 1
```

## Section 114.2: Switching variables

To switch the value of two variables you can use tuple unpacking.

```
x = True
y = False
x, y = y, x
x
# False
y
# True
```

## Section 114.3: Use truth value testing

Python will implicitly convert any object to a Boolean value for testing, so use it wherever possible.

```
# Good examples, using implicit truth testing
if attr:
    # do something

if not attr:
    # do something

# Bad examples, using specific types
if attr == 1:
    # do something

if attr == True:
    # do something

if attr != '':
    # do something

# If you are looking to specifically check for None, use 'is' or 'is not'
if attr is None:
    # do something
```

这通常会生成更易读的代码，并且在处理意外类型时通常更安全。

[点击这里](#) 查看将被评估为False的列表。

## 第114.4节：测试“`__main__`”以避免意外代码执行

在执行代码之前，测试调用程序的`__name__`变量是一个好习惯。

导入 `sys` 定义

```
main():
    # 你的代码从这里开始

    # 不要忘记提供返回码
    返回0

if __name__ == "__main__":
    sys.exit(main())
```

使用此模式可确保代码仅在预期时执行；例如，当你明确运行你的文件时：

```
python my_program.py
```

但是，如果你决定在另一个程序中导入你的文件（例如，如果你正在将其作为库的一部分编写），这将带来好处。你可以导入你的文件，`__main__`陷阱将确保不会意外执行任何代码：

```
# 一个新的程序文件
import my_program      # main() 不会被运行

# 但如果你真的想运行，可以显式调用 main():
my_program.main()
```

This generally produces more readable code, and is usually much safer when dealing with unexpected types.

[Click here](#) for a list of what will be evaluated to `False`.

## Section 114.4: Test for "`__main__`" to avoid unexpected code execution

It is good practice to test the calling program's `__name__` variable before executing your code.

```
import sys

def main():
    # Your code starts here

    # Don't forget to provide a return code
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

Using this pattern ensures that your code is only executed when you expect it to be; for example, when you run your file explicitly:

```
python my_program.py
```

The benefit, however, comes if you decide to `import` your file in another program (for example if you are writing it as part of a library). You can then `import` your file, and the `__main__` trap will ensure that no code is executed unexpectedly:

```
# A new program file
import my_program      # main() is not run

# But you can run main() explicitly if you really want it to run:
my_program.main()
```

# 视频：机器学习 A-Z：动手 Python数据科学

学习如何从两位数据科学专家那里用Python创建机器学习算法。包含代码模板。



- ✓ 精通Python上的机器学习
- ✓ 对多种机器学习模型有良好的直觉
- ✓ 做出准确的预测
- ✓ 进行强有力的数据分析
- ✓ 制作稳健的机器学习模型
- ✓ 为您的业务创造强大的附加价值
- ✓ 将机器学习用于个人目的
- ✓ 处理强化学习、自然语言处理和深度学习等特定主题
- ✓ 掌握降维等高级技术
- ✓ 了解针对每种问题应选择哪种机器学习模型
- ✓ 构建一支强大的机器学习模型军团，并知道如何组合它们以解决任何问题

今天观看 →

# VIDEO: Machine Learning A-Z: Hands-On Python In Data Science

Learn to create Machine Learning Algorithms in Python from two Data Science experts. Code templates included.



- ✓ Master Machine Learning on Python
- ✓ Have a great intuition of many Machine Learning models
- ✓ Make accurate predictions
- ✓ Make powerful analysis
- ✓ Make robust Machine Learning models
- ✓ Create strong added value to your business
- ✓ Use Machine Learning for personal purpose
- ✓ Handle specific topics like Reinforcement Learning, NLP and Deep Learning
- ✓ Handle advanced techniques like Dimensionality Reduction
- ✓ Know which Machine Learning model to choose for each type of problem
- ✓ Build an army of powerful Machine Learning models and know how to combine them to solve any problem

Watch Today →

# 第115章：数据序列化

参数	详细信息
协议	使用pickle或cPickle时，对象的序列化/反序列化就是通过这种方法完成的。你可能想使用pickle.HIGHEST_PROTOCOL，这意味着使用最新的方法。

## 第115.1节：使用JSON进行序列化

JSON是一种跨语言、广泛使用的数据序列化方法

支持的数据类型：*int*、*float*、*boolean*、*string*、*list*和*dict*。详见 -> [JSON Wiki](#)

下面是一个演示JSON基本用法的示例：

```
import json

families = (['约翰'], ['马克', '大卫', {'姓名': '亚伯拉罕'}])

# 将其转储为字符串
json_families = json.dumps(families)
# [[{"name": "John"}, [{"name": "Mark", "name": "David"}, {"name": "Abraham"}]]]

# 将其转储到文件中
with open('families.json', 'w') as json_file:
    json.dump(families, json_file)

# 从字符串加载
json_families = json.loads(json_families)

# 从文件加载
with open('families.json', 'r') as json_file:
    json_families = json.load(json_file)
```

有关 JSON 的详细信息，请参见 [JSON-Module](#)。

## 第 115.2 节：使用 Pickle 进行序列化

下面是一个演示 basic 使用 pickle 的示例：

```
# 导入 pickle
尝试:
    import cPickle as pickle # Python 2
except ImportError:
    import pickle # Python 3

# 创建 Python 对象：
class Family(object):
    def __init__(self, names):
        self.sons = names

    def __str__(self):
        return '\n'.join(self.sons)

my_family = Family(['John', 'David'])

# 转储为字符串
pickle_data = pickle.dumps(my_family, pickle.HIGHEST_PROTOCOL)
```

# Chapter 115: Data Serialization

Parameter	Details
protocol	Using <code>pickle</code> or <code>cPickle</code> , it is the method that objects are being Serialized/Unserialized. You probably want to use <code>pickle.HIGHEST_PROTOCOL</code> here, which means the newest method.

## Section 115.1: Serialization using JSON

JSON is a cross language, widely used method to serialize data

Supported data types : *int*, *float*, *boolean*, *string*, *list* and *dict*. See -> [JSON Wiki](#) for more

Here is an example demonstrating the **basic** usage of **JSON**:

```
import json

families = ([ 'John' ], [ 'Mark' , 'David' , { 'name' : 'Avraham' }])

# Dumping it into string
json_families = json.dumps(families)
# [[{"name": "John"}, {"name": "Mark", "name": "David"}, {"name": "Avraham"}]]

# Dumping it to file
with open('families.json', 'w') as json_file:
    json.dump(families, json_file)

# Loading it from string
json_families = json.loads(json_families)

# Loading it from file
with open('families.json', 'r') as json_file:
    json_families = json.load(json_file)
```

See [JSON-Module](#) for detailed information about JSON.

## Section 115.2: Serialization using Pickle

Here is an example demonstrating the **basic** usage of **pickle**:

```
# Importing pickle
try:
    import cPickle as pickle # Python 2
except ImportError:
    import pickle # Python 3

# Creating Pythonic object:
class Family(object):
    def __init__(self, names):
        self.sons = names

    def __str__(self):
        return '\n'.join(self.sons)

my_family = Family(['John', 'David'])

# Dumping to string
pickle_data = pickle.dumps(my_family, pickle.HIGHEST_PROTOCOL)
```

```
# 转储到文件
with open('family.p', 'w') as pickle_file:
    pickle.dump(families, pickle_file, pickle.HIGHEST_PROTOCOL)

# 从字符串加载
my_family = pickle.loads(pickle_data)

# 从文件加载
with open('family.p', 'r') as pickle_file:
    my_family = pickle.load(pickle_file)
```

有关 Pickle 的详细信息，请参见 [Pickle](#)。

警告：官方的 `pickle` 文档明确指出没有安全保障。请勿加载任何你不信任来源的数据。

```
# Dumping to file
with open('family.p', 'w') as pickle_file:
    pickle.dump(families, pickle_file, pickle.HIGHEST_PROTOCOL)

# Loading from string
my_family = pickle.loads(pickle_data)

# Loading from file
with open('family.p', 'r') as pickle_file:
    my_family = pickle.load(pickle_file)
```

See [Pickle](#) for detailed information about Pickle.

**WARNING:** The official documentation for `pickle` makes it clear that there are no security guarantees. Don't load any data you don't trust its origin.

# 第116章：多进程

## 第116.1节：运行两个简单进程

使用多进程的一个简单示例是两个分别执行的进程（工作者）。在下面的示例中，启动了两个进程：

- countUp() 每秒计数加1。
- countDown() 每秒计数减1。

```
import multiprocessing
import time
from random import randint

def countUp():
    i = 0
    while i <= 3:
        print('Up:{}\n'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i += 1

def countDown():
    i = 3
    while i >= 0:
        print('Down:{}\n'.format(i))
        time.sleep(randint(1, 3)) # 睡眠1、2或3秒
        i -= 1

if __name__ == '__main__':
    # 初始化工作进程。
    workerUp = multiprocessing.Process(target=countUp)
    workerDown = multiprocessing.Process(target=countDown)

    # 启动工作进程。
    workerUp.start()
    workerDown.start()

    # 等待工作进程结束。这将在主（父）进程中阻塞，直到工作进程完成。
    workerUp.join()
    workerDown.join()
```

输出如下：

```
Up: 0
Down: 3
Up: 1
Up: 2
Down: 2
Up: 3
Down: 1
Down: 0
```

## 第116.2节：使用Pool和Map

```
from multiprocessing import Pool
```

# Chapter 116: Multiprocessing

## Section 116.1: Running Two Simple Processes

A simple example of using multiple processes would be two processes (workers) that are executed separately. In the following example, two processes are started:

- countUp() counts 1 up, every second.
- countDown() counts 1 down, every second.

```
import multiprocessing
import time
from random import randint

def countUp():
    i = 0
    while i <= 3:
        print('Up:\t{}\n'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i += 1

def countDown():
    i = 3
    while i >= 0:
        print('Down:\t{}\n'.format(i))
        time.sleep(randint(1, 3)) # sleep 1, 2 or 3 seconds
        i -= 1

if __name__ == '__main__':
    # Initiate the workers.
    workerUp = multiprocessing.Process(target=countUp)
    workerDown = multiprocessing.Process(target=countDown)

    # Start the workers.
    workerUp.start()
    workerDown.start()

    # Join the workers. This will block in the main (parent) process
    # until the workers are complete.
    workerUp.join()
    workerDown.join()
```

The output is as follows:

```
Up: 0
Down: 3
Up: 1
Up: 2
Down: 2
Up: 3
Down: 1
Down: 0
```

## Section 116.2: Using Pool and Map

```
from multiprocessing import Pool
```

```
def cube(x):
    return x ** 3

if __name__ == "__main__":
    pool = Pool(5)
    result = pool.map(cube, [0, 1, 2, 3])
```

Pool 是一个类，负责在后台管理多个Workers（进程），并让你，程序员，使用它们。

Pool(5) 创建一个包含5个进程的新Pool，pool.map 的工作方式与map类似，但它使用多个进程（数量由创建池时定义）。

类似的结果也可以通过map\_async、apply和apply\_async实现，相关内容可在[文档中找到](#)。

```
def cube(x):
    return x ** 3

if __name__ == "__main__":
    pool = Pool(5)
    result = pool.map(cube, [0, 1, 2, 3])
```

Pool is a class which manages multiple Workers (processes) behind the scenes and lets you, the programmer, use.

Pool(5) creates a new Pool with 5 processes, and pool.map works just like map but it uses multiple processes (the amount defined when creating the pool).

Similar results can be achieved using map\_async, apply and apply\_async which can be found in [the documentation](#).

# 第117章：多线程

线程允许Python程序同时处理多个功能，而不是单独运行一系列命令。本文介绍了线程的原理并演示了其用法。

## 第117.1节：多线程基础

使用`threading`模块，可以通过创建一个新的`threading.Thread`并分配一个函数来执行，从而启动一个新的执行线程：

```
import threading

def foo():
    print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

`target`参数引用要运行的函数（或可调用对象）。线程在调用`Thread`对象的`start`之前不会开始执行。

### 启动线程

```
my_thread.start() # 输出 'Hello threading!'
```

现在`my_thread`已经运行并终止，再次调用`start`将会产生`RuntimeError`。如果你想让线程作为守护线程运行，可以传入`daemon=True`关键字参数，或者在调用`start()`之前将`my_thread.daemon`设置为`True`，这样你的`Thread`会作为守护线程在后台静默运行。

### 等待线程结束

在将一个大任务拆分成多个小任务并希望同时运行它们，但需要等待所有任务完成后再继续时，`Thread.join()`是你需要的方法。

例如，假设你想下载一个网站的多个页面并将它们合并成一个页面。你可以这样做：

```
import requests
from threading import Thread
from queue import Queue

q = Queue(maxsize=20)
def put_page_to_q(page_num):
    q.put(requests.get('http://some-website.com/page_%s.html' % page_num))

def compile(q):
    # 需要所有页面后才能执行的魔法函数
    if not q.full():
        raise ValueError
    else:
        print("合并完成！")

threads = []
for page_num in range(20):
    t = Thread(target=requests.get, args=(page_num,))
    t.start()
    threads.append(t)
```

# Chapter 117: Multithreading

Threads allow Python programs to handle multiple functions at once as opposed to running a sequence of commands individually. This topic explains the principles behind threading and demonstrates its usage.

## Section 117.1: Basics of multithreading

Using the `threading` module, a new thread of execution may be started by creating a new `threading.Thread` and assigning it a function to execute:

```
import threading

def foo():
    print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

The `target` parameter references the function (or callable object) to be run. The thread will not begin execution until `start` is called on the `Thread` object.

### Starting a Thread

```
my_thread.start() # prints 'Hello threading!'
```

Now that `my_thread` has run and terminated, calling `start` again will produce a `RuntimeError`. If you'd like to run your thread as a daemon, passing the `daemon=True` kwarg, or setting `my_thread.daemon` to `True` before calling `start()`, causes your `Thread` to run silently in the background as a daemon.

### Joining a Thread

In cases where you split up one big job into several small ones and want to run them concurrently, but need to wait for all of them to finish before continuing, `Thread.join()` is the method you're looking for.

For example, let's say you want to download several pages of a website and compile them into a single page. You'd do this:

```
import requests
from threading import Thread
from queue import Queue

q = Queue(maxsize=20)
def put_page_to_q(page_num):
    q.put(requests.get('http://some-website.com/page_%s.html' % page_num))

def compile(q):
    # magic function that needs all pages before being able to be executed
    if not q.full():
        raise ValueError
    else:
        print("Done compiling!")

threads = []
for page_num in range(20):
    t = Thread(target=requests.get, args=(page_num,))
    t.start()
    threads.append(t)
```

```

# 接下来，加入所有线程以确保所有线程运行完毕后
# 我们才继续。join() 是一个阻塞调用（除非在调用 join 时使用
# 关键字参数 blocking=False 指定非阻塞）
for t in threads:
    t.join()

# 现在调用 compile()，因为所有线程都已完成
compile(q)

```

关于 join() 工作原理的详细介绍可以在 [here](#) 找到。

### 创建自定义线程类

使用threading.Thread类，我们可以子类化新的自定义线程类。我们必须在子类中重写 run方法。

```

from threading import Thread
import time

class Sleepy(Thread):

    def run(self):
        time.sleep(5)
        print("Hello from Thread")

if __name__ == "__main__":
    t = Sleepy()
    t.start()      # start方法会自动调用Thread类的run方法。
    # print 'The main program continues to run in foreground.'
    t.join()
    print("主程序继续在前台运行。")

```

## 第117.2节：线程间通信

你的代码中有多个线程，你需要在线程之间安全地通信。

你可以使用queue库中的Queue。

```

from queue import Queue
from threading import Thread

# 创建一个数据生产者
def producer(output_queue):
    while True:
        data = data_computation()
        output_queue.put(data)

# 创建一个消费者
def consumer(input_queue):
    while True:
        # 获取数据（阻塞）
        data = input_queue.get()
        # 使用数据
        # 表示数据已被消费
        input_queue.task_done()

```

```

# Next, join all threads to make sure all threads are done running before
# we continue. join() is a blocking call (unless specified otherwise using
# the kwarg blocking=False when calling join)
for t in threads:
    t.join()

# Call compile() now, since all threads have completed
compile(q)

```

A closer look at how join() works can be found [here](#).

### Create a Custom Thread Class

Using [threading](#).Thread class we can subclass new custom Thread class. we must override run method in a subclass.

```

from threading import Thread
import time

class Sleepy(Thread):

    def run(self):
        time.sleep(5)
        print("Hello from Thread")

    if __name__ == "__main__":
        t = Sleepy()
        t.start()      # start method automatic call Thread class run method.
        # print 'The main program continues to run in foreground.'
        t.join()
        print("The main program continues to run in the foreground.")

```

## Section 117.2: Communicating between threads

There are multiple threads in your code and you need to safely communicate between them.

You can use a [Queue](#) from the queue library.

```

from queue import Queue
from threading import Thread

# create a data producer
def producer(output_queue):
    while True:
        data = data_computation()
        output_queue.put(data)

# create a consumer
def consumer(input_queue):
    while True:
        # retrieve data (blocking)
        data = input_queue.get()
        # do something with the data
        # indicate data has been consumed
        input_queue.task_done()

```

```

q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()

```

## 第117.3节：创建工作线程池

使用threading和queue：

```

from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_server(addr, nworkers):
    print('回声服务器运行于', addr)
    # 启动客户端工作线程
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # 运行服务器
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('', 15000), 128)

```

使用 concurrent.futures.Threadpoolexecutor：

```

from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_server(addr):
    print('回声服务器运行于', addr)
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('', 15000))

```

*Python Cookbook, 第3版, 作者David Beazley和Brian K. Jones (O'Reilly)。版权所有2013年David Beazley和Brian Jones, ISBN 978-1-449-34037-7。*

## 第117.4节：多线程的高级用法

本节将包含一些使用多线程实现的最先进示例。

```

q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()

```

## Section 117.3: Creating a worker pool

Using `threading` & `queue`:

```

from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_server(addr, nworkers):
    print('Echo server running at', addr)
    # Launch the client workers
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('', 15000), 128)

```

Using `concurrent.futures.Threadpoolexecutor`:

```

from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_server(addr):
    print('Echo server running at', addr)
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('', 15000))

```

*Python Cookbook, 3rd edition, by David Beazley and Brian K. Jones (O'Reilly). Copyright 2013 David Beazley and Brian Jones, 978-1-449-34037-7.*

## Section 117.4: Advanced use of multithreads

This section will contain some of the most advanced examples realized using Multithreading.

## 高级打印机（日志记录器）

一个线程打印接收到的所有内容，并根据终端宽度修改输出。好处是当终端宽度变化时，“已写入”的输出也会被修改。

```
#!/usr/bin/env python2

import threading
import Queue
import time
import sys
import subprocess
from backports.shutil_get_terminal_size import get_terminal_size

printq = Queue.Queue()
interrupt = False
lines = []

def main():

    ptt = threading.Thread(target=printer) # 打开打印机
    ptt.daemon = True
    ptt.start()

    # 愚蠢的打印示例
    for i in xrange(1,100):
        printq.put(''.join([str(x) for x in range(1,i)]))      # 发送内容到打印机的实际方法
        time.sleep(.5)

def split_line(line, cols):
    if len(line) > cols:
        new_line = ''
        ww = line.split()
        i = 0
        while len(new_line) <= (cols - len(ww[i]) - 1):
            new_line += ww[i] + ' '
            i += 1
            print len(new_line)
        if new_line == '':
            return (line, '')
        else:
            return (new_line, ' '.join(ww[i:]))

    def printer():

        while True:
            cols, rows = get_terminal_size() # 获取终端尺寸
            msg = '#' + '-' * (cols - 2) +
            '#' # 创建
            尝试:
            new_line = str(printq.get_nowait())
            if new_line != '!@#EXIT#@!': # 一个优雅地关闭打印机线程的方法
                lines.append(new_line)
                printq.task_done()
            else:
                printq.task_done()
                sys.exit()

def printer():

    while True:
        cols, rows = get_terminal_size() # Get the terminal dimensions
        msg = '#' + '-' * (cols - 2) + '#\n' # Create the
        try:
            new_line = str(printq.get_nowait())
            if new_line != '!@#EXIT#@!': # A nice way to turn the printer
                                         # thread out gracefully
                lines.append(new_line)
                printq.task_done()
            else:
                printq.task_done()
                sys.exit()
```

## Advanced printer (logger)

A thread that prints everything is received and modifies the output according to the terminal width. The nice part is that also the "already written" output is modified when the width of the terminal changes.

```
#!/usr/bin/env python2

import threading
import Queue
import time
import sys
import subprocess
from backports.shutil_get_terminal_size import get_terminal_size

printq = Queue.Queue()
interrupt = False
lines = []

def main():

    ptt = threading.Thread(target=printer) # Turn the printer on
    ptt.daemon = True
    ptt.start()

    # Stupid example of stuff to print
    for i in xrange(1,100):
        printq.put(''.join([str(x) for x in range(1,i)]))      # The actual way to send stuff
        time.sleep(.5)

def split_line(line, cols):
    if len(line) > cols:
        new_line = ''
        ww = line.split()
        i = 0
        while len(new_line) <= (cols - len(ww[i]) - 1):
            new_line += ww[i] + ' '
            i += 1
            print len(new_line)
        if new_line == '':
            return (line, '')
        else:
            return (new_line, ' '.join(ww[i:]))

    def printer():

        while True:
            cols, rows = get_terminal_size() # Get the terminal dimensions
            msg = '#' + '-' * (cols - 2) + '#\n' # Create the
            try:
                new_line = str(printq.get_nowait())
                if new_line != '!@#EXIT#@!': # A nice way to turn the printer
                                             # thread out gracefully
                    lines.append(new_line)
                    printq.task_done()
                else:
                    printq.task_done()
                    sys.exit()
```

```

except Queue.Empty:
    通过

    # 构建要显示的新消息并拆分过长的行
    for line in lines:
        res = line      # 以下用于拆分长度超过列数的行。
        while len(res) !=0:
            toprint, res = split_line(res, cols)
            msg += " " + to
            print

            # 清除终端并打印新输出
            subprocess.check_call('clear') # 保持终端清洁
            sys.stdout.write(msg)
            sys.stdout.flush()
            time.sleep(.5)

```

## 第117.5节：带有while循环的可停止线程

```

import threading
import time

class StoppableThread(threading.Thread):
    """带有stop()方法的线程类。线程本身必须定期检查
    stopped()条件。"""

    def __init__(self):
        super(StoppableThread, self).__init__()
        self._stop_event = threading.Event()

    def stop(self):
        self._stop_event.set()

    def join(self, *args, **kwargs):
        self.stop()
        super(StoppableThread, self).join(*args, **kwargs)

    def run(self):
        while not self._stop_event.is_set():
            print("Still running!")
            time.sleep(2)
        print("stopped!")

```

基于这个问题。[\\_\\_\\_\\_\\_](#)

```

except Queue.Empty:
    pass

    # Build the new message to show and split too long lines
    for line in lines:
        res = line      # The following is to split lines which are
        # longer than cols.
        while len(res) !=0:
            toprint, res = split_line(res, cols)
            msg += '\n' + toprint

            # Clear the shell and print the new output
            subprocess.check_call('clear') # Keep the shell clean
            sys.stdout.write(msg)
            sys.stdout.flush()
            time.sleep(.5)

```

## Section 117.5: Stoppable Thread with a while Loop

```

import threading
import time

class StoppableThread(threading.Thread):
    """Thread class with a stop() method. The thread itself has to check
    regularly for the stopped() condition."""

    def __init__(self):
        super(StoppableThread, self).__init__()
        self._stop_event = threading.Event()

    def stop(self):
        self._stop_event.set()

    def join(self, *args, **kwargs):
        self.stop()
        super(StoppableThread, self).join(*args, **kwargs)

    def run(self):
        while not self._stop_event.is_set():
            print("Still running!")
            time.sleep(2)
        print("stopped!")

```

Based on [this Question.](#)

# 第118章：进程与线程

大多数程序是逐行执行的，一次只运行一个进程。线程允许多个进程相互独立地运行。多处理器的线程允许程序同时运行多个进程。  
本章节记录了Python中线程的实现和使用。

## 第118.1节：全局解释器锁

Python多线程性能常常受到全局解释器锁的影响。简而言之，尽管Python程序中可以有多个线程，但在任何时刻，只有一个字节码指令可以并行执行，无论CPU数量多少。

因此，在操作被外部事件阻塞（如网络访问）的情况下，多线程可以非常有效：

```
import threading
import time

def process():
    time.sleep(2)

start = time.time()
process()
print("一次运行耗时 %.2fs" % (time.time() - start))

start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("四次运行耗时 %.2fs" % (time.time() - start))

# 输出：一次运行耗时 2.00秒
# 输出：四次运行耗时 2.00秒
```

注意，尽管每个进程执行耗时2秒，四个进程一起能够有效地并行运行，总共耗时2秒。

然而，在Python代码中进行大量计算时使用多线程——例如大量计算——并不会带来太大提升，甚至可能比并行运行更慢：

```
import threading
import time

def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
```

# Chapter 118: Processes and Threads

Most programs are executed line by line, only running a single process at a time. Threads allow multiple processes to flow independent of each other. Threading with multiple processors permits programs to run multiple processes simultaneously. This topic documents the implementation and usage of threads in Python.

## Section 118.1: Global Interpreter Lock

Python multithreading performance can often suffer due to the [Global Interpreter Lock](#). In short, even though you can have multiple threads in a Python program, only one bytecode instruction can execute in parallel at any one time, regardless of the number of CPUs.

As such, multithreading in cases where operations are blocked by external events - like network access - can be quite effective:

```
import threading
import time

def process():
    time.sleep(2)

start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))

start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.00
# Out: Four runs took 2.00
```

Note that even though each process took 2 seconds to execute, the four processes together were able to effectively run in parallel, taking 2 seconds total.

However, multithreading in cases where intensive computations are being done in Python code - such as a lot of computation - does not result in much improvement, and can even be slower than running in parallel:

```
import threading
import time

def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
```

```
result = 0
    for i in range(100000):
result = otherfunc(result, somefunc(i))
```

```
start = time.time()
process()
print("一次运行耗时 %.2fs" % (time.time() - start))
```

```
start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("四次运行耗时 %.2fs" % (time.time() - start))
```

```
# 输出：一次运行耗时2.05秒
# 输出：四次运行耗时14.42秒
```

在后一种情况下，多进程可以发挥作用，因为多个进程当然可以同时执行多条指令：

```
import multiprocessing
import time
```

```
def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
result = 0
    for i in range(100000):
result = otherfunc(result, somefunc(i))
```

```
start = time.time()
process()
print("一次运行耗时 %.2fs" % (time.time() - start))
```

```
start = time.time()
processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()
print("四次运行耗时 %.2fs" % (time.time() - start))
```

```
# 输出：一次运行耗时2.07秒
# 输出：四次运行耗时2.30秒
```

## 第118.2节：多线程运行

使用`threading.Thread`在另一个线程中运行函数。

```
result = 0
for i in range(100000):
    result = otherfunc(result, somefunc(i))
```

```
start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))
```

```
start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))
```

```
# Out: One run took 2.05s
# Out: Four runs took 14.42s
```

In the latter case, multiprocessing can be effective as multiple processes can, of course, execute multiple instructions simultaneously:

```
import multiprocessing
import time
```

```
def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
result = 0
    for i in range(100000):
result = otherfunc(result, somefunc(i))
```

```
start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))
```

```
start = time.time()
processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()
print("Four runs took %.2fs" % (time.time() - start))
```

```
# Out: One run took 2.07s
# Out: Four runs took 2.30s
```

## Section 118.2: Running in Multiple Threads

Use `threading.Thread` to run a function in another thread.

```

import threading
import os

def process():
    print("进程ID是 %s, 线程ID是 %s" % (os.getpid(), threading.current_thread().name))

threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()

# 输出: 进程ID是 11240, 线程ID是 Thread-1
# 输出: 进程ID是 11240, 线程ID是 Thread-2
# 输出: 进程ID是 11240, 线程ID是 Thread-3
# 输出: 进程ID是 11240, 线程ID是 Thread-4

```

## 第118.3节：在多个进程中运行

使用multiprocessing.Process在另一个进程中运行函数。接口与threading.Thread类似：

```

import multiprocessing
import os

def process():
    print("进程ID是 %s" % (os.getpid(),))

processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()

# 输出: 进程ID是 11206
# 输出: 进程ID是 11207
# 输出: 进程ID是 11208
# 输出: 进程ID是 11209

```

## 第118.4节：线程间共享状态

由于所有线程都运行在同一进程中，所有线程都可以访问相同的数据。

但是，对共享数据的并发访问应使用锁进行保护，以避免同步问题。

```

import threading

obj = {}
obj_lock = threading.Lock()

def objify(key, val):
    print("Obj 有 %d 个值" % len(obj))
    with obj_lock:
        obj[key] = val
    print("Obj 现在有 %d 个值" % len(obj))

ts = [threading.Thread(target=objify, args=(str(n), n)) for n in range(4)]
对于 t 在 ts 中：
    t.start()
对于 t 在 ts 中：

```

```

import threading
import os

def process():
    print("Pid is %s, thread id is %s" % (os.getpid(), threading.current_thread().name))

threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()

# Out: Pid is 11240, thread id is Thread-1
# Out: Pid is 11240, thread id is Thread-2
# Out: Pid is 11240, thread id is Thread-3
# Out: Pid is 11240, thread id is Thread-4

```

## Section 118.3: Running in Multiple Processes

Use multiprocessing.Process to run a function in another process. The interface is similar to `threading.Thread`:

```

import multiprocessing
import os

def process():
    print("Pid is %s" % (os.getpid(),))

processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()

# Out: Pid is 11206
# Out: Pid is 11207
# Out: Pid is 11208
# Out: Pid is 11209

```

## Section 118.4: Sharing State Between Threads

As all threads are running in the same process, all threads have access to the same data.

However, concurrent access to shared data should be protected with a lock to avoid synchronization issues.

```

import threading

obj = {}
obj_lock = threading.Lock()

def objify(key, val):
    print("Obj has %d values" % len(obj))
    with obj_lock:
        obj[key] = val
    print("Obj now has %d values" % len(obj))

ts = [threading.Thread(target=objify, args=(str(n), n)) for n in range(4)]
for t in ts:
    t.start()
for t in ts:

```

```

t.join()
print("最终对象结果:")
import pprint; pprint.pprint(obj)

# 输出: 对象包含0个值
# 输出: 对象包含0个值
# 输出: 对象现在包含1个值
# 输出: 对象现在包含2个值对象包含2个值
# 输出: 对象现在包含3个值
# 输出:
# 输出: 对象包含3个值
# 输出: 对象现在包含4个值
# 输出: 对象最终结果:
# 输出: {'0': 0, '1': 1, '2': 2, '3': 3}

```

## 第118.5节：进程间状态共享

默认情况下，不同进程中运行的代码不会共享相同的数据。然而，`multiprocessing` 模块包含一些原语，帮助在多个进程间共享数值。

```

import multiprocessing

plain_num = 0
shared_num = multiprocessing.Value('d', 0)
lock = multiprocessing.Lock()

def increment():
    global plain_num
    with lock:
        # 普通变量的修改在进程间不可见
    plain_num += 1
    # multiprocessing.Value 的修改是可见的
    shared_num.value += 1

ps = [multiprocessing.Process(target=increment) for n in range(4)]
for p in ps:
    p.start()
for p in ps:
    p.join()

print("plain_num is %d, shared_num is %d" % (plain_num, shared_num.value))

# 输出: plain_num 是 0, shared_num 是 4

```

```

t.join()
print("Obj final result:")
import pprint; pprint.pprint(obj)

# Out: Obj has 0 values
# Out: Obj has 0 values
# Out: Obj now has 1 values
# Out: Obj now has 2 valuesObj has 2 values
# Out: Obj now has 3 values
# Out:
# Out: Obj has 3 values
# Out: Obj now has 4 values
# Out: Obj final result:
# Out: {'0': 0, '1': 1, '2': 2, '3': 3}

```

## Section 118.5: Sharing State Between Processes

Code running in different processes do not, by default, share the same data. However, the `multiprocessing` module contains primitives to help share values across multiple processes.

```

import multiprocessing

plain_num = 0
shared_num = multiprocessing.Value('d', 0)
lock = multiprocessing.Lock()

def increment():
    global plain_num
    with lock:
        # ordinary variable modifications are not visible across processes
    plain_num += 1
    # multiprocessing.Value modifications are
    shared_num.value += 1

ps = [multiprocessing.Process(target=increment) for n in range(4)]
for p in ps:
    p.start()
for p in ps:
    p.join()

print("plain_num is %d, shared_num is %d" % (plain_num, shared_num.value))

# Out: plain_num is 0, shared_num is 4

```

# 第119章：Python并发

## 第119.1节：multiprocessing模块

```
from __future__ import print_function
import multiprocessing

def 倒计时(count):
    while count > 0:
        print("计数值", count)
        count -= 1
    return

if __name__ == "__main__":
    p1 = multiprocessing.Process(target=倒计时, args=(10,))
    p1.start()

    p2 = multiprocessing.Process(target=倒计时, args=(20,))
    p2.start()

    p1.join()
    p2.join()
```

这里，每个函数都在一个新的进程中执行。由于运行代码的是Python虚拟机的新实例，没有 GIL，因此可以在多个核心上实现并行。

`Process.start` 方法启动这个新进程，并运行传递给 `target` 参数的函数以及 `args` 参数中的参数。`Process.join` 方法等待进程 `p1` 和 `p2` 执行结束。

新进程的启动方式取决于 Python 版本和代码运行的平台，例如：

- Windows 使用 `spawn` 来创建新进程。
- 在 Unix 系统和 3.3 版本之前，进程是通过 `fork` 创建的。  
注意，该方法不遵循 POSIX 对 `fork` 的使用，因此会导致意外行为，尤其是在与其他多进程库交互时。
- 在 Unix 系统和 3.4 及以上版本中，可以在程序开始时使用 `multiprocessing.set_start_method` 选择以 `fork`、`forkserver` 或 `spawn` 启动新进程。`forkserver` 和 `spawn` 方法比 `fork` 慢，但避免了一些意外行为。

### POSIX fork 使用：

在多线程程序中 `fork` 后，子进程只能安全调用异步信号安全函数，直到调用 `execve` 为止。

(见)

使用 `fork` 时，新进程将以所有当前互斥锁完全相同的状态启动，但仅限于 `MainThread` 将被启动。这是不安全的，因为可能导致竞态条件例如：

- 如果你在主线程（`MainThread`）中使用了一个锁（`Lock`），并将其传递给另一个线程，该线程在某个时刻应该锁定它。如果在此时发生了`fork`，新进程将以一个已锁定的锁开始，而这个锁永远不会被释放，因为第二个线程在这个新进程中并不存在。

# Chapter 119: Python concurrency

## Section 119.1: The multiprocessing module

```
from __future__ import print_function
import multiprocessing

def countdown(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

if __name__ == "__main__":
    p1 = multiprocessing.Process(target=countdown, args=(10,))
    p1.start()

    p2 = multiprocessing.Process(target=countdown, args=(20,))
    p2.start()

    p1.join()
    p2.join()
```

Here, each function is executed in a new process. Since a new instance of Python VM is running the code, there is no GIL and you get parallelism running on multiple cores.

The `Process.start` method launches this new process and run the function passed in the `target` argument with the arguments `args`. The `Process.join` method waits for the end of the execution of processes `p1` and `p2`.

The new processes are launched differently depending on the version of python and the platform on which the code is running e.g.:

- Windows uses `spawn` to create the new process.
- With unix systems and version earlier than 3.3, the processes are created using a `fork`. Note that this method does not respect the POSIX usage of `fork` and thus leads to unexpected behaviors, especially when interacting with other multiprocessing libraries.
- With unix system and version 3.4+, you can choose to start the new processes with either `fork`, `forkserver` or `spawn` using `multiprocessing.set_start_method` at the beginning of your program. `forkserver` and `spawn` methods are slower than forking but avoid some unexpected behaviors.

### POSIX fork usage:

After a fork in a multithreaded program, the child can safely call only async-signal-safe functions until such time as it calls `execve`.

(see)

Using `fork`, a new process will be launched with the exact same state for all the current mutex but only the `MainThread` will be launched. This is unsafe as it could lead to race conditions e.g.:

- If you use a `Lock` in `MainThread` and pass it to another thread which is supposed to lock it at some point. If the `fork` occurs simultaneously, the new process will start with a locked lock which will never be released as the second thread does not exist in this new process.

实际上，这种行为在纯Python中不应该发生，因为多进程（multiprocessing）模块会正确处理它，但如果你与其他库交互，这种行为可能会发生，导致系统崩溃（例如在macOS上使用numpy/accelerated时）。

## 第119.2节：线程模块

```
from __future__ import print_function
import threading
def counter(count):
    while count > 0:
        print("计数值", count)
        count -= 1
    return

t1 = threading.Thread(target=countdown, args=(10,))
t1.start()
t2 = threading.Thread(target=countdown, args=(20,))
t2.start()
```

在某些Python实现中，如CPython，由于使用了所谓的全局解释器锁（Global Interpreter Lock），线程并不能实现真正的并行。

这里有一个关于Python并发的优秀概述：

[David Beazley 的 Python 并发 \(YouTube\)](#)

## 第119.3节：在多进程之间传递数据

由于在两个线程之间处理数据是敏感的（考虑并发读取和并发写入可能相互冲突，导致竞态条件），因此创建了一组独特的对象以便在线程之间来回传递数据。任何真正的原子操作都可以在线程之间使用，但始终使用 Queue 是安全的。

```
import multiprocessing
import queue
my_Queue=multiprocessing.Queue()
# 创建一个最大大小未定义的队列
# 这可能很危险，因为队列会变得越来越大
# 复制数据到/从每个读写线程将花费很长时间
```

大多数人建议在使用队列时，总是将队列数据放在 try: except: 块中，而不是使用 empty。然而，对于那些跳过扫描周期无关紧要的应用（数据可以在队列状态从queue.Empty==True切换到queue.Empty==False时放入队列），通常更好将读写访问放在我称之为 Iftry 块中，因为从技术上讲，'if' 语句的性能优于捕获异常。

```
import multiprocessing
import queue
'''导入必要的 Python 标准库，multiprocessing 用于类，queue 用于其提供的队列异常'''

def Queue_Iftry_Get(get_queue, default=None, use_default=False, func=None, use_func=False):
    '''此 Iftry 块的全局方法提供其重用和标准功能，if 语句相比捕获异常（代价高昂）也能提升性能。'''

    它还允许用户为传出的数据指定一个函数，
```

Actually, this kind of behavior should not occur in pure python as multiprocessing handles it properly but if you are interacting with other library, this kind of behavior can occur, leading to crash of your system (for instance with numpy/accelerated on macOS).

## Section 119.2: The threading module

```
from __future__ import print_function
import threading
def counter(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

t1 = threading.Thread(target=countdown, args=(10,))
t1.start()
t2 = threading.Thread(target=countdown, args=(20,))
t2.start()
```

In certain implementations of Python such as CPython, true parallelism is not achieved using threads because of using what is known as the GIL, or Global Interpreter Lock.

Here is an excellent overview of Python concurrency:

[Python concurrency by David Beazley \(YouTube\)](#)

## Section 119.3: Passing data between multiprocessing processes

Because data is sensitive when dealt with between two threads (think concurrent read and concurrent write can conflict with one another, causing race conditions), a set of unique objects were made in order to facilitate the passing of data back and forth between threads. Any truly atomic operation can be used between threads, but it is always safe to stick with Queue.

```
import multiprocessing
import queue
my_Queue=multiprocessing.Queue()
#Creates a queue with an undefined maximum size
#this can be dangerous as the queue becomes increasingly large
#it will take a long time to copy data to/from each read/write thread
```

Most people will suggest that when using queue, to always place the queue data in a try: except: block instead of using empty. However, for applications where it does not matter if you skip a scan cycle (data can be placed in the queue while it is flipping states from queue.Empty==True to queue.Empty==False) it is usually better to place read and write access in what I call an Iftry block, because an 'if' statement is technically more performant than catching the exception.

```
import multiprocessing
import queue
'''Import necessary Python standard libraries, multiprocessing for classes and queue for the queue exceptions it provides'''
def Queue_Iftry_Get(get_queue, default=None, use_default=False, func=None, use_func=False):
    '''This global method for the Iftry block is provided for its reuse and standard functionality, the if also saves on performance as opposed to catching the exception, which is expensive.

    It also allows the user to specify a function for the outgoing data to use,
```

以及当函数无法从队列中返回值时的默认返回值

```
如果 get_queue.empty():
    如果 使用默认值:
        返回 默认值
    否则:
        尝试:
值 = get_queue.get_nowait()
    异常 queue.Empty:
        如果 使用默认值:
            返回 默认值
    否则:
        如果 使用函数:
            返回 func(值)
        否则:
            返回 值
```

```
def Queue_Iftry_Put(put_queue, value):
    '''此全局方法用于Iftry块，因其可重用性
```

及

标准功能，If语句相比捕获异常（代价较高）还能提升性能。

如果成功将值放入队列则返回True，否则返回False''

```
if put_queue.full():
    return False
else:
    尝试:
        put_queue.put_nowait(value)
except queue.Full:
    return False
else:
    return True
```

and a default value to return if the function cannot return the value from the queue'''

```
if get_queue.empty():
    if use_default:
        return default
else:
    try:
        value = get_queue.get_nowait()
    except queue.Empty:
        if use_default:
            return default
    else:
        if use_func:
            return func(value)
        else:
            return value
def Queue_Iftry_Put(put_queue, value):
    '''This global method for the Iftry block is provided because of its reuse
and
standard functionality, the If also saves on performance as opposed to catching
the exception, which is expensive.
Return True if placing value in the queue was successful. Otherwise, false'''
if put_queue.full():
    return False
else:
    try:
        put_queue.put_nowait(value)
    except queue.Full:
        return False
    else:
        return True
```

# 视频：机器学习、数据科学和使用 Python 的深度学习

完整的动手机器学习教程，涵盖  
数据科学、Tensorflow、人工智能  
和神经网络



- ✓ 使用Tensorflow和Keras构建人工神经网络
- ✓ 使用深度学习对图像、数据和情感进行分类
- ✓ 使用线性回归、多项式回归和多元回归进行预测
- ✓ 使用Matplotlib和Seaborn进行数据可视化
- ✓ 使用 Apache Spark 的 MLLib 实现大规模机器学习
- ✓ 理解强化学习——以及如何构建一个吃豆人机器人
- ✓ 使用 K-Means 聚类、支持向量机 (SVM) 、KNN、决策树、朴素贝叶斯和主成分分析 (PCA) 对数据进行分类
- ✓ 使用训练/测试和 K 折交叉验证来选择和调整模型
- ✓ 使用基于物品和基于用户的协同过滤构建电影推荐系统

今天观看 →

# VIDEO: Machine Learning, Data Science and Deep Learning with Python

Complete hands-on machine learning tutorial with data science, Tensorflow, artificial intelligence, and neural networks



- ✓ Build artificial neural networks with Tensorflow and Keras
- ✓ Classify images, data, and sentiments using deep learning
- ✓ Make predictions using linear regression, polynomial regression, and multivariate regression
- ✓ Data Visualization with Matplotlib and Seaborn
- ✓ Implement machine learning at massive scale with Apache Spark's MLLib
- ✓ Understand reinforcement learning - and how to build a Pac-Man bot
- ✓ Classify data using K-Means clustering, Support Vector Machines (SVM), KNN, Decision Trees, Naive Bayes, and PCA
- ✓ Use train/test and K-Fold cross validation to choose and tune your models
- ✓ Build a movie recommender system using item-based and user-based collaborative filtering

Watch Today →

# 第120章：并行计算

## 第120.1节：使用multiprocessing模块实现任务并行化

```
import multiprocessing

def fib(n):
    """以低效方式计算斐波那契数列
    目的是为了减慢CPU速度。"""
    if n <= 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
p = multiprocessing.Pool()
print(p.map(fib,[38,37,36,35,34,33]))

# 输出: [39088169, 24157817, 14930352, 9227465, 5702887, 3524578]
```

由于每次调用fib都是并行执行的，整个示例的执行时间比在双处理器上顺序执行快1.8倍。

Python 2.2+

## 第120.2节：使用C扩展实现任务并行化

这里的想法是将计算密集型任务移到C语言（使用特殊宏），独立于Python，并且让C代码在运行时释放全局解释器锁（GIL）。

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // 多线程C代码
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

## 第120.3节：使用父脚本和子脚本并行执行代码

child.py

```
import time

def main():
    print "开始工作"
    time.sleep(1)
    print "work work work work work"
    time.sleep(1)
    print "done working"

if __name__ == '__main__':
    main()
```

# Chapter 120: Parallel computation

## Section 120.1: Using the multiprocessing module to parallelise tasks

```
import multiprocessing

def fib(n):
    """computing the Fibonacci in an inefficient way
    was chosen to slow down the CPU."""
    if n <= 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
p = multiprocessing.Pool()
print(p.map(fib,[38,37,36,35,34,33]))

# Out: [39088169, 24157817, 14930352, 9227465, 5702887, 3524578]
```

As the execution of each call to fib happens in parallel, the time of execution of the full example is **1.8× faster** than if done in a sequential way on a dual processor.

Python 2.2+

## Section 120.2: Using a C-extension to parallelize tasks

The idea here is to move the computationally intensive jobs to C (using special macros), independent of Python, and have the C code release the GIL while it's working.

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

## Section 120.3: Using Parent and Children scripts to execute code in parallel

child.py

```
import time

def main():
    print "starting work"
    time.sleep(1)
    print "work work work work work"
    time.sleep(1)
    print "done working"

if __name__ == '__main__':
    main()
```

## parent.py

```
import os

def main():
    for i in range(5):
        os.system("python child.py &")

if __name__ == '__main__':
    main()
```

这对于并行、独立的HTTP请求/响应任务或数据库的查询/插入非常有用。命令行参数也可以传递给child.py脚本。脚本之间的同步可以通过所有脚本定期检查一个独立的服务器（例如Redis实例）来实现。

## 第120.4节：使用PyPar模块实现并行化

PyPar是一个使用消息传递接口（MPI）来提供Python并行性的库。PyPar中的一个简单示例（见<https://github.com/daleroberts/pypar>）如下：

```
import pypar as pp

ncpus = pp.size()
rank = pp.rank()
node = pp.get_processor_name()

print '我在节点 %s 上的排名是 %d / %d' % (rank, ncpus, node)

if rank == 0:
    msh = 'P0'
    pp.send(msg, destination=1)
    msg = pp.receive(source=rank-1)
    print '处理器 0 从排名 %d 收到消息 "%s"' % (rank-1, msg)
else:
    source = rank-1
    destination = (rank+1) % ncpus
    msg = pp.receive(source)
    msg = msg + 'P' + str(rank)
    pypar.send(msg, destination)
pp.finalize()
```

## parent.py

```
import os

def main():
    for i in range(5):
        os.system("python child.py &")

if __name__ == '__main__':
    main()
```

This is useful for parallel, independent HTTP request/response tasks or Database select/inserts. Command line arguments can be given to the `child.py` script as well. Synchronization between scripts can be achieved by all scripts regularly checking a separate server (like a Redis instance).

## Section 120.4: Using PyPar module to parallelize

PyPar is a library that uses the message passing interface (MPI) to provide parallelism in Python. A simple example in PyPar (as seen at <https://github.com/daleroberts/pypar>) looks like this:

```
import pypar as pp

ncpus = pp.size()
rank = pp.rank()
node = pp.get_processor_name()

print 'I am rank %d of %d on node %s' % (rank, ncpus, node)

if rank == 0:
    msh = 'P0'
    pp.send(msg, destination=1)
    msg = pp.receive(source=rank-1)
    print 'Processor 0 received message "%s" from rank %d' % (msg, rank-1)
else:
    source = rank-1
    destination = (rank+1) % ncpus
    msg = pp.receive(source)
    msg = msg + 'P' + str(rank)
    pypar.send(msg, destination)
pp.finalize()
```

# 第121章：套接字

参数	描述
socket.AF_UNIX	UNIX Socket
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	TCP
socket.SOCK_DGRAM	UDP

许多编程语言使用套接字在进程间或设备间进行通信。本主题解释了如何正确使用Python中的sockets模块，以便通过常见网络协议发送和接收数据。

## 第121.1节：Linux上的原始套接字

首先禁用网卡的自动校验和功能：

```
sudo ethtool -K eth1 tx off
```

然后使用SOCK\_RAW套接字发送数据包：

```
#!/usr/bin/env python
from socket import socket, AF_PACKET, SOCK_RAW
s = socket(AF_PACKET, SOCK_RAW)
s.bind(("eth1", 0))

# 我们正在组装一个以太网帧,
# 但你也可以组装任何你想要的内容
# 可以查看 'struct' 模块以获得更
# 灵活的二进制数据打包/解包
# 以及 'binascii' 用于32位CRC校验
src_addr = "\x01\x02\x03\x04\x05\x06"
dst_addr = "\x01\x02\x03\x04\x05\x06"
payload = ("[" * 30) + "PAYLOAD" + ("]" * 30)
checksum = "\x1a\x2b\x3c\x4d"
ethertype = "\x08\x06"

s.send(dst_addr+src_addr+ethertype+payload+checksum)
```

## 第121.2节：通过UDP发送数据

UDP是一种无连接协议。向其他进程或计算机发送消息时，不需要建立任何形式的连接。消息是否被接收没有自动确认。UDP通常用于对延迟敏感的应用或发送网络广播的应用。

下面的代码使用UDP向监听本地主机6667端口的进程发送消息

请注意，发送后无需“关闭”套接字，因为UDP是无连接的。

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
msg = ("Hello you there!").encode('utf-8') # socket.sendto() 接受字节作为输入，因此我们必须先编码字符串。
s.sendto(msg, ('localhost', 6667))
```

# Chapter 121: Sockets

Parameter	Description
socket.AF_UNIX	UNIX Socket
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	TCP
socket.SOCK_DGRAM	UDP

Many programming languages use sockets to communicate across processes or between devices. This topic explains proper usage the sockets module in Python to facilitate sending and receiving data over common networking protocols.

## Section 121.1: Raw Sockets on Linux

First you disable your network card's automatic checksumming:

```
sudo ethtool -K eth1 tx off
```

Then send your packet, using a SOCK\_RAW socket:

```
#!/usr/bin/env python
from socket import socket, AF_PACKET, SOCK_RAW
s = socket(AF_PACKET, SOCK_RAW)
s.bind(("eth1", 0))

# We're putting together an ethernet frame here,
# but you could have anything you want instead
# Have a look at the 'struct' module for more
# flexible packing/unpacking of binary data
# and 'binascii' for 32 bit CRC
src_addr = "\x01\x02\x03\x04\x05\x06"
dst_addr = "\x01\x02\x03\x04\x05\x06"
payload = ("[" * 30) + "PAYLOAD" + ("]" * 30)
checksum = "\x1a\x2b\x3c\x4d"
ethertype = "\x08\x06"

s.send(dst_addr+src_addr+ethertype+payload+checksum)
```

## Section 121.2: Sending data via UDP

UDP is a connectionless protocol. Messages to other processes or computers are sent without establishing any sort of connection. There is no automatic confirmation if your message has been received. UDP is usually used in latency sensitive applications or in applications sending network wide broadcasts.

The following code sends a message to a process listening on localhost port 6667 using UDP

Note that there is no need to "close" the socket after the send, because UDP is [connectionless](#).

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
msg = ("Hello you there!").encode('utf-8') # socket.sendto() takes bytes as input, hence we must
# encode the string first.
s.sendto(msg, ('localhost', 6667))
```

## 第121.3节：通过UDP接收数据

UDP是一种无连接协议。这意味着发送消息的对等方在发送消息之前不需要建立连接。`socket.recvfrom`因此返回一个元组（`msg` [套接字接收到的消息],`addr` [发送方的地址]）

仅使用 `socket` 模块的 UDP 服务器：

```
from socket import socket, AF_INET, SOCK_DGRAM
sock = socket(AF_INET, SOCK_DGRAM)
sock.bind(('localhost', 6667))

while True:
    msg, addr = sock.recvfrom(8192) # 这是最大读取字节数
    print("从 %s 收到消息: %s" % (addr, msg))
```

以下是使用 `socketserver.UDPServer` 的另一种实现方式：

```
from socketserver import BaseRequestHandler, UDPServer

class MyHandler(BaseRequestHandler):
    def handle(self):
        print("收到来自: %s 的连接" % self.client_address)
        msg, sock = self.request
        print("内容是: %s" % msg)
        sock.sendto("收到你的消息!".encode(), self.client_address) # 发送回复

serv = UDPServer(('localhost', 6667), MyHandler)
serv.serve_forever()
```

默认情况下，sockets是阻塞的。这意味着脚本的执行会等待，直到socket接收到数据。

## 第121.4节：通过TCP发送数据

通过多个模块可以实现通过互联网发送数据。sockets模块提供了对底层操作系统操作的低级访问，这些操作负责从其他计算机或进程发送或接收数据。

以下代码将字节字符串b'Hello'发送到监听本地主机6667端口的TCP服务器，完成后关闭连接：

```
from socket import socket, AF_INET, SOCK_STREAM
s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 6667)) # 监听的TCP服务器地址
s.send(b'Hello')
s.close()
```

Socket输出默认是阻塞的，这意味着程序将在connect和send调用中等待，直到操作“完成”。对于connect来说，这意味着服务器实际接受了连接。对于send来说，这仅意味着操作系统有足够的缓冲区空间来排队稍后发送的数据。

Socket使用后应始终关闭。

## 第121.5节：多线程TCP Socket服务器

当无参数运行时，该程序启动一个TCP Socket服务器，监听127.0.0.1的连接

## Section 121.3: Receiving data via UDP

UDP is a connectionless protocol. This means that peers sending messages do not require establishing a connection before sending messages. `socket.recvfrom` thus returns a tuple (`msg` [the message the socket received], `addr` [the address of the sender])

A UDP server using solely the `socket` module:

```
from socket import socket, AF_INET, SOCK_DGRAM
sock = socket(AF_INET, SOCK_DGRAM)
sock.bind(('localhost', 6667))

while True:
    msg, addr = sock.recvfrom(8192) # This is the amount of bytes to read at maximum
    print("Got message from %s: %s" % (addr, msg))
```

Below is an alternative implementation using `socketserver.UDPServer`:

```
from socketserver import BaseRequestHandler, UDPServer

class MyHandler(BaseRequestHandler):
    def handle(self):
        print("Got connection from: %s" % self.client_address)
        msg, sock = self.request
        print("It said: %s" % msg)
        sock.sendto("Got your message!".encode(), self.client_address) # Send reply

serv = UDPServer(('localhost', 6667), MyHandler)
serv.serve_forever()
```

By default, sockets block. This means that execution of the script will wait until the socket receives data.

## Section 121.4: Sending data via TCP

Sending data over the internet is made possible using multiple modules. The sockets module provides low-level access to the underlying Operating System operations responsible for sending or receiving data from other computers or processes.

The following code sends the byte string b'Hello' to a TCP server listening on port 6667 on the host localhost and closes the connection when finished:

```
from socket import socket, AF_INET, SOCK_STREAM
s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 6667)) # The address of the TCP server listening
s.send(b'Hello')
s.close()
```

Socket output is blocking by default, that means that the program will wait in the connect and send calls until the action is 'completed'. For connect that means the server actually accepting the connection. For send it only means that the operating system has enough buffer space to queue the data to be send later.

Sockets should always be closed after use.

## Section 121.5: Multi-threaded TCP Socket Server

When run with no arguments, this program starts a TCP socket server that listens for connections to 127.0.0.1 on

端口5000。服务器在单独的线程中处理每个连接。

当使用-c参数运行时，该程序连接到服务器，读取客户端列表并打印出来。客户端列表以JSON字符串形式传输。客户端名称可以通过传递-n参数来指定。通过传递不同的名称，可以观察对客户端列表的影响。

### client\_list.py

```
import argparse
import json
import socket
import threading

def handle_client(client_list, conn, address):
    name = conn.recv(1024)
    entry = dict(zip(['name', 'address', 'port'], [name, address[0], address[1]]))
    client_list[name] = entry
    conn.sendall(json.dumps(client_list))
    conn.shutdown(socket.SHUT_RDWR)
    conn.close()

def server(client_list):
    print "Starting server..."
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind(('127.0.0.1', 5000))
    s.listen(5)
    while True:
        (conn, address) = s.accept()
        t = threading.Thread(target=handle_client, args=(client_list, conn, address))
        t.daemon = True
        t.start()

    def client(name):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('127.0.0.1', 5000))
        s.send(name)
        data = s.recv(1024)
        result = json.loads(data)
        print json.dumps(result, indent=4)

    def parse_arguments():
        parser = argparse.ArgumentParser()
        parser.add_argument('-c', dest='client', action='store_true')
        parser.add_argument('-n', dest='name', type=str, default='name')
        result = parser.parse_args()
        return result

    def main():
        client_list = dict()
        args = parse_arguments()
        if args.client:
            client(args.name)
        else:
            try:
                server(client_list)
            except KeyboardInterrupt:
                print "Keyboard interrupt"

    if __name__ == '__main__':

```

port 5000. The server handles each connection in a separate thread.

When run with the -c argument, this program connects to the server, reads the client list, and prints it out. The client list is transferred as a JSON string. The client name may be specified by passing the -n argument. By passing different names, the effect on the client list may be observed.

### client\_list.py

```
import argparse
import json
import socket
import threading

def handle_client(client_list, conn, address):
    name = conn.recv(1024)
    entry = dict(zip(['name', 'address', 'port'], [name, address[0], address[1]]))
    client_list[name] = entry
    conn.sendall(json.dumps(client_list))
    conn.shutdown(socket.SHUT_RDWR)
    conn.close()

def server(client_list):
    print "Starting server..."
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind(('127.0.0.1', 5000))
    s.listen(5)
    while True:
        (conn, address) = s.accept()
        t = threading.Thread(target=handle_client, args=(client_list, conn, address))
        t.daemon = True
        t.start()

    def client(name):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect(('127.0.0.1', 5000))
        s.send(name)
        data = s.recv(1024)
        result = json.loads(data)
        print json.dumps(result, indent=4)

    def parse_arguments():
        parser = argparse.ArgumentParser()
        parser.add_argument('-c', dest='client', action='store_true')
        parser.add_argument('-n', dest='name', type=str, default='name')
        result = parser.parse_args()
        return result

    def main():
        client_list = dict()
        args = parse_arguments()
        if args.client:
            client(args.name)
        else:
            try:
                server(client_list)
            except KeyboardInterrupt:
                print "Keyboard interrupt"

    if __name__ == '__main__':

```

```
main()
```

## 服务器输出

```
$ python client_list.py  
启动服务器...
```

## 客户端输出

```
$ python client_list.py -c -n name1  
{  
    "name1": {  
        "address": "127.0.0.1",  
        "port": 62210,  
        "name": "name1"  
    }  
}
```

接收缓冲区限制为1024字节。如果客户端列表的JSON字符串表示超过此大小，将被截断。这将导致抛出以下异常：

```
ValueError: 未终止的字符串, 起始于: 第1行, 第1023列 (字符1022)
```

```
main()
```

## Server Output

```
$ python client_list.py  
Starting server...
```

## Client Output

```
$ python client_list.py -c -n name1  
{  
    "name1": {  
        "address": "127.0.0.1",  
        "port": 62210,  
        "name": "name1"  
    }  
}
```

The receive buffers are limited to 1024 bytes. If the JSON string representation of the client list exceeds this size, it will be truncated. This will cause the following exception to be raised:

```
ValueError: Unterminated string starting at: line 1 column 1023 (char 1022)
```

# 第122章 : Websockets

## 第122.1节 : 使用aiohttp的简单回声

aiohttp 提供异步websockets。

```
Python 3.x 版本 ≥ 3.5
import asyncio
from aiohttp import ClientSession

with ClientSession() as session:
    async def hello_world():

        websocket = await session.ws_connect("wss://echo.websocket.org")

        websocket.send_str("Hello, world!")

        print("Received:", (await websocket.receive()).data)

    await websocket.close()

    loop = asyncio.get_event_loop()
    loop.run_until_complete(hello_world())
```

## 第122.2节 : 使用aiohttp的包装类

aiohttp.ClientSession 可以用作自定义 WebSocket 类的父类。

```
Python 3.x 版本 ≥ 3.5
import asyncio
from aiohttp import ClientSession

class EchoWebSocket(ClientSession):

    URL = "wss://echo.websocket.org"

    def __init__(self):
        super().__init__()
        self.websocket = None

    async def connect(self):
        """连接到 WebSocket."""
        self.websocket = await self.ws_connect(self.URL)

    async def send(self, message):
        """向 WebSocket 发送消息."""
        assert self.websocket is not None, "你必须先连接!"
        self.websocket.send_str(message)
        print("已发送:", message)

    async def receive(self):
        """从 WebSocket 接收一条消息."""
        assert self.websocket is not None, "你必须先连接!"
        return (await self.websocket.receive()).data

    async def read(self):
        """从 WebSocket 读取消息."""
        assert self.websocket is not None, "你必须先连接!"
```

# Chapter 122: Websockets

## Section 122.1: Simple Echo with aiohttp

aiohttp provides asynchronous websockets.

```
Python 3.x Version ≥ 3.5
import asyncio
from aiohttp import ClientSession

with ClientSession() as session:
    async def hello_world():

        websocket = await session.ws_connect("wss://echo.websocket.org")

        websocket.send_str("Hello, world!")

        print("Received:", (await websocket.receive()).data)

    await websocket.close()

    loop = asyncio.get_event_loop()
    loop.run_until_complete(hello_world())
```

## Section 122.2: Wrapper Class with aiohttp

aiohttp.ClientSession may be used as a parent for a custom WebSocket class.

```
Python 3.x Version ≥ 3.5
import asyncio
from aiohttp import ClientSession

class EchoWebSocket(ClientSession):

    URL = "wss://echo.websocket.org"

    def __init__(self):
        super().__init__()
        self.websocket = None

    async def connect(self):
        """Connect to the WebSocket."""
        self.websocket = await self.ws_connect(self.URL)

    async def send(self, message):
        """Send a message to the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
        self.websocket.send_str(message)
        print("Sent:", message)

    async def receive(self):
        """Receive one message from the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
        return (await self.websocket.receive()).data

    async def read(self):
        """Read messages from the WebSocket."""
        assert self.websocket is not None, "You must connect first!"
```

```

while self.websocket.receive():
    message = await self.receive()
        print("收到：" , message)
    if message == "Echo 9!":
        break

async def send(websocket):
    for n in range(10):
        await websocket.send("Echo {}!".format(n))
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()

with EchoWebSocket() as websocket:
    loop.run_until_complete(websocket.connect())

    tasks = (
        send(websocket),
        websocket.read()
    )

loop.run_until_complete(asyncio.wait(tasks))

loop.close()

```

## 第122.3节：使用Autobahn作为WebSocket工厂

Autobahn包可用于Python的WebSocket服务器工厂。

[Python Autobahn包文档](#)

安装时，通常只需使用终端命令

(Linux系统)：

```
sudo pip install autobahn
```

(适用于 Windows)：

```
python -m pip install autobahn
```

然后，可以在 Python 脚本中创建一个简单的回声服务器：

```

from autobahn.asyncio.websocket import WebSocketServerProtocol
class MyServerProtocol(WebSocketServerProtocol):
    """创建服务器协议时，  

    用户定义的类继承自  

    WebSocketServerProtocol，需要重写  

    onMessage、onConnect 等事件以实现  

    用户指定的功能，这些事件  

    定义了服务器的协议，本质上如此"""
    def onMessage(self,payload,isBinary):
        """当服务器接收到消息时，  

        会调用 onMessage 例程。  

    它有必需的参数 payload  

    和布尔值 isBinary。payload 是  

    “消息”的实际内容，isBinary  

    只是一个标志，告诉用户

```

```

while self.websocket.receive():
    message = await self.receive()
    print("Received：" , message)
    if message == "Echo 9!":
        break

async def send(websocket):
    for n in range(10):
        await websocket.send("Echo {}!".format(n))
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()

with EchoWebSocket() as websocket:
    loop.run_until_complete(websocket.connect())

    tasks = (
        send(websocket),
        websocket.read()
    )

loop.run_until_complete(asyncio.wait(tasks))

loop.close()

```

## Section 122.3: Using Autobahn as a Websocket Factory

The Autobahn package can be used for Python web socket server factories.

[Python Autobahn package documentation](#)

To install, typically one would simply use the terminal command

(For Linux):

```
sudo pip install autobahn
```

(For Windows):

```
python -m pip install autobahn
```

Then, a simple echo server can be created in a Python script:

```

from autobahn.asyncio.websocket import WebSocketServerProtocol
class MyServerProtocol(WebSocketServerProtocol):
    """When creating server protocol, the  

    user defined class inheriting the  

    WebSocketServerProtocol needs to override  

    the onMessage, onConnect, et-c events for  

    user specified functionality, these events  

    define your server's protocol, in essence"""
    def onMessage(self,payload,isBinary):
        """The onMessage routine is called  

        when the server receives a message.  

        It has the required arguments payload  

        and the bool isBinary. The payload is the  

        actual contents of the "message" and isBinary  

        is simply a flag to let the user know that

```

负载包含二进制数据。我通常  
否则假设负载是字符串。  
在此示例中，负载将原样返回给发送者。'''  
self.sendMessage(payload,isBinary)

```
if __name__=='__main__':
    尝试:
        import asyncio
    except ImportError:
        "'Trollius = 0.3 被重命名了'"
        import trollius as asyncio
    from autobahn.asyncio.websocketimportWebSocketServerFactory
    factory=WebSocketServerFactory()
    '''初始化 websocket 工厂，并将协议设置为
    上面定义的协议（继承自
    autobahn.asyncio.websocket.WebSocketServerProtocol 的类）'''
    factory.protocol=MyServerProtocol
    '''上面这行可以理解为将 MyServerProtocol 类中描述的 onConnect、onMessage
    等方法“绑定”到服务器上，设置服务器的功能，即协议'''
    loop=asyncio.get_event_loop()
    coro=loop.create_server(factory,'127.0.0.1',9000)
    server=loop.run_until_complete(coro)
    '''在无限循环中运行服务器'''
    尝试:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.close()
        loop.close()
```

在此示例中，服务器在本地主机（127.0.0.1）的9000端口上创建。这是监听的IP和端口。  
这是重要信息，因为通过它，你可以识别计算机的局域网地址，并从你的调制解调器进行端口转发，经过你所有的路由器到达计算机。然后，使用谷歌查询你的广域网IP，你可以设计你的网站向你的广域网IP的9000端口（本例中）发送WebSocket消息。

重要的是你要从调制解调器进行端口转发，也就是说，如果你有多个路由器串联连接到调制解调器，进入调制解调器的配置设置，从调制解调器转发端口到连接的路由器，依此类推，直到最终连接计算机的路由器接收到调制解调器9000端口（本例中）转发的信息。

```
the payload contains binary data. I typically
elsewise assume that the payload is a string.
In this example, the payload is returned to sender verbatim.'''
self.sendMessage(payload,isBinary)
if __name__=='__main__':
    尝试:
        import asyncio
    except ImportError:
        "'Trollius = 0.3 was renamed'"
        import trollius as asyncio
    from autobahn.asyncio.websocketimportWebSocketServerFactory
    factory=WebSocketServerFactory()
    '''Initialize the websocket factory, and set the protocol to the
    above defined protocol(the class that inherits from
    autobahn.asyncio.websocket.WebSocketServerProtocol)'''
    factory.protocol=MyServerProtocol
    '''This above line can be thought of as "binding" the methods
    onConnect, onMessage, et-c that were described in the MyServerProtocol class
    to the server, setting the servers functionality, ie, protocol'''
    loop=asyncio.get_event_loop()
    coro=loop.create_server(factory,'127.0.0.1',9000)
    server=loop.run_until_complete(coro)
    '''Run the server in an infinite loop'''
    尝试:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.close()
        loop.close()
```

In this example, a server is being created on the localhost (127.0.0.1) on port 9000. This is the listening IP and port.  
This is important information, as using this, you could identify your computer's LAN address and port forward from your modem, though whatever routers you have to the computer. Then, using google to investigate your WAN IP, you could design your website to send WebSocket messages to your WAN IP, on port 9000 (in this example).

It is important that you port forward from your modem back, meaning that if you have routers daisy chained to the modem, enter into the modem's configuration settings, port forward from the modem to the connected router, and so forth until the final router your computer is connected to is having the information being received on modem port 9000 (in this example) forwarded to it.

# 第123章：客户端与服务器之间的套接字及消息加密/解密

密码学用于安全目的。使用IDEA加密CTR模式在Python中进行加密/解密的例子不多。本文件的目标：

扩展并实现站对站通信中的RSA数字签名方案。使用哈希保证消息完整性，即SHA-1。生成简单的密钥传输协议。使用IDEA加密密钥。分组密码模式为计数器模式（Counter Mode）。

## 第123.1节：服务器端实现

```
import socket
import hashlib
import os
import time
import itertools
import threading
import sys
import Crypto.Cipher.AES as AES
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#服务器地址和端口号由管理员输入
host= raw_input("服务器地址 -> ")
port = int(input("端口 -> "))
#用于检查服务器和端口的布尔值
check = False
done = False

def animate():
    for c in itertools.cycle(['.', '..', '..', '..', '..']):
        if done:
            break
        sys.stdout.write('\r正在检查IP地址和未使用的端口 '+c)
        sys.stdout.flush()
        time.sleep(0.1)
    sys.stdout.write('\r -----服务器已启动。等待客户端连接-----')

尝试:
    # 设置套接字
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    server.bind((host,port))
server.listen(5)
    check = True
except BaseException:
    print "-----请检查服务器地址或端口-----"
    check = False

if check is True:
    # 服务器退出
shutdown = False
# 打印“服务器启动消息”
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
```

# Chapter 123: Sockets And Message Encryption/Decryption Between Client and Server

Cryptography is used for security purposes. There are not so many examples of Encryption/Decryption in Python using IDEA encryption MODE CTR. **Aim of this documentation :**

Extend and implement of the RSA Digital Signature scheme in station-to-station communication. Using Hashing for integrity of message, that is SHA-1. Produce simple Key Transport protocol. Encrypt Key with IDEA encryption. Mode of Block Cipher is Counter Mode

## Section 123.1: Server side Implementation

```
import socket
import hashlib
import os
import time
import itertools
import threading
import sys
import Crypto.Cipher.AES as AES
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#server address and port number input from admin
host= raw_input("Server Address -> ")
port = int(input("Port -> "))
#boolean for checking server and port
check = False
done = False

def animate():
    for c in itertools.cycle(['.', '..', '..', '..', '..']):
        if done:
            break
        sys.stdout.write('\rCHECKING IP ADDRESS AND NOT USED PORT '+c)
        sys.stdout.flush()
        time.sleep(0.1)
    sys.stdout.write('\r -----SERVER STARTED. WAITING FOR CLIENT----\n')

try:
    #setting up socket
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    server.bind((host,port))
server.listen(5)
    check = True
except BaseException:
    print "-----Check Server Address or Port-----"
    check = False

if check is True:
    # server Quit
shutdown = False
# printing "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
```

```

done = True
#绑定客户端和地址
client,address = server.accept()
print ("客户端已连接。客户端地址 ->",address)
print ("----等待公钥和公钥哈希----")#客户端消息 (公钥)

getpbk = client.recv(2048)

#字符串转换为密钥
server_public_key = RSA.importKey(getpbk)

# 在服务器端对公钥进行哈希，以验证来自客户端的哈希值
hash_object = hashlib.sha1(getpbk)
hex_digest = hash_object.hexdigest()

if getpbk != "":
    print (getpbk)
    client.send("YES")
    gethash = client.recv(1024)
    print ("----公钥的哈希值---- "+gethash)if hex_digest == gethash
    :
        # 创建会话密钥
key_128 = os.urandom(16)
        # 使用CTR模式加密会话密钥
en = AES.new(key_128, AES.MODE_CTR, counter = lambda:key_128)
        encrypto = en.encrypt(key_128)
        #hashing sha1
en_object = hashlib.sha1(encrypto)
        en_digest = en_object.hexdigest()

        print ("----SESSION KEY----"+en_digest)#加密会话密钥和

        公钥
E = server_public_key.encrypt(encrypto,16)
        print ("----ENCRYPTED PUBLIC KEY AND SESSION KEY----"+str(E))print ("----HAND
        SHAKE COMPLETE----")
client.send(str(E))
        while True:
            #来自客户端的消息
newmess = client.recv(1024)
            #将消息从十六进制解码，仅解密消息的加密版本
decoded = newmess.decode("hex")
            #将 en_digest (会话密钥) 作为密钥
key = en_digest[:16]
            print ("来自客户端的加密消息 -> "+newmess)# 解密来自客户端的消
息
ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
        dMsg = ideaDecrypt.decrypt(decoded)
        print ("**新消息** "+time.ctime(time.time()) +" > "+dMsg+"")        mess = raw_inp
ut("发送给客户端的消息 -> ")
        if mess != "":
ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
        eMsg = ideaEncrypt.encrypt(mess)
eMsg = eMsg.encode("hex").upper()
        if eMsg != "":
            print ("发送给客户端的加密消息-> " + eMsg)
            client.send(eMsg)
            client.close()
else:
        print ("----公钥哈希不匹配----")

```

```

done = True
#binding client and address
client,address = server.accept()
print ("CLIENT IS CONNECTED. CLIENT'S ADDRESS ->",address)
print ("\n----WAITING FOR PUBLIC KEY & PUBLIC KEY HASH----\n")

#client's message(Public Key)
getpbk = client.recv(2048)

#conversion of string to KEY
server_public_key = RSA.importKey(getpbk)

#hashing the public key in server side for validating the hash from client
hash_object = hashlib.sha1(getpbk)
hex_digest = hash_object.hexdigest()

if getpbk != "":
    print (getpbk)
    client.send("YES")
    gethash = client.recv(1024)
    print ("\n----HASH OF PUBLIC KEY---- \n"+gethash)
if hex_digest == gethash:
    # creating session key
key_128 = os.urandom(16)
#encrypt CTR MODE session key
en = AES.new(key_128, AES.MODE_CTR, counter = lambda:key_128)
        encrypto = en.encrypt(key_128)
        #hashing sha1
en_object = hashlib.sha1(encrypto)
        en_digest = en_object.hexdigest()

        print ("\n----SESSION KEY----\n"+en_digest)

#encrypting session key and public key
E = server_public_key.encrypt(encrypto,16)
        print ("\n----ENCRYPTED PUBLIC KEY AND SESSION KEY----\n"+str(E))
        print ("\n----HANDSHAKE COMPLETE----")
client.send(str(E))
        while True:
            #message from client
newmess = client.recv(1024)
            #decoding the message from HEXADECIMAL to decrypt the encrypted version of the message only
decoded = newmess.decode("hex")
            #making en_digest(session_key) as the key
key = en_digest[:16]
            print ("\nENCRYPTED MESSAGE FROM CLIENT -> "+newmess)
            #decrypting message from the client
ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
        dMsg = ideaDecrypt.decrypt(decoded)
        print ("\n**New Message** "+time.ctime(time.time()) +" > "+dMsg+"\n")
mess = raw_input("\nMessage To Client -> ")
        if mess != "":
ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
        eMsg = ideaEncrypt.encrypt(mess)
eMsg = eMsg.encode("hex").upper()
        if eMsg != "":
            print ("ENCRYPTED MESSAGE TO CLIENT-> " + eMsg)
            client.send(eMsg)
            client.close()
else:
        print ("\n----PUBLIC KEY HASH DOESNOT MATCH----\n")

```

## 第123.2节：客户端实现

```
import time
import socket
import threading
import hashlib
import itertools
import sys
from Crypto import Random
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#动画加载
done = False
def animate():
    for c in itertools.cycle(['.', '..', '..', '..', '..']):
        if done:
            break
        sys.stdout.write('\r确认连接服务器 '+c)
        sys.stdout.flush()
        time.sleep(0.1)

#公钥和私钥
random_generator = Random.new().read
key = RSA.generate(1024, random_generator)
public = key.publickey().exportKey()
private = key.exportKey()

#对公钥进行哈希处理
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()

# 设置套接字
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

# 主机和端口用户输入
host = raw_input("要连接的服务器地址 -> ")
port = int(input("服务器端口 -> "))
# 绑定地址和端口
server.connect((host, port))
# 打印“服务器启动消息”
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
done = True

def send(t,name,key):
mess = raw_input(name + " : ")
key = key[:16]
# 合并消息和名称
whole = name+ " : "+mess
ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
eMsg = ideaEncrypt.encrypt(whole)
#将加密消息转换为可读的十六进制
eMsg = eMsg.encode("hex").upper()
if eMsg != "":
    print ("发送到服务器的加密消息-> "+eMsg)
server.send(eMsg)
def recv(t,key):
newmess = server.recv(1024)
```

## Section 123.2: Client side Implementation

```
import time
import socket
import threading
import hashlib
import itertools
import sys
from Crypto import Random
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

#animating loading
done = False
def animate():
    for c in itertools.cycle(['.', '..', '..', '..', '..']):
        if done:
            break
        sys.stdout.write('\rCONFIRMING CONNECTION TO SERVER '+c)
        sys.stdout.flush()
        time.sleep(0.1)

#public key and private key
random_generator = Random.new().read
key = RSA.generate(1024, random_generator)
public = key.publickey().exportKey()
private = key.exportKey()

#hashing the public key
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()

#Setting up socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#host and port input user
host = raw_input("Server Address To Be Connected -> ")
port = int(input("Port of The Server -> "))
#binding the address and port
server.connect((host, port))
# printing "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
done = True

def send(t, name, key):
mess = raw_input(name + " : ")
key = key[:16]
#merging the message and the name
whole = name+ " : "+mess
ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
eMsg = ideaEncrypt.encrypt(whole)
#converting the encrypted message to HEXADECIMAL to readable
eMsg = eMsg.encode("hex").upper()
if eMsg != "":
    print ("ENCRYPTED MESSAGE TO SERVER-> "+eMsg)
server.send(eMsg)
def recv(t, key):
newmess = server.recv(1024)
```

```

print ("从服务器接收的加密消息-> " + newmess)    key = key[:16]

decoded = newmess.decode("hex")
ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
dMsg = ideaDecrypt.decrypt(decoded)
print ("**来自服务器的新消息** " + time.ctime(time.time()) + " : " + dMsg + "")

while True:
server.send(public)
confirm = server.recv(1024)
if confirm == "YES":
    server.send(hex_digest)

#connected msg
msg = server.recv(1024)
en = eval(msg)
decrypt = key.decrypt(en)
# hashing sha1
en_object = hashlib.sha1(decrypt)
en_digest = en_object.hexdigest()

print ("----ENCRYPTED PUBLIC KEY AND SESSION KEY FROM SERVER----")print (msg)

print ("----DECRYPTED SESSION KEY----")print (en_di
gest)
print ("----握手完成----")    alais = raw_input("你的
名字 -> ")

while True:
thread_send = threading.Thread(target=send,args=("----发送消息----",
",alais,en_digest))
thread_recv = threading.Thread(target=recv,args=("----接收消息----",
",en_digest))
thread_send.start()
thread_recv.start()

thread_send.join()
thread_recv.join()
time.sleep(0.5)
time.sleep(60)
server.close()

```

```

print ("\nENCRYPTED MESSAGE FROM SERVER-> " + newmess)
key = key[:16]
decoded = newmess.decode("hex")
ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
dMsg = ideaDecrypt.decrypt(decoded)
print ("\n**New Message From Server** " + time.ctime(time.time()) + " : " + dMsg + "\n")

while True:
server.send(public)
confirm = server.recv(1024)
if confirm == "YES":
    server.send(hex_digest)

#connected msg
msg = server.recv(1024)
en = eval(msg)
decrypt = key.decrypt(en)
# hashing sha1
en_object = hashlib.sha1(decrypt)
en_digest = en_object.hexdigest()

print ("\n----ENCRYPTED PUBLIC KEY AND SESSION KEY FROM SERVER----")
print (msg)
print ("\n----DECRYPTED SESSION KEY----")
print (en_digest)
print ("\n----HANDSHAKE COMPLETE----\n")
alais = raw_input("\nYour Name -> ")

while True:
    thread_send = threading.Thread(target=send,args=("----Sending Message----",
",alais,en_digest))
    thread_recv = threading.Thread(target=recv,args=("----Receiving Message----",
",en_digest))
    thread_send.start()
    thread_recv.start()

    thread_send.join()
    thread_recv.join()
    time.sleep(0.5)
    time.sleep(60)
    server.close()

```

# 第124章：Python网络编程

## 第124.1节：创建一个简单的Http服务器

要在本地网络中共享文件或托管简单的网站（http和javascript），你可以使用Python内置的SimpleHTTPServer模块。Python应已添加到你的Path变量中。进入你的文件所在文件夹，输入：

对于python2版本：

```
$ python -m SimpleHTTPServer <端口号>
```

对于python3版本：

```
$ python3 -m http.server <端口号>
```

如果未指定端口号，默认端口为8000。因此输出将是：

```
| 在 0.0.0.0 的 8000 端口上提供 HTTP 服务 ...
```

您可以通过任何连接到本地网络的设备，输入以下地址访问您的文件  
http://hostipaddress:8000/。

hostipaddress 是您的本地 IP 地址，通常以192.168.x.x开头。

要结束该模块，只需按下ctrl+c。

## 第124.2节：创建 TCP 服务器

您可以使用 socketserver 库创建 TCP 服务器。以下是一个简单的回声服务器。

服务器端

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('连接来自:', self.client_address)
        while True:
            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    server = TCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

客户端

```
from socket import socket, AF_INET, SOCK_STREAM
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('localhost', 5000))
sock.send(b'Monty Python')
sock.recv(8192) # 返回 b'Monty Python'
```

# Chapter 124: Python Networking

## Section 124.1: Creating a Simple Http Server

To share files or to host simple websites(http and javascript) in your local network, you can use Python's builtin SimpleHTTPServer module. Python should be in your Path variable. Go to the folder where your files are and type:

For python 2:

```
$ python -m SimpleHTTPServer <portnumber>
```

For python 3:

```
$ python3 -m http.server <portnumber>
```

If port number is not given 8000 is the default port. So the output will be:

```
| Serving HTTP on 0.0.0.0 port 8000 ...
```

You can access to your files through any device connected to the local network by typing  
http://hostipaddress:8000/.

hostipaddress is your local IP address which probably starts with 192.168.x.x.

To finish the module simply press ctrl+c.

## Section 124.2: Creating a TCP server

You can create a TCP server using the socketserver library. Here's a simple echo server.

Server side

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('connection from:', self.client_address)
        while True:
            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    server = TCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Client side

```
from socket import socket, AF_INET, SOCK_STREAM
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('localhost', 5000))
sock.send(b'Monty Python')
sock.recv(8192) # returns b'Monty Python'
```

socketserver 使创建简单的 TCP 服务器相对容易。然而，你应该注意，默认情况下，服务器是单线程的，一次只能服务一个客户端。如果你想处理多个客户端，可以实例化一个 ThreadingTCPServer。

```
from socketserver import ThreadingTCPServer
...
if __name__ == '__main__':
    server = ThreadingTCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

## 第124.3节：创建UDP服务器

使用 socketserver 库可以轻松创建 UDP 服务器。

一个简单的时间服务器：

```
import time
from socketserver import BaseRequestHandler, UDPServer

class CtimeHandler(BaseRequestHandler):
    def handle(self):
        print('连接来自: ', self.client_address)
        # 获取消息和客户端套接字
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    server = UDPServer(('', 5000), CtimeHandler)
    server.serve_forever()
```

测试：

```
>>> 从 socket 导入 socket, AF_INET, SOCK_DGRAM
>>> sock = socket(AF_INET, SOCK_DGRAM)
>>> sock.sendto(b'', ('localhost', 5000))
0
>>> sock.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 5000))
```

## 第124.4节：在线程中启动简单的HttpServer并打开浏览器

如果你的程序在运行过程中输出网页，这非常有用。

```
从 http.server 导入 HTTPServer, CGIHTTPRequestHandler
导入 webbrowser
import threading

定义 start_server(path, port=8000):
    '''启动一个在指定端口提供path服务的简单web服务器'''
    os.chdir(path)
httpd = HTTPServer(('', port), CGIHTTPRequestHandler)
httpd.serve_forever()

# 在新线程中启动服务器
port = 8000
daemon = threading.Thread(name='daemon_server',
```

socketserver makes it relatively easy to create simple TCP servers. However, you should be aware that, by default, the servers are single threaded and can only serve one client at a time. If you want to handle multiple clients, either instantiate a ThreadingTCPServer instead.

```
from socketserver import ThreadingTCPServer
...
if __name__ == '__main__':
    server = ThreadingTCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

## Section 124.3: Creating a UDP Server

A UDP server is easily created using the socketserver library.

a simple time server:

```
import time
from socketserver import BaseRequestHandler, UDPServer

class CtimeHandler(BaseRequestHandler):
    def handle(self):
        print('connection from: ', self.client_address)
        # Get message and client socket
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    server = UDPServer(('', 5000), CtimeHandler)
    server.serve_forever()
```

Testing:

```
>>> 从 socket import socket, AF_INET, SOCK_DGRAM
>>> sock = socket(AF_INET, SOCK_DGRAM)
>>> sick.sendto(b'', ('localhost', 5000))
0
>>> sock.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 5000))
```

## Section 124.4: Start Simple HttpServer in a thread and open the browser

Useful if your program is outputting web pages along the way.

```
从 http.server 导入 HTTPServer, CGIHTTPRequestHandler
import webbrowser
import threading

def start_server(path, port=8000):
    '''Start a simple webserver serving path on port'''
    os.chdir(path)
    httpd = HTTPServer(('', port), CGIHTTPRequestHandler)
    httpd.serve_forever()

    # Start the server in a new thread
    port = 8000
    daemon = threading.Thread(name='daemon_server',
```

```

target=start_server,
        args=('.', port)
daemon.setDaemon(True) # 设置为守护线程, 这样主线程结束时它也会被终止。
daemon.start()

# 打开网页浏览器
webbrowser.open('http://localhost:{}'.format(port))

```

## 第124.5节：最简单的Python套接字客户端-服务器示例

服务器端：

```

import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8089))
serversocket.listen(5) # 变成服务器套接字, 最大连接数为5

while True:
connection, address = serversocket.accept()
buf = connection.recv(64)
if len(buf) > 0:
    print(buf)
break

```

客户端：

```

import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')

```

先运行 SocketServer.py，确保服务器已准备好监听/接收数据，然后客户端向服务器发送信息；服务器收到数据后，程序终止

```

target=start_server,
        args=('.', port)
daemon.setDaemon(True) # Set as a daemon so it will be killed once the main thread is dead.
daemon.start()

# Open the web browser
webbrowser.open('http://localhost:{}' .format(port))

```

## Section 124.5: The simplest Python socket client-server example

Server side:

```

import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8089))
serversocket.listen(5) # become a server socket, maximum 5 connections

while True:
connection, address = serversocket.accept()
buf = connection.recv(64)
if len(buf) > 0:
    print(buf)
break

```

Client Side:

```

import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')

```

First run the SocketServer.py, and make sure the server is ready to listen/receive sth Then the client send info to the server; After the server received sth, it terminates

# 视频：完整的Python训练营：从零开始成为Python 3高手

像专业人士一样学习Python！从基础开始，直到创建你自己的应用程序和游戏！

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ 学习专业使用Python，掌握Python 2和Python 3！
- ✓ 用Python创建游戏，比如井字棋和二十一点！
- ✓ 学习高级Python功能，如collections模块和时间戳的使用！
- ✓ 学习使用面向对象编程和类！
- ✓ 理解复杂主题，如装饰器。
- ✓ 理解如何使用Jupyter Notebook并创建.py文件
- ✓ 了解如何在Jupyter Notebook系统中创建图形用户界面（GUI）！
- ✓ 从零开始全面掌握Python！

今天观看 →

# VIDEO: Complete Python Bootcamp: Go from zero to hero in Python 3

Learn Python like a Professional! Start from the basics and go all the way to creating your own applications and games!

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ Learn to use Python professionally, learning both Python 2 and Python 3!
- ✓ Create games with Python, like Tic Tac Toe and Blackjack!
- ✓ Learn advanced Python features, like the collections module and how to work with timestamps!
- ✓ Learn to use Object Oriented Programming with classes!
- ✓ Understand complex topics, like decorators.
- ✓ Understand how to use both the Jupyter Notebook and create .py files
- ✓ Get an understanding of how to create GUIs in the Jupyter Notebook system!
- ✓ Build a complete understanding of Python from the ground up!

Watch Today →

# 第125章：Python HTTP服务器

## 第125.1节：运行一个简单的HTTP服务器

Python 2.x 版本 ≥ 2.3

```
python -m SimpleHTTPServer 9000
```

Python 3.x 版本 ≥ 3.0

```
python -m http.server 9000
```

运行此命令将在端口9000提供当前目录的文件服务。

如果未提供端口号参数，则服务器将在默认端口8000运行。

-m 标志会在 sys.path 中搜索对应的 .py 文件作为模块运行。

如果你只想在本地主机上提供服务，则需要编写自定义的Python程序，例如：

```
import sys
import BaseHTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

HandlerClass = SimpleHTTPRequestHandler
ServerClass = BaseHTTPServer.HTTPServer
Protocol = "HTTP/1.0"

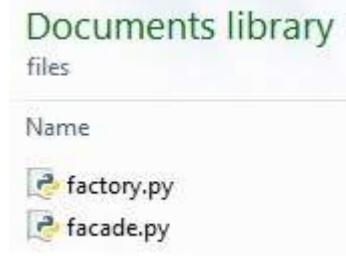
if sys.argv[1:]:
    port = int(sys.argv[1])
else:
    port = 8000
server_address = ('127.0.0.1', port)

HandlerClass.protocol_version = Protocol
httpd = ServerClass(server_address, HandlerClass)

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

## 第125.2节：提供文件服务

假设你有以下文件目录：



你可以按如下方式设置一个网络服务器来提供这些文件：

Python 2.x 版本 ≥ 2.3

```
import SimpleHTTPServer
import SocketServer
```

# Chapter 125: Python HTTP Server

## Section 125.1: Running a simple HTTP server

Python 2.x Version ≥ 2.3

```
python -m SimpleHTTPServer 9000
```

Python 3.x Version ≥ 3.0

```
python -m http.server 9000
```

Running this command serves the files of the current directory at port 9000.

If no argument is provided as port number then server will run on default port 8000.

The -m flag will search sys.path for the corresponding .py file to run as a module.

If you want to only serve on localhost you'll need to write a custom Python program such as:

```
import sys
import BaseHTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

HandlerClass = SimpleHTTPRequestHandler
ServerClass = BaseHTTPServer.HTTPServer
Protocol = "HTTP/1.0"

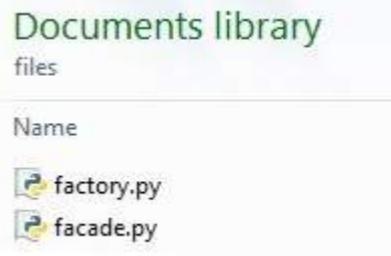
if sys.argv[1:]:
    port = int(sys.argv[1])
else:
    port = 8000
server_address = ('127.0.0.1', port)

HandlerClass.protocol_version = Protocol
httpd = ServerClass(server_address, HandlerClass)

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

## Section 125.2: Serving files

Assuming you have the following directory of files:



You can setup a web server to serve these files as follows:

Python 2.x Version ≥ 2.3

```
import SimpleHTTPServer
import SocketServer
```

PORT = 8000

```
handler = SimpleHTTPServer.SimpleHTTPRequestHandler
httpd = SocketServer.TCPServer(("localhost", PORT), handler)
print "Serving files at port {}".format(PORT)
httpd.serve_forever()
```

Python 3.x 版本 ≥ 3.0

```
import http.server
import socketserver
```

PORT = 8000

```
handler = http.server.SimpleHTTPRequestHandler
httpd = socketserver.TCPServer(("", PORT), handler)
print("serving at port", PORT)
httpd.serve_forever()
```

SocketServer模块提供了设置网络服务器的类和功能。

SocketServer的TCPServer类使用TCP协议设置服务器。构造函数接受一个元组，表示服务器的地址（即IP地址和端口）以及处理服务器请求的类。

SimpleHTTPServer模块的SimpleHTTPRequestHandler类允许提供当前目录下的文件服务。

将脚本保存到相同目录并运行。

运行 HTTP 服务器：

Python 2.x 版本 ≥ 2.3

```
python -m SimpleHTTPServer 8000
```

Python 3.x 版本 ≥ 3.0

```
python -m http.server 8000
```

"-m" 标志会在"sys.path"中搜索对应的".py"文件作为模块运行。

在浏览器中打开 [localhost:8000](http://localhost:8000)，将显示以下内容：

## Directory listing for /

- 
- [facade.py](#)
  - [factory.py](#)
  - [server.py](#)
- 

## 第 125.3 节：使用

BaseHTTPRequestHandler 对 GET、POST、PUT 的基本处理

```
# 来自 BaseHTTPServer 的 BaseHTTPRequestHandler, HTTPServer # python2
```

PORT = 8000

```
handler = SimpleHTTPServer.SimpleHTTPRequestHandler
httpd = SocketServer.TCPServer(("localhost", PORT), handler)
print "Serving files at port {}".format(PORT)
httpd.serve_forever()
```

Python 3.x Version ≥ 3.0

```
import http.server
import socketserver
```

PORT = 8000

```
handler = http.server.SimpleHTTPRequestHandler
httpd = socketserver.TCPServer(("", PORT), handler)
print("serving at port", PORT)
httpd.serve_forever()
```

The `SocketServer` module provides the classes and functionalities to setup a network server.

`SocketServer`'s `TCPServer` class sets up a server using the TCP protocol. The constructor accepts a tuple representing the address of the server (i.e. the IP address and port) and the class that handles the server requests.

The `SimpleHTTPRequestHandler` class of the `SimpleHTTPServer` module allows the files at the current directory to be served.

Save the script at the same directory and run it.

Run the HTTP Server :

Python 2.x Version ≥ 2.3

```
python -m SimpleHTTPServer 8000
```

Python 3.x Version ≥ 3.0

```
python -m http.server 8000
```

The '-m' flag will search 'sys.path' for the corresponding '.py' file to run as a module.

Open [localhost:8000](http://localhost:8000) in the browser, it will give you the following:

## Directory listing for /

- 
- [facade.py](#)
  - [factory.py](#)
  - [server.py](#)
- 

## Section 125.3: Basic handling of GET, POST, PUT using BaseHTTPRequestHandler

```
# from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer # python2
```

```

from http.server import BaseHTTPRequestHandler, HTTPServer # python3
class HandleRequests(BaseHTTPRequestHandler):
    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_GET(self):
        self._set_headers()
        self.wfile.write("received get request")

    def do_POST(self):
        "读取 post 请求体"
        self._set_headers()
        content_len = int(self.headers.getheader('content-length', 0))
        post_body = self.rfile.read(content_len)
        self.wfile.write("received post request:<br>{}".format(post_body))

    def do_PUT(self):
        self.do_POST()

host = ''
port = 80
HTTPServer((host, port), HandleRequests).serve_forever()

```

使用curl的示例输出：

```
$ curl http://localhost/
收到 GET 请求%
```

```
$ curl -X POST http://localhost/
收到 POST 请求 :<br>%
```

```
$ curl -X PUT http://localhost/
收到 POST 请求 :<br>%
```

```
$ echo 'hello world' | curl --data-binary @- http://localhost/
收到 POST 请求 :<br>hello world
```

## 第125.4节：SimpleHTTPServer的程序化API

执行 `python -m SimpleHTTPServer 9000` 会发生什么？

要回答这个问题，我们需要了解 `SimpleHTTPServer` (<https://hg.python.org/cpython/file/2.7/Lib/SimpleHTTPServer.py>) 和 `BaseHTTPServer` (<https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py>) 的结构。

首先，Python 使用参数 `9000` 调用 `SimpleHTTPServer` 模块。现在观察 `SimpleHTTPServer` 代码，

```
def test(HandlerClass = SimpleHTTPRequestHandler,
        ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)
```

```

from http.server import BaseHTTPRequestHandler, HTTPServer # python3
class HandleRequests(BaseHTTPRequestHandler):
    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_GET(self):
        self._set_headers()
        self.wfile.write("received get request")

    def do_POST(self):
        '''Reads post request body'''
        self._set_headers()
        content_len = int(self.headers.getheader('content-length', 0))
        post_body = self.rfile.read(content_len)
        self.wfile.write("received post request:<br>{}".format(post_body))

    def do_PUT(self):
        self.do_POST()

host = ''
port = 80
HTTPServer((host, port), HandleRequests).serve_forever()

```

Example output using curl:

```
$ curl http://localhost/
received get request%
```

```
$ curl -X POST http://localhost/
received post request:<br>%
```

```
$ curl -X PUT http://localhost/
received post request:<br>%
```

```
$ echo 'hello world' | curl --data-binary @- http://localhost/
received post request:<br>hello world
```

## Section 125.4: Programmatic API of SimpleHTTPServer

What happens when we execute `python -m SimpleHTTPServer 9000`?

To answer this question we should understand the construct of `SimpleHTTPServer` (<https://hg.python.org/cpython/file/2.7/Lib/SimpleHTTPServer.py>) and `BaseHTTPServer` (<https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py>).

Firstly, Python invokes the `SimpleHTTPServer` module with `9000` as an argument. Now observing the `SimpleHTTPServer` code,

```
def test(HandlerClass = SimpleHTTPRequestHandler,
        ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)
```

```

if __name__ == '__main__':
    test()

test 函数被调用，传入请求处理器和 ServerClass。现在调用 BaseHTTPServer.test

def test(HandlerClass = BaseHTTPRequestHandler,
        ServerClass = HTTPServer, protocol="HTTP/1.0"):
    测试HTTP请求处理类。

这将在端口8000（或第一个命令行参数）上运行一个HTTP服务器。
"""

if sys.argv[1:]:
    port = int(sys.argv[1])
else:
    port = 8000
server_address = ('', port)

HandlerClass.protocol_version = protocol
httpd = ServerClass(server_address, HandlerClass)

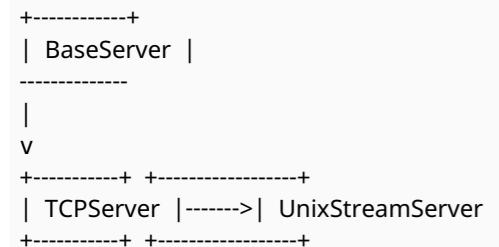
sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()

```

因此，这里解析了用户作为参数传入的端口号，并将其绑定到主机地址。

接着执行使用给定端口和协议的套接字编程的基本步骤。最后，套接字服务器被启动。

这是从SocketServer类继承到其他类的基本概述：



<https://hg.python.org/cpython/file/2.7/Lib/SocketServer.py> 对于查找更多信息非常有用。

```

if __name__ == '__main__':
    test()

```

The test function is invoked following request handlers and ServerClass. Now BaseHTTPServer.test is invoked

```

def test(HandlerClass = BaseHTTPRequestHandler,
        ServerClass = HTTPServer, protocol="HTTP/1.0"):
    """Test the HTTP request handler class.

    This runs an HTTP server on port 8000 (or the first command line
    argument).

"""

if sys.argv[1:]:
    port = int(sys.argv[1])
else:
    port = 8000
server_address = ('', port)

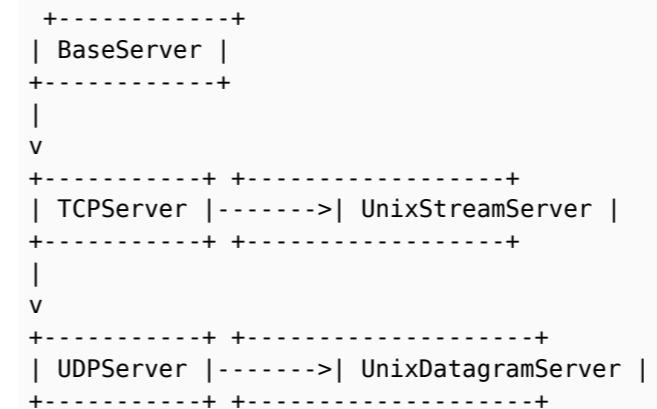
HandlerClass.protocol_version = protocol
httpd = ServerClass(server_address, HandlerClass)

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()

```

Hence here the port number, which the user passed as argument is parsed and is bound to the host address. Further basic steps of socket programming with given port and protocol is carried out. Finally socket server is initiated.

This is a basic overview of inheritance from SocketServer class to other classes:



The links <https://hg.python.org/cpython/file/2.7/Lib/BaseHTTPServer.py> and <https://hg.python.org/cpython/file/2.7/Lib/SocketServer.py> are useful for finding further information.

# 第126章：Flask

Flask 是一个 Python 微型网页框架，用于运行包括 Pinterest、Twilio 和 LinkedIn 在内的主要网站。本主题解释并演示了 Flask 为前端和后端网页开发提供的各种功能。

## 第126.1节：文件和模板

我们可以不用在返回语句中直接输入 HTML 标记，而是使用 `render_template()` 函数：

```
from flask import Flask
from flask import render_template
app = Flask(__name__)

@app.route("/about")
def about():
    return render_template("about-us.html")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

这将使用我们的模板文件 `about-us.html`。为了确保我们的应用程序能够找到该文件，我们必须按以下格式组织目录：

```
- application.py
/templates
  - about-us.html
- login-form.html
/static
  /styles
    - about-style.css
  - login-style.css
/scripts
  - about-script.js
  - login-script.js
```

最重要的是，HTML 中对这些文件的引用必须如下所示：

```
<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-style.css')}}">
```

这将指示应用程序在 `static` 文件夹下的 `styles` 文件夹中查找 `about-style.css`。相同的路径格式适用于所有对图像、样式、脚本或文件的引用。

## 第126.2节：基础知识

下面的示例是一个基本服务器的示例：

```
# 导入 Flask 类
from flask import Flask
# 创建一个应用并检查它是主程序还是被导入的
app = Flask(__name__)

# 指定哪个 URL 触发 hello_world()
@app.route('/')
# 在索引路上运行的函数
def hello_world():
    # 返回要显示的文本
```

# Chapter 126: Flask

Flask is a Python micro web framework used to run major websites including Pinterest, Twilio, and LinkedIn. This topic explains and demonstrates the variety of features Flask offers for both front and back end web development.

## Section 126.1: Files and Templates

Instead of typing our HTML markup into the return statements, we can use the `render_template()` function:

```
from flask import Flask
from flask import render_template
app = Flask(__name__)

@app.route("/about")
def about():
    return render_template("about-us.html")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

This will use our template file `about-us.html`. To ensure our application can find this file we must organize our directory in the following format:

```
- application.py
/templates
  - about-us.html
  - login-form.html
/static
  /styles
    - about-style.css
    - login-style.css
  /scripts
    - about-script.js
    - login-script.js
```

Most importantly, references to these files in the HTML must look like this:

```
<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-style.css')}}">
```

which will direct the application to look for `about-style.css` in the `styles` folder under the `static` folder. The same format of path applies to all references to images, styles, scripts, or files.

## Section 126.2: The basics

The following example is an example of a basic server:

```
# Imports the Flask class
from flask import Flask
# Creates an app and checks if it's the main or imported
app = Flask(__name__)

# Specifies what URL triggers hello_world()
@app.route('/')
# The function run on the index route
def hello_world():
    # Returns the text to be displayed
```

```
return "Hello World!"
```

```
# 如果此脚本不是导入
if __name__ == "__main__":
    # 运行应用直到停止
    app.run()
```

运行此脚本（并安装所有正确的依赖项）应启动本地服务器。主机是127.0.0.1  
通常称为**localhost**。该服务器默认运行在端口**5000**。要访问您的网页服务器，请打开网页  
浏览器并输入URL**localhost:5000**或127.0.0.1:**5000**（无区别）。目前，只有您的计算机可以  
访问该网页服务器。

`app.run()`有三个参数，**host**, **port**和**debug**。主机默认是127.0.0.1，但将其设置为  
0.0.0.0将使您的网页服务器可通过网络中任何设备使用您的私有IP地址在  
URL中访问。端口默认是5000，但如果参数设置为端口80，用户将无需指定端口号  
因为浏览器默认使用端口80。至于调试选项，在开发过程中（绝不在生产环境中）  
将此参数设置为True很有帮助，因为当对Flask项目进行更改时，服务器会自动重启。

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

## 第126.3节：URL路由

在Flask中，URL路由传统上是通过装饰器完成的。这些装饰器可用于静态路由，  
也可用于带参数的URL路由。以下示例中，假设此Flask脚本正在运行该网站  
[www.example.com](http://www.example.com)。

```
@app.route("/")
def index():
    return "你访问了 www.example.com"

@app.route("/about")
def about():
    return "你访问了 www.example.com/about"

@app.route("/users/guido-van-rossum")
def about():
    return "你访问了 www.example.com/guido-van-rossum"
```

通过最后这个路由，你可以看到，给定一个带有 /users/ 和用户名的 URL，我们可以返回一个用户资料。由于为每个用户包含一个 `@app.route()` 会非常低效且混乱，Flask 提供了从 URL 中获取参数的功能：

```
@app.route("/users/<username>")
def profile(username):
    return "欢迎来到" + username + "的个人主页"

城市 = ["奥马哈", "墨尔本", "尼泊尔", "斯图加特", "利马", "开罗", "上海"]

@app.route("/stores/locations/<city>")
def 店面(city):
    if city in 城市:
        return "是的！我们位于 " + city
    else:
        return "不。我们不在 " + city
```

```
return "Hello World!"
```

```
# If this script isn't an import
if __name__ == "__main__":
    # Run the app until stopped
    app.run()
```

Running this script (with all the right dependencies installed) should start up a local server. The host is 127.0.0.1  
commonly known as **localhost**. This server by default runs on port **5000**. To access your webserver, open a web  
browser and enter the URL **localhost:5000** or 127.0.0.1:**5000** (no difference). Currently, only your computer can  
access the webserver.

`app.run()` has three parameters, **host**, **port**, and **debug**. The host is by default 127.0.0.1, but setting this to  
0.0.0.0 will make your web server accessible from any device on your network using your private IP address in the  
URL. the port is by default 5000 but if the parameter is set to port 80, users will not need to specify a port number  
as browsers use port 80 by default. As for the debug option, during the development process (never in production)  
it helps to set this parameter to True, as your server will restart when changes made to your Flask project.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

## Section 126.3: Routing URLs

With Flask, URL routing is traditionally done using decorators. These decorators can be used for static routing, as  
well as routing URLs with parameters. For the following example, imagine this Flask script is running the website  
[www.example.com](http://www.example.com).

```
@app.route("/")
def index():
    return "You went to www.example.com"

@app.route("/about")
def about():
    return "You went to www.example.com/about"

@app.route("/users/guido-van-rossum")
def about():
    return "You went to www.example.com/guido-van-rossum"
```

With that last route, you can see that given a URL with /users/ and the profile name, we could return a profile. Since  
it would be horribly inefficient and messy to include a `@app.route()` for every user, Flask offers to take parameters  
from the URL:

```
@app.route("/users/<username>")
def profile(username):
    return "Welcome to the profile of " + username

cities = ["OMAHA", "MELBOURNE", "NEPAL", "STUTTGART", "LIMA", "CAIRO", "SHANGHAI"]

@app.route("/stores/locations/<city>")
def storefronts(city):
    if city in cities:
        return "Yes! We are located in " + city
    else:
        return "No. We are not located in " + city
```

## 第126.4节：HTTP方法

最常用的两种HTTP方法是GET和POST。Flask可以根据使用的HTTP方法，在相同的URL上运行不同的代码。例如，在有账户的网络服务中，通过相同的URL路由登录页面和登录过程是最方便的。GET请求，也就是你在浏览器中打开URL时发出的请求，应显示登录表单，而POST请求（携带登录数据）应单独处理。还创建了路由来处理DELETE和PUT HTTP方法。

```
@app.route("/login", methods=["GET"])
def login_form():
    return "这是登录表单"
@app.route("/login", methods=["POST"])
def login_auth():
    return "正在处理您的数据"
@app.route("/login", methods=["DELETE", "PUT"])
def deny():
    return "此方法不被允许"
```

为了简化代码，我们可以从 flask 中导入request包。

```
from flask import request

@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "此方法不被允许"
    elif request.method == "GET":
        return "这是登录表单"
    elif request.method == "POST":
        return "正在处理您的数据"
```

要从 POST 请求中获取数据，我们必须使用 request 包：

```
from flask import request

@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "此方法不被允许"
    elif request.method == "GET":
        return "这是登录表单"
    elif request.method == "POST":
        return "用户名是 " + request.form["username"] + "，密码是 " +
request.form["password"]
```

## 第126.5节：Jinja 模板

类似于 Meteor.js，Flask 与前端模板服务集成良好。Flask 默认使用 Jinja 模板。

模板允许在 HTML 文件中使用小段代码，如条件语句或循环。

当我们渲染模板时，除模板文件名外的任何参数都会传递给 HTML 模板服务。以下路由将用户名和加入日期（来自其他地方的函数）传递给 HTML。

```
@app.route("/users/<username>")
def profile(username):
joinedDate = get_joined_date(username) # 此函数代码无关紧要
awards = get_awards(username) # 此函数代码无关紧要
```

## Section 126.4: HTTP Methods

The two most common HTTP methods are **GET** and **POST**. Flask can run different code from the same URL dependent on the HTTP method used. For example, in a web service with accounts, it is most convenient to route the sign in page and the sign in process through the same URL. A GET request, the same that is made when you open a URL in your browser should show the login form, while a POST request (carrying login data) should be processed separately. A route is also created to handle the DELETE and PUT HTTP method.

```
@app.route("/login", methods=["GET"])
def login_form():
    return "This is the login form"
@app.route("/login", methods=["POST"])
def login_auth():
    return "Processing your data"
@app.route("/login", methods=["DELETE", "PUT"])
def deny():
    return "This method is not allowed"
```

To simplify the code a bit, we can import the request package from flask.

```
from flask import request

@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Processing your data"
```

To retrieve data from the POST request, we must use the request package:

```
from flask import request

@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "This method is not allowed"
    elif request.method == "GET":
        return "This is the login forum"
    elif request.method == "POST":
        return "Username was " + request.form["username"] + " and password was " +
request.form["password"]
```

## Section 126.5: Jinja Templating

Similar to Meteor.js, Flask integrates well with front end templating services. Flask uses by default Jinja Templating. Templates allow small snippets of code to be used in the HTML file such as conditionals or loops.

When we render a template, any parameters beyond the template file name are passed into the HTML templating service. The following route will pass the username and joined date (from a function somewhere else) into the HTML.

```
@app.route("/users/<username>")
def profile(username):
joinedDate = get_joined_date(username) # This function's code is irrelevant
awards = get_awards(username) # This function's code is irrelevant
```

```
# joinDate 是字符串, awards 是字符串数组
return render_template("profile.html", username=username, joinDate=joinDate, awards=awards)
```

当此模板被渲染时, 可以使用从 `render_template()` 函数传递给它的变量。这里是 `profile.html` 的内容:

```
<!DOCTYPE html>
<html>
  <head>
    # 如果有 username
    <title>{{ username }} 的个人资料</title>
    # 否则
    <title>未找到用户</title>
  </head>
  <body>
    {% if username %}
      <h1>{{ username }} 于 {{ date }} 加入</h1>
      {% if len(awards) > 0 %}
        <h3>{{ username }} 拥有以下奖项:</h3>
        <ul>
          {% for award in awards %}
            <li>{{ award }}</li>
          {% endfor %}
        </ul>
      {% else %}
        <h3>{{ username }} 没有奖项</h3>
      {% endif %}
    {% else %}
      <h1>未找到该用户名的用户</h1>
    {% endif %}
    {%# 这是一个注释, 不影响输出 %}
  </body>
</html>
```

以下分隔符用于不同的解释:

- `{% ... %}` 表示一个语句
- `{{ ... }}` 表示输出模板的表达式
- `{# ... #}` 表示注释 (不包含在模板输出中)
- `{# ... ##` 表示该行剩余部分应被解释为语句

## 第126.6节：请求对象

`request` 对象提供了对路由所发起请求的信息。要使用该对象, 必须从 `flask` 模块中导入它:

```
from flask import request
```

### URL 参数

在之前的示例中使用了 `request.method` 和 `request.form`, 然而我们也可以使用 `request.args` 属性来获取 URL 参数中键/值的字典。

```
@app.route("/api/users/<username>")
def user_api(username):
  尝试:
    token = request.args.get("key")
    if key == "pA55w0Rd":
```

```
# The joinDate is a string and awards is an array of strings
return render_template("profile.html", username=username, joinDate=joinDate, awards=awards)
```

When this template is rendered, it can use the variables passed to it from the `render_template()` function. Here are the contents of `profile.html`:

```
<!DOCTYPE html>
<html>
  <head>
    # if username
    <title>Profile of {{ username }}</title>
    # else
    <title>No User Found</title>
  </head>
  <body>
    {% if username %}
      <h1>{{ username }} joined on the date {{ date }}</h1>
      {% if len(awards) > 0 %}
        <h3>{{ username }} has the following awards:</h3>
        <ul>
          {% for award in awards %}
            <li>{{ award }}</li>
          {% endfor %}
        </ul>
      {% else %}
        <h3>{{ username }} has no awards</h3>
      {% endif %}
    {% else %}
      <h1>No user was found under that username</h1>
    {% endif %}
    {%# This is a comment and doesn't affect the output %}
  </body>
</html>
```

The following delimiters are used for different interpretations:

- `{% ... %}` denotes a statement
- `{{ ... }}` denotes an expression where a template is outputted
- `{# ... #}` denotes a comment (not included in template output)
- `{# ... ##` implies the rest of the line should be interpreted as a statement

## Section 126.6: The Request Object

The `request` object provides information on the request that was made to the route. To utilize this object, it must be imported from the `flask` module:

```
from flask import request
```

### URL Parameters

In previous examples `request.method` and `request.form` were used, however we can also use the `request.args` property to retrieve a dictionary of the keys/values in the URL parameters.

```
@app.route("/api/users/<username>")
def user_api(username):
  尝试:
    token = request.args.get("key")
    if key == "pA55w0Rd":
```

```

if isUser(username): # 此方法的代码无关紧要
    joined = joinDate(username) # 此方法的代码无关紧要
    return "User " + username + " joined on " + joined
else:
    return "User not found"
else:
    return "密钥错误"
# 如果没有密钥参数
except KeyError:
    return "未提供密钥"

```

要在此环境中正认证，需要以下URL（将用户名替换为任意用户名）：

[www.example.com/api/users/guido-van-rossum?key=pa55w0Rd](http://www.example.com/api/users/guido-van-rossum?key=pa55w0Rd)

## 文件上传

如果文件上传是POST请求中提交表单的一部分，可以使用request对象来处理文件：

```

@app.route("/upload", methods=["POST"])
def upload_file():
    f = request.files["wordlist-upload"]
    f.save("/var/www/uploads/" + f.filename) # 使用原始文件名保存

```

## Cookies

请求还可以包含类似于URL参数的字典形式的Cookies。

```

@app.route("/home")
def home():
    try:
        username = request.cookies.get("username")
        return "您存储的用户名是 " + username
    except KeyError:
        return "未找到用户名Cookie"

```

```

if isUser(username): # The code of this method is irrelevant
    joined = joinDate(username) # The code of this method is irrelevant
    return "User " + username + " joined on " + joined
else:
    return "User not found"
else:
    return "Incorrect key"
# If there is no key parameter
except KeyError:
    return "No key provided"

```

To correctly authenticate in this context, the following URL would be needed (replacing the username with any username):

[www.example.com/api/users/guido-van-rossum?key=pa55w0Rd](http://www.example.com/api/users/guido-van-rossum?key=pa55w0Rd)

## File Uploads

If a file upload was part of the submitted form in a POST request, the files can be handled using the request object:

```

@app.route("/upload", methods=["POST"])
def upload_file():
    f = request.files["wordlist-upload"]
    f.save("/var/www/uploads/" + f.filename) # Store with the original filename

```

## Cookies

The request may also include cookies in a dictionary similar to the URL parameters.

```

@app.route("/home")
def home():
    try:
        username = request.cookies.get("username")
        return "Your stored username is " + username
    except KeyError:
        return "No username cookies was found"

```

# 第127章：RabbitMQ简介 使用AMQPStorm

## 第127.1节：如何从RabbitMQ消费消息

首先导入库。

```
from amqpstorm import Connection
```

在消费消息时，我们首先需要定义一个函数来处理接收到的消息。这个函数可以是任何可调用的函数，并且必须接受一个消息对象，或者一个消息元组（取决于`to_tuple`参数在`start_consuming`中定义）。

除了处理传入消息中的数据外，我们还需要确认（Acknowledge）或拒绝（Reject）该消息。这很重要，因为我们需要让RabbitMQ知道我们已经正确接收并处理了该消息。

```
def on_message(message):
    """该函数在接收到消息时被调用。

:param message: 传递的消息。
:return:
"""
    print("Message:", message.body)

# 确认我们已成功处理该消息，无任何问题。
message.ack()

# 拒绝消息。
# message.reject()

# 拒绝消息，并将其放回队列。
# message.reject(requeue=True)
```

接下来我们需要设置与 RabbitMQ 服务器的连接。

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

之后我们需要设置一个通道。每个连接可以有多个通道，通常在执行多线程任务时，建议（但不是必须）每个线程使用一个通道。

```
channel = connection.channel()
```

通道设置好后，我们需要让 RabbitMQ 知道我们想开始消费消息。在这种情况下，我们将使用之前定义的 `on_message` 函数来处理所有消费的消息。

我们将在 RabbitMQ 服务器上监听的队列是 `simple_queue`，同时我们也告诉 RabbitMQ，处理完所有消息后会进行确认。

```
channel.basic.consume(callback=on_message, queue='simple_queue', no_ack=False)
```

最后，我们需要启动IO循环以开始处理RabbitMQ服务器传递的消息。

```
channel.start_consuming(to_tuple=False)
```

# Chapter 127: Introduction to RabbitMQ using AMQPStorm

## Section 127.1: How to consume messages from RabbitMQ

Start with importing the library.

```
from amqpstorm import Connection
```

When consuming messages, we first need to define a function to handle the incoming messages. This can be any callable function, and has to take a message object, or a message tuple (depending on the `to_tuple` parameter defined in `start_consuming`).

Besides processing the data from the incoming message, we will also have to Acknowledge or Reject the message. This is important, as we need to let RabbitMQ know that we properly received and processed the message.

```
def on_message(message):
    """This function is called on message received.

:param message: Delivered message.
:return:
"""
    print("Message:", message.body)

# Acknowledge that we handled the message without any issues.
message.ack()

# Reject the message.
# message.reject()

# Reject the message, and put it back in the queue.
# message.reject(requeue=True)
```

Next we need to set up the connection to the RabbitMQ server.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

After that we need to set up a channel. Each connection can have multiple channels, and in general when performing multi-threaded tasks, it's recommended (but not required) to have one per thread.

```
channel = connection.channel()
```

Once we have our channel set up, we need to let RabbitMQ know that we want to start consuming messages. In this case we will use our previously defined `on_message` function to handle all our consumed messages.

The queue we will be listening to on the RabbitMQ server is going to be `simple_queue`, and we are also telling RabbitMQ that we will be acknowledging all incoming messages once we are done with them.

```
channel.basic.consume(callback=on_message, queue='simple_queue', no_ack=False)
```

Finally we need to start the IO loop to start processing messages delivered by the RabbitMQ server.

```
channel.start_consuming(to_tuple=False)
```

## 第127.2节：如何向RabbitMQ发布消息

首先导入库。

```
from amqpstorm import Connection
from amqpstorm import Message
```

接下来我们需要打开与RabbitMQ服务器的连接。

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

之后我们需要设置一个通道。每个连接可以有多个通道，通常在执行多线程任务时，建议（但不是必须）每个线程使用一个通道。

```
channel = connection.channel()
```

一旦我们设置好通道，就可以开始准备我们的消息。

```
# 消息属性。
properties = {
    'content_type': 'text/plain',
    'headers': {'key': 'value'}
}

# 创建消息。
message = Message.create(channel=channel, body='Hello World!', properties=properties)
```

现在我们只需调用publish并提供一个 routing\_key即可发布消息。在本例中，我们将消息发送到名为 simple\_queue 的队列。

```
message.publish(routing_key='simple_queue')
```

## 第127.3节：如何在RabbitMQ中创建延迟队列

首先我们需要设置两个基本通道，一个用于主队列，一个用于延迟队列。在我的示例结尾，我包含了一些额外的标志，这些标志不是必需的，但能使代码更可靠；例如

confirm delivery、delivery\_mode 和 durable。你可以在 RabbitMQ 的 manual 中找到更多相关信息。

设置好通道后，我们为主通道添加一个绑定，用于将消息从延迟通道发送到主队列。

```
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')
```

接下来我们需要配置延迟通道，使其在消息过期后将消息转发到主队列。

```
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000,
    'x-dead-letter-exchange': 'amq.direct',
    'x-dead-letter-routing-key': 'hello'
})
```

- x-message-ttl (消息存活时间)

这通常用于在特定时间后自动删除队列中的旧消息，但通过

## Section 127.2: How to publish messages to RabbitMQ

Start with importing the library.

```
from amqpstorm import Connection
from amqpstorm import Message
```

Next we need to open a connection to the RabbitMQ server.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

After that we need to set up a channel. Each connection can have multiple channels, and in general when performing multi-threaded tasks, it's recommended (but not required) to have one per thread.

```
channel = connection.channel()
```

Once we have our channel set up, we can start to prepare our message.

```
# Message Properties.
properties = {
    'content_type': 'text/plain',
    'headers': {'key': 'value'}
}

# Create the message.
message = Message.create(channel=channel, body='Hello World!', properties=properties)
```

Now we can publish the message by simply calling publish and providing a routing\_key. In this case we are going to send the message to a queue called simple\_queue.

```
message.publish(routing_key='simple_queue')
```

## Section 127.3: How to create a delayed queue in RabbitMQ

First we need to set up two basic channels, one for the main queue, and one for the delay queue. In my example at the end, I include a couple of additional flags that are not required, but makes the code more reliable; such as confirm delivery, delivery\_mode and durable. You can find more information on these in the RabbitMQ [manual](#).

After we have set up the channels we add a binding to the main channel that we can use to send messages from the delay channel to our main queue.

```
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')
```

Next we need to configure our delay channel to forward messages to the main queue once they have expired.

```
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000,
    'x-dead-letter-exchange': 'amq.direct',
    'x-dead-letter-routing-key': 'hello'
})
```

- x-message-ttl (Message - Time To Live)

This is normally used to automatically remove old messages in the queue after a specific duration, but by

添加两个可选参数，我们可以改变此行为，而是让该参数以毫秒为单位决定消息在延迟队列中停留的时间。

- [x-dead-letter-routing-key](#)

该变量允许我们在消息过期后将其转移到另一个队列，而不是默认的完全删除行为。

- [x-dead-letter-exchange](#)

该变量决定用于将消息从 `hello_delay` 转移到 `hello` 队列的交换机 (Exchange)。

## 发布到延迟队列

当我们完成所有基本的 Pika 参数设置后，你只需使用 `basic publish` 向延迟队列发送消息。

```
delay_channel.basic.publish(exchange='',
                           routing_key='hello_delay',
                           body='test',
                           properties={'delivery_mod': 2})
```

执行脚本后，你应该会在 RabbitMQ 管理模块中看到以下队列被创建。

Overview				Messages				Message rates			
Name	Exclusive	Parameters	Policy	Status	Ready	Unacked	Total	Incoming	deliver / get	ack	
hello		D		Idle	1	0	1				
hello_delay		TTL DLX DLK D		Idle	0	0	0	0.00/s			

## 示例。

```
from amqpstorm import Connection

connection = Connection('127.0.0.1', 'guest', 'guest')

# 创建普通的"Hello World"类型通道。
channel = connection.channel()
channel.confirm_deliveries()
channel.queue.declare(queue='hello', durable=True)

# 我们需要将此通道绑定到一个交换机，该交换机将用于从我们的延迟队列传输消息。
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')

# 创建我们的延迟通道。
delay_channel = connection.channel()
delay_channel.confirm_deliveries()

# 这里是我们声明延迟和延迟通道的路由的地方。
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000, # 消息在被转移前的延迟时间，单位为毫秒。
    'x-dead-letter-exchange': 'amq.direct', # 用于将消息从A转移到B的交换机。
    'x-dead-letter-routing-key': 'hello' # 我们希望消息被转移到的队列名称。})
```

adding two optional arguments we can change this behaviour, and instead have this parameter determine in milliseconds how long messages will stay in the delay queue.

- [x-dead-letter-routing-key](#)

This variable allows us to transfer the message to a different queue once they have expired, instead of the default behaviour of removing it completely.

- [x-dead-letter-exchange](#)

This variable determines which Exchange used to transfer the message from `hello_delay` to `hello` queue.

## Publishing to the delay queue

When we are done setting up all the basic Pika parameters you simply send a message to the delay queue using `basic publish`.

```
delay_channel.basic.publish(exchange='',
                           routing_key='hello_delay',
                           body='test',
                           properties={'delivery_mod': 2})
```

Once you have executed the script you should see the following queues created in your RabbitMQ management module.

Overview				Messages				Message rates			
Name	Exclusive	Parameters	Policy	Status	Ready	Unacked	Total	Incoming	deliver / get	ack	
hello		D		Idle	1	0	1				
hello_delay		TTL DLX DLK D		Idle	0	0	0	0.00/s			

## Example.

```
from amqpstorm import Connection

connection = Connection('127.0.0.1', 'guest', 'guest')

# Create normal 'Hello World' type channel.
channel = connection.channel()
channel.confirm_deliveries()
channel.queue.declare(queue='hello', durable=True)

# We need to bind this channel to an exchange, that will be used to transfer
# messages from our delay queue.
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')

# Create our delay channel.
delay_channel = connection.channel()
delay_channel.confirm_deliveries()

# This is where we declare the delay, and routing for our delay channel.
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000, # Delay until the message is transferred in milliseconds.
    'x-dead-letter-exchange': 'amq.direct', # Exchange used to transfer the message from A to B.
    'x-dead-letter-routing-key': 'hello' # Name of the queue we want the message transferred to.})
```

```
)
```

```
delay_channel.basic.publish(exchange=' ',  
                           routing_key='hello_delay',  
                           body='test',  
                           properties={'delivery_mode': 2})  
  
print("[x] Sent")
```

```
)
```

```
delay_channel.basic.publish(exchange=' ',  
                           routing_key='hello_delay',  
                           body='test',  
                           properties={'delivery_mode': 2})  
  
print("[x] Sent")
```

# 第128章：描述符

## 第128.1节：简单描述符

有两种不同类型的描述符。数据描述符被定义为同时定义了`__get__()`和`__set__()`方法的对象，而非数据描述符只定义了`__get__()`方法。在考虑重写和实例字典的命名空间时，这一区别非常重要。如果数据描述符和实例字典中的条目同名，数据描述符将优先；但如果是非数据描述符和实例字典中的条目同名，则实例字典中的条目优先。

要创建只读数据描述符，需要定义`get()`和`set()`，其中`set()`在被调用时抛出`AttributeError`异常。定义带有抛出异常占位符的`set()`方法就足以使其成为数据描述符。

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

一个实现示例：

```
class DescPrinter(object):
    """一个记录活动的数据描述符。””
    _val = 7

    def __get__(self, obj, objtype=None):
        print('正在获取...')
        return self._val

    def __set__(self, obj, val):
        print('设置', val)
        self._val = val

    def __delete__(self, obj):
        print('删除中 ...')
        del self._val

class Foo():
    x = DescPrinter()

    i = Foo()
    i.x
    # 获取中 ...
    # 7

    i.x = 100
    # 设置 100
    i.x
    # 获取中 ...
    # 100

    del i.x
    # 删除中 ...
    i.x
    # 获取中 ...
    # 7
```

# Chapter 128: Descriptor

## Section 128.1: Simple descriptor

There are two different types of descriptors. Data descriptors are defined as objects that define both a `__get__()` and a `__set__()` method, whereas non-data descriptors only define a `__get__()` method. This distinction is important when considering overrides and the namespace of an instance's dictionary. If a data descriptor and an entry in an instance's dictionary share the same name, the data descriptor will take precedence. However, if instead a non-data descriptor and an entry in an instance's dictionary share the same name, the instance dictionary's entry will take precedence.

To make a read-only data descriptor, define both `get()` and `set()` with the `set()` raising an `AttributeError` when called. Defining the `set()` method with an exception raising placeholder is enough to make it a data descriptor.

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

An implemented example:

```
class DescPrinter(object):
    """A data descriptor that logs activity.”
    _val = 7

    def __get__(self, obj, objtype=None):
        print('Getting ...')
        return self._val

    def __set__(self, obj, val):
        print('Setting', val)
        self._val = val

    def __delete__(self, obj):
        print('Deleting ...')
        del self._val

class Foo():
    x = DescPrinter()

    i = Foo()
    i.x
    # Getting ...
    # 7

    i.x = 100
    # Setting 100
    i.x
    # Getting ...
    # 100

    del i.x
    # Deleting ...
    i.x
    # Getting ...
    # 7
```

## 第128.2节：双向转换

描述符对象可以使相关对象属性自动响应变化。

假设我们想要模拟一个具有给定频率（赫兹）和周期（秒）的振荡器。当我们更新频率时，希望周期也随之更新；当我们更新周期时，希望频率也随之更新：

```
>>> oscillator = Oscillator(freq=100.0) # 设置频率为100.0赫兹
>>> oscillator.period # 周期是频率的倒数，即0.01秒
0.01
>>> oscillator.period = 0.02 # 设置周期为0.02秒
>>> oscillator.freq # 频率自动调整
50.0
>>> oscillator.freq = 200.0 # 设置频率为200.0赫兹
>>> oscillator.period # 周期自动调整
0.005
```

我们选择其中一个值（频率，单位为赫兹）作为“锚点”，即可以直接设置且无需转换的值，并为其编写一个描述符类：

```
class Hertz(object):
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)
```

“另一个”值（周期，单位为秒）是以锚点为基础定义的。我们编写了一个描述符类来进行转换：

```
class Second(object):
    def __get__(self, instance, owner):
        # 读取周期时，从频率转换
        return 1 / instance.freq

    def __set__(self, instance, value):
        # 设置周期时，更新频率
        instance.freq = 1 / float(value)
```

现在我们可以编写振荡器类：

```
class 振荡器(object):
    period = Second() # 将另一个值设置为类属性

    def __init__(self, freq):
        self.freq = Hertz() # 将锚点值设置为实例属性
        self.freq = freq # 赋值传入的值 - self.period 将被调整
```

## Section 128.2: Two-way conversions

Descriptor objects can allow related object attributes to react to changes automatically.

Suppose we want to model an oscillator with a given frequency (in Hertz) and period (in seconds). When we update the frequency we want the period to update, and when we update the period we want the frequency to update:

```
>>> oscillator = Oscillator(freq=100.0) # Set frequency to 100.0 Hz
>>> oscillator.period # Period is 1 / frequency, i.e. 0.01 seconds
0.01
>>> oscillator.period = 0.02 # Set period to 0.02 seconds
>>> oscillator.freq # The frequency is automatically adjusted
50.0
>>> oscillator.freq = 200.0 # Set the frequency to 200.0 Hz
>>> oscillator.period # The period is automatically adjusted
0.005
```

We pick one of the values (frequency, in Hertz) as the "anchor," i.e. the one that can be set with no conversion, and write a descriptor class for it:

```
class Hertz(object):
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)
```

The "other" value (period, in seconds) is defined in terms of the anchor. We write a descriptor class that does our conversions:

```
class Second(object):
    def __get__(self, instance, owner):
        # When reading period, convert from frequency
        return 1 / instance.freq

    def __set__(self, instance, value):
        # When setting period, update the frequency
        instance.freq = 1 / float(value)
```

Now we can write the Oscillator class:

```
class Oscillator(object):
    period = Second() # Set the other value as a class attribute

    def __init__(self, freq):
        self.freq = Hertz() # Set the anchor value as an instance attribute
        self.freq = freq # Assign the passed value - self.period will be adjusted
```

# 第129章 : tempfile NamedTemporaryFile

参数	描述
模式	打开文件的模式, 默认=w+b
关闭时删除文件, 默认=True	
文件名后缀, 默认=""	
前缀	文件名前缀, 默认='tmp'
目录	放置临时文件的目录名, 默认=None
缓冲区大小, 默认=-1, (使用操作系统默认)	

## 第129.1节 : 创建 (并写入) 已知的持久性临时文件

您可以创建在文件系统上具有可见名称的临时文件, 该文件可以通过name属性访问。在Unix系统上, 该文件可以配置为关闭时删除 (由delete参数设置, 默认值为True), 或者可以稍后重新打开。

下面的代码将创建并打开一个命名的临时文件, 并向该文件写入“Hello World!”。临时文件的文件路径可以通过name访问, 在此示例中, 它被保存到变量path中并打印给用户。

关闭文件后, 文件被重新打开, 临时文件的内容被读取并打印给用户。

```
import tempfile

with tempfile.NamedTemporaryFile(delete=False) as t:
    t.write('Hello World!')
    path = t.name
    print path

with open(path) as t:
    print t.read()
```

输出：

```
/tmp/tmp6pireJ
Hello World!
```

# Chapter 129: tempfile NamedTemporaryFile

param	description
mode	mode to open file, default=w+b
delete	To delete file on closure, default=True
suffix	filename suffix, default=""
prefix	filename prefix, default='tmp'
dir	dirname to place tempfile, default=None
bufsize	default=-1, (operating system default used)

## Section 129.1: Create (and write to a) known, persistent temporary file

You can create temporary files which has a visible name on the file system which can be accessed via the name property. The file can, on unix systems, be configured to delete on closure (set by delete param, default is True) or can be reopened later.

The following will create and open a named temporary file and write 'Hello World!' to that file. The filepath of the temporary file can be accessed via name, in this example it is saved to the variable path and printed for the user. The file is then re-opened after closing the file and the contents of the tempfile are read and printed for the user.

```
import tempfile

with tempfile.NamedTemporaryFile(delete=False) as t:
    t.write('Hello World!')
    path = t.name
    print path

with open(path) as t:
    print t.read()
```

Output:

```
/tmp/tmp6pireJ
Hello World!
```

# 视频：Python数据科学与机器学习训练营

学习如何使用NumPy、Pandas、Seaborn、Matplotlib、Plotly、Scikit-Learn、机器学习、Tensorflow等！



- ✓ 使用Python进行数据科学和机器学习
- ✓ 使用Spark进行大数据分析
- ✓ 实现机器学习算法
- ✓ 学习使用NumPy进行数值数据处理
- ✓ 学习使用Pandas进行数据分析
- ✓ 学习使用Matplotlib进行Python绘图
- ✓ 学习使用Seaborn进行统计图表绘制
- ✓ 使用Plotly进行交互式动态可视化
- ✓ 使用SciKit-Learn完成机器学习任务
- ✓ K均值聚类
- ✓ 逻辑回归
- ✓ 线性回归
- ✓ 随机森林和决策树
- ✓ 神经网络
- ✓ 支持向量机今天观看→

# VIDEO: Python for Data Science and Machine Learning Bootcamp

Learn how to use NumPy, Pandas, Seaborn, Matplotlib , Plotly, Scikit-Learn , Machine Learning, Tensorflow, and more!



- ✓ Use Python for Data Science and Machine Learning
- ✓ Use Spark for Big Data Analysis
- ✓ Implement Machine Learning Algorithms
- ✓ Learn to use NumPy for Numerical Data
- ✓ Learn to use Pandas for Data Analysis
- ✓ Learn to use Matplotlib for Python Plotting
- ✓ Learn to use Seaborn for statistical plots
- ✓ Use Plotly for interactive dynamic visualizations
- ✓ Use SciKit-Learn for Machine Learning Tasks
- ✓ K-Means Clustering
- ✓ Logistic Regression
- ✓ Linear Regression
- ✓ Random Forest and Decision Trees
- ✓ Neural Networks
- ✓ Support Vector Machines

Watch Today →

# 第130章：输入、子集和输出 使用Pandas处理外部数据文件

本节展示了使用pandas读取、子集化和写入外部数据文件的基本代码。

## 第130.1节：使用Pandas导入、子集化和写入外部数据文件的基本代码

```
# 打印当前工作目录
import os
print os.getcwd()
# C:\Python27\Scripts

# 设置工作目录
os.chdir('C:/Users/general1/Documents/simple Python files')
print os.getcwd()
# C:\Users\general1\Documents\simple Python files

# 加载pandas
import pandas as pd

# 读取名为'small_dataset.csv'的csv数据文件，包含4行3个变量
my_data = pd.read_csv("small_dataset.csv")
my_data
#      x   y   z
# 0    1   2   3
# 1    4   5   6
# 2    7   8   9
# 3   10  11  12

my_data.shape      # 数据集中的行数和列数
# (4, 3)

my_data.shape[0]   # 数据集中的行数
# 4

my_data.shape[1]   # 数据集中的列数
# 3

# Python 使用基于0的索引。数据集中的第一行或第一列位于
# 位置0。在 R 中，数据集中的第一行或第一列位于
# 位置1。

# 选择前两行
my_data[0:2]
#      x   y   z
# 0    1   2   3
# 1    4   5   6

# 选择第二行和第三行
my_data[1:3]
#      x   y   z
# 1    4   5   6
# 2    7   8   9

# 选择第三行
my_data[2:3]
#      x   y   z
# 2    7   8   9
```

# Chapter 130: Input, Subset and Output External Data Files using Pandas

This section shows basic code for reading, sub-setting and writing external data files using pandas.

## Section 130.1: Basic Code to Import, Subset and Write External Data Files Using Pandas

```
# Print the working directory
import os
print os.getcwd()
# C:\Python27\Scripts

# Set the working directory
os.chdir('C:/Users/general1/Documents/simple Python files')
print os.getcwd()
# C:\Users\general1\Documents\simple Python files

# load pandas
import pandas as pd

# read a csv data file named 'small_dataset.csv' containing 4 lines and 3 variables
my_data = pd.read_csv("small_dataset.csv")
my_data
#      x   y   z
# 0    1   2   3
# 1    4   5   6
# 2    7   8   9
# 3   10  11  12

my_data.shape      # number of rows and columns in data set
# (4, 3)

my_data.shape[0]   # number of rows in data set
# 4

my_data.shape[1]   # number of columns in data set
# 3

# Python uses 0-based indexing. The first row or column in a data set is located
# at position 0. In R the first row or column in a data set is located
# at position 1.

# Select the first two rows
my_data[0:2]
#      x   y   z
# 0    1   2   3
# 1    4   5   6

# Select the second and third rows
my_data[1:3]
#      x   y   z
# 1    4   5   6
# 2    7   8   9

# Select the third row
my_data[2:3]
#      x   y   z
# 2    7   8   9
```

```
# 选择第一列的前两个元素  
my_data.iloc[0:2, 0:1]  
#      x  
# 0    1  
# 1    4  
  
# 选择变量 y 和 z 的第一个元素  
my_data.loc[0, ['y', 'z']]  
# y    2  
# z    3  
  
# 选择变量 y 和 z 的前三个元素  
my_data.loc[0:2, ['y', 'z']]  
#      y  z  
# 0    2  3  
# 1    5  6  
# 2    8  9  
  
# 将变量 y 和 z 的前三个元素写入外部文件。这里 index = 0 表示不写入行名。  
  
my_data2 = my_data.loc[0:2, ['y', 'z']]  
  
my_data2.to_csv('my.output.csv', index = 0)
```

```
# Select the first two elements of the first column  
my_data.iloc[0:2, 0:1]  
#      x  
# 0    1  
# 1    4  
  
# Select the first element of the variables y and z  
my_data.loc[0, ['y', 'z']]  
# y    2  
# z    3  
  
# Select the first three elements of the variables y and z  
my_data.loc[0:2, ['y', 'z']]  
#      y  z  
# 0    2  3  
# 1    5  6  
# 2    8  9  
  
# Write the first three elements of the variables y and z  
# to an external file. Here index = 0 means do not write row names.  
  
my_data2 = my_data.loc[0:2, ['y', 'z']]  
  
my_data2.to_csv('my.output.csv', index = 0)
```

# 第131章：解压文件

要解压或解压缩 tarball、ZIP 或 gzip 文件，Python 分别提供了 tarfile、zipfile 和 gzip 模块。Python 的 tarfile 模块提供了 TarFile.extractall(path=".\\", members=None) 函数用于从 tarball 文件中提取。Python 的 zipfile 模块提供了 ZipFile.extractall([path[, members[, pwd]]]) 函数用于提取或解压 ZIP 压缩文件。最后，Python 的 gzip 模块提供了 GzipFile 类用于解压缩。

## 第131.1节：使用 Python ZipFile.extractall() 解压 ZIP 文件

```
file_unzip = 'filename.zip'  
unzip = zipfile.ZipFile(file_unzip, 'r')  
unzip.extractall()  
unzip.close()
```

## 第131.2节：使用 Python TarFile.extractall() 解压 tarball

```
file_untar = 'filename.tar.gz'  
untar = tarfile.TarFile(file_untar)  
untar.extractall()  
untar.close()
```

# Chapter 131: Unzipping Files

To extract or uncompress a tarball, ZIP, or gzip file, Python's tarfile, zipfile, and gzip modules are provided respectively. Python's tarfile module provides the TarFile.extractall(path=".\\", members=None) function for extracting from a tarball file. Python's zipfile module provides the ZipFile.extractall([path[, members[, pwd]]]) function for extracting or unzipping ZIP compressed files. Finally, Python's gzip module provides the GzipFile class for decompressing.

## Section 131.1: Using Python ZipFile.extractall() to decompress a ZIP file

```
file_unzip = 'filename.zip'  
unzip = zipfile.ZipFile(file_unzip, 'r')  
unzip.extractall()  
unzip.close()
```

## Section 131.2: Using Python TarFile.extractall() to decompress a tarball

```
file_untar = 'filename.tar.gz'  
untar = tarfile.TarFile(file_untar)  
untar.extractall()  
untar.close()
```

# 第132章：使用ZIP档案

## 第132.1节：检查Zip文件内容

有几种方法可以检查zip文件的内容。你可以使用`printdir`来获取各种信息并发送到`stdout`

```
使用 zipfile.ZipFile(filename) as zip:  
    zip.printdir()
```

```
# 输出:  
# 文件名  
# pyexpat.pyd  
# python.exe  
# python3.dll  
# python35.dll  
# 等等。  
  
修改时间          大小  
2016-06-25 22:13:34 157336  
2016-06-25 22:13:34 39576  
2016-06-25 22:13:34 51864  
2016-06-25 22:13:34 3127960
```

我们也可以使用`namelist`方法获取文件名列表。这里，我们简单地打印该列表：

```
使用 zipfile.ZipFile(filename) as zip:  
    print(zip.namelist())
```

```
# 输出: ['pyexpat.pyd', 'python.exe', 'python3.dll', 'python35.dll', ... 等等 ...]
```

我们可以调用`infolist`方法，而不是`namelist`方法，`infolist`方法返回一个`ZipInfo`对象的列表，这些对象包含每个文件的额外信息，例如时间戳和文件大小：

```
使用 zipfile.ZipFile(filename) as zip:  
info = zip.infolist()  
print(zip[0].filename)  
print(zip[0].date_time)  
print(info[0].file_size)  
  
# 输出: pyexpat.pyd  
# 输出: (2016, 6, 25, 22, 13, 34)  
# 输出: 157336
```

## 第132.2节：打开Zip文件

首先，导入`zipfile`模块，并设置文件名。

```
import zipfile  
filename = 'zipfile.zip'
```

处理zip归档文件与处理普通文件非常相似，你通过打开zip文件创建对象，这样你就可以在关闭文件之前对其进行操作。

```
zip = zipfile.ZipFile(filename)  
print(zip)  
# <zipfile.ZipFile object at 0x000000002E51A90>  
zip.close()
```

在 Python 2.7 以及高于 3.2 的 Python 3 版本中，我们可以使用`with`上下文管理器。我们以“读取”模式打开文件，然后打印文件名列表：

# Chapter 132: Working with ZIP archives

## Section 132.1: Examining Zipfile Contents

There are a few ways to inspect the contents of a zipfile. You can use the `printdir` to just get a variety of information sent to `stdout`

```
with zipfile.ZipFile(filename) as zip:  
    zip.printdir()  
  
# Out:  
# File Name  
# pyexpat.pyd  
# python.exe  
# python3.dll  
# python35.dll  
# etc.  
  
Modified           Size  
2016-06-25 22:13:34 157336  
2016-06-25 22:13:34 39576  
2016-06-25 22:13:34 51864  
2016-06-25 22:13:34 3127960
```

We can also get a list of filenames with the `namelist` method. Here, we simply print the list:

```
with zipfile.ZipFile(filename) as zip:  
    print(zip.namelist())  
  
# Out: ['pyexpat.pyd', 'python.exe', 'python3.dll', 'python35.dll', ... etc. ...]
```

Instead of `namelist`, we can call the `infolist` method, which returns a list of `ZipInfo` objects, which contain additional information about each file, for instance a timestamp and file size:

```
with zipfile.ZipFile(filename) as zip:  
    info = zip.infolist()  
    print(zip[0].filename)  
    print(zip[0].date_time)  
    print(info[0].file_size)  
  
# Out: pyexpat.pyd  
# Out: (2016, 6, 25, 22, 13, 34)  
# Out: 157336
```

## Section 132.2: Opening Zip Files

To start, import the `zipfile` module, and set the filename.

```
import zipfile  
filename = 'zipfile.zip'
```

Working with zip archives is very similar to working with files, you create the object by opening the `zipfile`, which lets you work on it before closing the file up again.

```
zip = zipfile.ZipFile(filename)  
print(zip)  
# <zipfile.ZipFile object at 0x000000002E51A90>  
zip.close()
```

In Python 2.7 and in Python 3 versions higher than 3.2, we can use the `with` context manager. We open the file in “read” mode, and then print a list of filenames:

```
with zipfile.ZipFile(filename, 'r') as z:  
    print(z)  
    # <zipfile.ZipFile object at 0x000000002E51A90>
```

## 第 132.3 节：将 zip 文件内容解压到目录

提取 zip 文件的所有文件内容

```
import zipfile  
with zipfile.ZipFile('zipfile.zip','r') as zfile:  
    zfile.extractall('path')
```

如果您想提取单个文件，请使用提取方法，该方法以名称列表和路径作为输入参数

```
import zipfile  
f=open('zipfile.zip','rb')  
zfile=zipfile.ZipFile(f)  
for cont in zfile.namelist():  
    zfile.extract(cont,path)
```

## 第132.4节：创建新归档文件

要创建新归档文件，请以写入模式打开zipfile。

```
import zipfile  
new_arch=zipfile.ZipFile("filename.zip",mode="w")
```

要向此归档添加文件，请使用write()方法。

```
new_arch.write('filename.txt','filename_in_archive.txt') #第一个参数是文件名，第二个参数是归档中的文件名，默认情况下如果未提供则使用文件名  
new_arch.close()
```

如果想将字节字符串写入归档，可以使用writestr()方法。

```
str_bytes="字符串缓冲区"  
new_arch.writestr('filename_string_in_archive.txt',str_bytes)  
new_arch.close()
```

```
with zipfile.ZipFile(filename, 'r') as z:  
    print(z)  
    # <zipfile.ZipFile object at 0x000000002E51A90>
```

## Section 132.3: Extracting zip file contents to a directory

Extract all file contents of a zip file

```
import zipfile  
with zipfile.ZipFile('zipfile.zip','r') as zfile:  
    zfile.extractall('path')
```

If you want extract single files use extract method, it takes name list and path as input parameter

```
import zipfile  
f=open('zipfile.zip','rb')  
zfile=zipfile.ZipFile(f)  
for cont in zfile.namelist():  
    zfile.extract(cont,path)
```

## Section 132.4: Creating new archives

To create new archive open zipfile with write mode.

```
import zipfile  
new_arch=zipfile.ZipFile("filename.zip",mode="w")
```

To add files to this archive use write() method.

```
new_arch.write('filename.txt','filename_in_archive.txt') #first parameter is filename and second parameter is filename in archive by default filename will be taken if not provided  
new_arch.close()
```

If you want to write string of bytes into the archive you can use writestr() method.

```
str_bytes="string buffer"  
new_arch.writestr('filename_string_in_archive.txt',str_bytes)  
new_arch.close()
```

# 第133章：开始使用GZip

该模块提供了一个简单的接口，用于压缩和解压文件，就像GNU程序gzip和gunzip一样。

数据压缩由zlib模块提供。

gzip模块提供了GzipFile类，该类仿照Python的文件对象设计。GzipFile类读取和写入gzip格式的文件，自动压缩或解压数据，使其看起来像普通的文件对象。

## 第133.1节：读取和写入GNU压缩文件

```
import gzip
import os

outfilename = 'example.txt.gz'
output = gzip.open(outfilename, 'wb')
try:
    output.write('示例文件的内容写在这里。')
finally:
    output.close()

print outfilename, 'contains', os.stat(outfilename).st_size, 'bytes of compressed data'
os.system('file -b --mime %s' % outfilename)
```

将其保存为 1gzip\_write.py1。通过终端运行它。

```
$ python gzip_write.py
application/x-gzip; charset=binary
example.txt.gz 包含 68 字节 的压缩数据
```

# Chapter 133: Getting start with GZip

This module provides a simple interface to compress and decompress files just like the GNU programs gzip and gunzip would.

The data compression is provided by the zlib module.

The gzip module provides the GzipFile class which is modeled after Python's File Object. The GzipFile class reads and writes gzip-format files, automatically compressing or decompressing the data so that it looks like an ordinary file object.

## Section 133.1: Read and write GNU zip files

```
import gzip
import os

outfilename = 'example.txt.gz'
output = gzip.open(outfilename, 'wb')
try:
    output.write('Contents of the example file go here.\n')
finally:
    output.close()

print outfilename, 'contains', os.stat(outfilename).st_size, 'bytes of compressed data'
os.system('file -b --mime %s' % outfilename)
```

Save it as 1gzip\_write.py1. Run it through terminal.

```
$ python gzip_write.py
application/x-gzip; charset=binary
example.txt.gz contains 68 bytes of compressed data
```

# 第134章：栈

栈是一种按照后进先出（LIFO）原则插入和移除对象的容器。在下推栈中只允许两种操作：将元素压入栈中，以及从栈中弹出元素。栈是一种有限访问的数据结构——元素只能在栈顶被添加和移除。以下是栈的结构定义：栈要么为空，要么由栈顶和剩余部分组成，剩余部分也是一个栈。

## 第134.1节：使用列表对象创建栈类

使用list对象，你可以创建一个功能齐全的通用栈，并带有辅助方法，如查看栈顶元素和检查栈是否为空。请查看官方Python文档，了解如何将list用作Stack。

```
#定义一个栈类
class Stack:
    def __init__(self):
        self.items = []

    #检查栈是否为空的方法
    def isEmpty(self):
        return self.items == []

    #入栈方法
    def push(self, item):
        self.items.append(item)

    #出栈方法
    def pop(self):
        return self.items.pop()

    #查看栈顶元素但不移除它
    def peek(self):
        return self.items[-1]

    #获取大小的方法
    def size(self):
        return len(self.items)

    #查看整个栈
    def fullStack(self):
        return self.items
```

一个示例运行：

```
stack = Stack()
print('当前栈:', stack.fullStack())
print('栈为空?:', stack.isEmpty())
print('压入整数 1')
stack.push(1)
print('压入字符串 "我说过了，我是通用栈!"')
stack.push('我说过了，我是通用栈！')
print('压入整数 3')
stack.push(3)
print('当前栈:', stack.fullStack())
print('弹出项:', stack.pop())
print('当前栈:', stack.fullStack())
print('栈为空?:', stack.isEmpty())
```

# Chapter 134: Stack

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push the item into the stack, and pop the item out of the stack**. A stack is a limited access data structure - **elements can be added and removed from the stack only at the top**. Here is a structural definition of a Stack: a stack is either empty or it consists of a top and the rest which is a Stack.

## Section 134.1: Creating a Stack class with a List Object

Using a [list](#) object you can create a fully functional generic Stack with helper methods such as peeking and checking if the stack is Empty. Check out the official python docs for using [list](#) as Stack [here](#).

```
#define a stack class
class Stack:
    def __init__(self):
        self.items = []

    #method to check the stack is empty or not
    def isEmpty(self):
        return self.items == []

    #method for pushing an item
    def push(self, item):
        self.items.append(item)

    #method for popping an item
    def pop(self):
        return self.items.pop()

    #check what item is on top of the stack without removing it
    def peek(self):
        return self.items[-1]

    #method to get the size
    def size(self):
        return len(self.items)

    #to view the entire stack
    def fullStack(self):
        return self.items
```

An example run:

```
stack = Stack()
print('Current stack:', stack.fullStack())
print('Stack empty?:', stack.isEmpty())
print('Pushing integer 1')
stack.push(1)
print('Pushing string "Told you, I am generic stack!"')
stack.push('Told you, I am generic stack!')
print('Pushing integer 3')
stack.push(3)
print('Current stack:', stack.fullStack())
print('Popped item:', stack.pop())
print('Current stack:', stack.fullStack())
print('Stack empty?:', stack.isEmpty())
```

输出：

```
当前栈: []
栈为空?: True
压入整数 1
压入字符串 "我说过了，我是通用栈！"
压入整数 3
当前栈: [1, '我说过了，我是通用栈！', 3]
弹出项: 3
当前栈: [1, '我说过了，我是通用栈！']
栈为空?: False
```

Output:

```
Current stack: []
Stack empty?: True
Pushing integer 1
Pushing string "Told you, I am generic stack!"
Pushing integer 3
Current stack: [1, 'Told you, I am generic stack!', 3]
Popped item: 3
Current stack: [1, 'Told you, I am generic stack!']
Stack empty?: False
```

## 第134.2节：解析括号

栈常用于解析。一个简单的解析任务是检查一串括号是否匹配。

例如，字符串([])是匹配的，因为外层和内层括号成对。()<>不是匹配的，因为最后的)没有对应的配对。([)]也不是匹配的，因为配对必须完全在其他配对的内部或外部。

```
def checkParenth(str):
stack = Stack()
pushChars, popChars = "<({[", "}>)]"
for c in str:
    if c in pushChars:
        stack.push(c)
    elif c in popChars:
        if stack.isEmpty():
            return False
        else:
            stackTop = stack.pop()
            # 检查开括号是否与闭括号匹配
            balancingBracket = pushChars[popChars.index(c)]
            if stackTop != balancingBracket:
                return False
            else:
                return False
return not stack.isEmpty()
```

## Section 134.2: Parsing Parentheses

Stacks are often used for parsing. A simple parsing task is to check whether a string of parentheses are matching.

For example, the string ([]) is matching, because the outer and inner brackets form pairs. ()<> is not matching, because the last ) has no partner. ([)] is also not matching, because pairs must be either entirely inside or outside other pairs.

```
def checkParenth(str):
stack = Stack()
pushChars, popChars = "<({[", "}>)]"
for c in str:
    if c in pushChars:
        stack.push(c)
    elif c in popChars:
        if stack.isEmpty():
            return False
        else:
            stackTop = stack.pop()
            # Checks to see whether the opening bracket matches the closing one
            balancingBracket = pushChars[popChars.index(c)]
            if stackTop != balancingBracket:
                return False
            else:
                return False
return not stack.isEmpty()
```

# 视频：机器学习 A-Z：动手 Python数据科学

学习如何从两位数据科学专家那里用Python创建机器学习算法。包含代码模板。



- ✓ 精通Python上的机器学习
- ✓ 对多种机器学习模型有良好的直觉
- ✓ 做出准确的预测
- ✓ 进行强有力的数据分析
- ✓ 制作稳健的机器学习模型
- ✓ 为您的业务创造强大的附加价值
- ✓ 将机器学习用于个人目的
- ✓ 处理强化学习、自然语言处理和深度学习等特定主题
- ✓ 掌握降维等高级技术
- ✓ 了解针对每种问题应选择哪种机器学习模型
- ✓ 构建一支强大的机器学习模型军团，并知道如何组合它们以解决任何问题

今天观看 →

# VIDEO: Machine Learning A-Z: Hands-On Python In Data Science

Learn to create Machine Learning Algorithms in Python from two Data Science experts. Code templates included.



- ✓ Master Machine Learning on Python
- ✓ Have a great intuition of many Machine Learning models
- ✓ Make accurate predictions
- ✓ Make powerful analysis
- ✓ Make robust Machine Learning models
- ✓ Create strong added value to your business
- ✓ Use Machine Learning for personal purpose
- ✓ Handle specific topics like Reinforcement Learning, NLP and Deep Learning
- ✓ Handle advanced techniques like Dimensionality Reduction
- ✓ Know which Machine Learning model to choose for each type of problem
- ✓ Build an army of powerful Machine Learning models and know how to combine them to solve any problem

Watch Today →

# 第135章：绕过全局解释器锁（GIL）

## 第135.1节：Multiprocessing.Pool

当被问及如何在Python中使用线程时，简单的回答是：“不要。改用进程。”multiprocessing模块让你可以用类似创建线程的语法来创建进程，但我更喜欢使用它们方便的Pool对象。

使用[David Beazley最初用来展示线程对抗GIL危险性的代码](#)，我们将用multiprocessing.Pool重写它：

**David Beazley 展示 GIL 线程问题的代码**

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 00000000

t1 = Thread(target=countdown, args=(COUNT/2,))
t2 = Thread(target=countdown, args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Re-written using multiprocessing.Pool:

```
import multiprocessing
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

start = time.time()
with multiprocessing.Pool as pool:
    pool.map(countdown, [COUNT/2, COUNT/2])

    pool.close()
pool.join()

end = time.time()
print(end-start)
```

这不是创建线程，而是创建新的进程。由于每个进程都有自己的解释器，因此不存在GIL冲突。multiprocessing.Pool会根据机器上的核心数打开相应数量的进程，尽管在上面的示例中，只需要两个进程。在实际场景中，您希望设计的列表长度至少与机器上的处理器数量相同。Pool会运行您指定的函数，并为每个参数调用该函数，最多同时运行它创建的进程数。当函数执行完毕后，列表中剩余的函数将会在该进程上继续运行。

我发现，即使使用with语句，如果不关闭并加入池，进程仍会继续存在。

# Chapter 135: Working around the Global Interpreter Lock (GIL)

## Section 135.1: Multiprocessing.Pool

The simple answer, when asking how to use threads in Python is: "Don't. Use processes, instead." The multiprocessing module lets you create processes with similar syntax to creating threads, but I prefer using their convenient Pool object.

Using [the code that David Beazley first used to show the dangers of threads against the GIL](#), we'll rewrite it using multiprocessing.Pool:

**David Beazley's code that showed GIL threading problems**

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown, args=(COUNT/2,))
t2 = Thread(target=countdown, args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Re-written using multiprocessing.Pool:

```
import multiprocessing
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

start = time.time()
with multiprocessing.Pool as pool:
    pool.map(countdown, [COUNT/2, COUNT/2])

    pool.close()
pool.join()

end = time.time()
print(end-start)
```

Instead of creating threads, this creates new processes. Since each process is its own interpreter, there are no GIL collisions. multiprocessing.Pool will open as many processes as there are cores on the machine, though in the example above, it would only need two. In a real-world scenario, you want to design your list to have at least as much length as there are processors on your machine. The Pool will run the function you tell it to run with each argument, up to the number of processes it creates. When the function finishes, any remaining functions in the list will be run on that process.

I've found that, even using the `with` statement, if you don't close and join the pool, the processes continue to exist.

为了清理资源，我总是关闭并加入我的池。

## 第135.2节：Cython nogil：

Cython是一个替代的Python解释器。它使用GIL，但允许您禁用它。请参阅他们的文档作为示例，使用David Beazley最初用来展示线程对抗GIL危险的代码，我们将使用nogil重写它：

**David Beazley 展示 GIL 线程问题的代码**

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 00000000

t1 = Thread(target=countdown, args=(COUNT/2,))
t2 = Thread(target=countdown, args=(COUNT/2,))
start = time.time()
t1.start(); t2.start()
t1.join(); t2.join()
end = time.time()
print end-start
```

**使用 nogil 重写（仅适用于 CYTHON）：**

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

with nogil:
    t1 = Thread(target=countdown, args=(COUNT/2,))
    t2 = Thread(target=countdown, args=(COUNT/2,))
    start = time.time()
    t1.start(); t2.start()
    t1.join(); t2.join()

end = time.time()
print end-start
```

只要你使用的是 Cython，事情就这么简单。请注意文档中提到你必须确保不更改任何 Python 对象：

语句体中的代码不得以任何方式操作 Python 对象，且不得调用任何操作 Python 对象的函数，除非先重新获取 GIL。Cython 目前不会检查这一点。

To clean up resources, I always close and join my pools.

## Section 135.2: Cython nogil:

Cython is an alternative python interpreter. It uses the GIL, but lets you disable it. See [their documentation](#)

As an example, using [the code that David Beazley first used to show the dangers of threads against the GIL](#), we'll rewrite it using nogil:

**David Beazley's code that showed GIL threading problems**

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown, args=(COUNT/2,))
t2 = Thread(target=countdown, args=(COUNT/2,))
start = time.time()
t1.start(); t2.start()
t1.join(); t2.join()
end = time.time()
print end-start
```

**Re-written using nogil (ONLY WORKS IN CYTHON):**

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

with nogil:
    t1 = Thread(target=countdown, args=(COUNT/2,))
    t2 = Thread(target=countdown, args=(COUNT/2,))
    start = time.time()
    t1.start(); t2.start()
    t1.join(); t2.join()

end = time.time()
print end-start
```

It's that simple, as long as you're using cython. Note that the documentation says you must make sure not to change any python objects:

Code in the body of the statement must not manipulate Python objects in any way, and must not call anything that manipulates Python objects without first re-acquiring the GIL. Cython currently does not check this.

# 第136章：部署

## 第136.1节：上传 Conda 包

开始之前你必须具备：

系统上已安装的 Anaconda 账户 Binstar 账户 如果您未使用Anaconda 1.6+, 请安装[binstar](#)  
命令行客户端：

```
$ conda install binstar  
$ conda update binstar
```

如果您未使用 Anaconda, Binstar 也可通过 pipi 获取：

```
$ pip install binstar
```

现在我们可以登录了：

```
$ binstar login
```

使用 whoami 命令测试您的登录状态：

```
$ binstar whoami
```

我们将上传一个包含简单'hello world'函数的软件包。请先从 Github 获取我的演示包仓库以便跟进：

```
$ git clone https://github.com/<NAME>/<Package>
```

这是一个看起来像这样的简易目录：

```
package/  
setup.py  
test_package/  
    __init__.py  
    hello.py  
bld.bat  
build.sh  
meta.yaml
```

Setup.py 是标准的 Python 构建文件，hello.py 包含我们唯一的 hello\_world() 函数。

bld.bat、build.sh 和 meta.yaml 是用于 Conda 包的脚本和元数据。你可以阅读 [Conda build](#) 页面，了解这三个文件及其用途的更多信息。

现在我们通过运行以下命令来创建包：

```
$ conda build test_package/
```

这就是创建 Conda 包所需的全部步骤。

最后一步是通过复制并粘贴运行 conda build test\_package/ 命令后打印输出的最后一行，将包上传到 binstar。在我的系统中，命令是：

# Chapter 136: Deployment

## Section 136.1: Uploading a Conda Package

Before starting you must have:

Anaconda installed on your system Account on Binstar If you are not using [Anaconda](#) 1.6+ install the [binstar](#) command line client:

```
$ conda install binstar  
$ conda update binstar
```

If you are not using Anaconda the Binstar is also available on pipi:

```
$ pip install binstar
```

Now we can login:

```
$ binstar login
```

Test your login with the whoami command:

```
$ binstar whoami
```

We are going to be uploading a package with a simple 'hello world' function. To follow along start by getting my demonstration package repo from Github:

```
$ git clone https://github.com/<NAME>/<Package>
```

This a small directory that looks like this:

```
package/  
    setup.py  
    test_package/  
        __init__.py  
        hello.py  
    bld.bat  
    build.sh  
    meta.yaml
```

Setup.py is the standard python build file and hello.py has our single hello\_world() function.

The bld.bat, build.sh, and meta.yaml are scripts and metadata for the Conda package. You can read the [Conda build](#) page for more info on those three files and their purpose.

Now we create the package by running:

```
$ conda build test_package/
```

That is all it takes to create a Conda package.

The final step is uploading to binstar by copying and pasting the last line of the print out after running the conda build test\_package/ command. On my system the command is:

```
$ binstar upload /home/xavier/anaconda/conda-bld/linux-64/test_package-0.1.0-py27_0.tar.bz2
```

由于这是您第一次创建软件包和发布，系统将提示您填写一些文本字段，这些字段也可以通过网页应用程序完成。

您将看到完成的提示，以确认您已成功将Conda软件包上传到Binstar。

```
$ binstar upload /home/xavier/anaconda/conda-bld/linux-64/test_package-0.1.0-py27_0.tar.bz2
```

Since it is your first time creating a package and release you will be prompted to fill out some text fields which could alternatively be done through the web app.

You will see a *done* printed out to confirm you have successfully uploaded your Conda package to Binstar.

# 第137章：日志记录

## 第137.1节：Python日志记录简介

该模块定义了实现灵活事件日志系统的函数和类，适用于应用程序和库。

标准库模块提供的日志API的主要好处是所有Python模块都可以参与日志记录，因此您的应用程序日志可以包含您自己的消息，并与第三方模块的消息集成。

那么，让我们开始吧：

### 代码中的示例配置

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
    '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('这是一个 %s 测试', 'debug')
```

输出示例：

```
2016-07-26 18:53:55,332 root DEBUG 这是一个调试测试
```

### 通过INI文件的示例配置

假设文件名为 `logging_config.ini`。有关文件格式的更多详细信息，请参见[logging 教程的logging 配置部分](#)。

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
```

# Chapter 137: Logging

## Section 137.1: Introduction to Python Logging

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules.

So, let's start:

### Example Configuration Directly in Code

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
    '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('this is a %s test', 'debug')
```

Output example:

```
2016-07-26 18:53:55,332 root DEBUG this is a debug test
```

### Example Configuration via an INI File

Assuming the file is named `logging_config.ini`. More details for the file format are in the [logging configuration section of the logging tutorial](#).

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
```

```
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

然后在代码中使用logging.config.fileConfig()

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('经常能很好地处理 %s', '来访游客')
```

#### 通过字典的示例配置

从Python 2.7开始，你可以使用包含配置信息的字典。PEP 391包含了配置字典中必需和可选元素的列表。

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
               '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
    },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
              'formatter': 'f',
              'level': logging.DEBUG}
    },
    root = {
        'handlers': ['h'],
        'level': logging.DEBUG,
    },
)
dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('经常能很好地处理 %s', '来访游客')
```

## 第137.2节：记录异常

如果你想记录异常，可以且应该使用logging.exception(msg)方法：

```
>>> import logging
>>> logging.basicConfig()
>>> try:
...     raise Exception('foo')
... except:
...     logging.exception('bar')
...
错误:root:bar
追踪 (最近一次调用最后) :
文件 "<stdin>", 第 2 行, 在 <module>
异常: foo
```

不要将异常作为参数传递：

```
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

Then use `logging.config.fileConfig()` in the code:

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

#### Example Configuration via a Dictionary

As of Python 2.7, you can use a dictionary with configuration details. [PEP 391](#) contains a list of the mandatory and optional elements in the configuration dictionary.

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
               '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
    },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
              'formatter': 'f',
              'level': logging.DEBUG}
    },
    root = {
        'handlers': ['h'],
        'level': logging.DEBUG,
    },
)
dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

## Section 137.2: Logging exceptions

If you want to log exceptions you can and should make use of the `logging.exception(msg)` method:

```
>>> import logging
>>> logging.basicConfig()
>>> try:
...     raise Exception('foo')
... except:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

**Do not pass the exception as argument:**

由于logging.exception(msg)期望一个msg参数，常见的错误是像下面这样将异常传递给日志调用：

```
>>> try:  
...     raise Exception('foo')  
... except Exception as e:  
...     logging.exception(e)  
  
ERROR:root:foo  
追迹 (最近一次调用最后) :  
文件 "<stdin>", 第 2 行, 在 <module>  
异常: foo
```

虽然乍一看这似乎是正确的做法，但实际上由于异常和各种编码在日志模块中的交互方式，这样做是有问题的：

```
>>> try:  
...     raise Exception(u'föö')  
... except Exception as e:  
...     logging.exception(e)  
  
追迹 (最近一次调用最后) :  
文件 "/.../python2.7/logging/__init__.py", 第 861 行, 在 emit  
    msg = self.format(record)  
文件 "/.../python2.7/logging/__init__.py", 第 734 行, 在 format  
    return fmt.format(record)  
文件 "/.../python2.7/logging/__init__.py", 第 469 行, 在 format  
    s = self._fmt % record._dict_  
UnicodeEncodeError: 'ascii' 编解码器无法编码位置 1-2 的字符：序号不在  
范围(128)内  
日志记录自文件 <stdin>, 第 4 行
```

尝试记录包含 Unicode 字符的异常，这种方式会惨遭失败。它会通过在格式化你的logging.exception(e)调用时引发的新异常覆盖原始异常，从而隐藏原始异常的堆栈跟踪。

显然，在你自己的代码中，你可能知道异常的编码方式。然而，第三方库可能会以不同的方式处理这个问题。

#### 正确用法：

如果你不是传递异常，而是传递一条消息并让 Python 处理，它将正常工作：

```
>>> try:  
...     raise Exception(u'föö')  
... except Exception as e:  
...     logging.exception('bar')  
  
错误: root: bar  
追迹 (最近一次调用最后) :  
文件 "<stdin>", 第 2 行, 在 <module>  
异常: f\xf6\xf6
```

如你所见，在这种情况下我们实际上并没有使用 e，调用logging.exception(...)会神奇地格式化最近的异常。

#### 使用非 ERROR 日志级别记录异常

As `logging.exception(msg)` expects a `msg` arg, it is a common pitfall to pass the exception into the logging call like this:

```
>>> try:  
...     raise Exception('foo')  
... except Exception as e:  
...     logging.exception(e)  
  
ERROR:root:foo  
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
Exception: foo
```

While it might look as if this is the right thing to do at first, it is actually problematic due to the reason how exceptions and various encoding work together in the logging module:

```
>>> try:  
...     raise Exception(u'föö')  
... except Exception as e:  
...     logging.exception(e)  
  
Traceback (most recent call last):  
  File "/.../python2.7/logging/__init__.py", line 861, in emit  
    msg = self.format(record)  
  File "/.../python2.7/logging/__init__.py", line 734, in format  
    return fmt.format(record)  
  File "/.../python2.7/logging/__init__.py", line 469, in format  
    s = self._fmt % record._dict_  
UnicodeEncodeError: 'ascii' codec can't encode characters in position 1-2: ordinal not in  
range(128)  
Logged from file <stdin>, line 4
```

Trying to log an exception that contains unicode chars, this way will [fail miserably](#). It will hide the stacktrace of the original exception by overriding it with a new one that is raised during formatting of your `logging.exception(e)` call.

Obviously, in your own code, you might be aware of the encoding in exceptions. However, 3rd party libs might handle this in a different way.

#### Correct Usage:

If instead of the exception you just pass a message and let python do its magic, it will work:

```
>>> try:  
...     raise Exception(u'föö')  
... except Exception as e:  
...     logging.exception('bar')  
  
ERROR:root:bar  
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
Exception: f\xf6\xf6
```

As you can see we don't actually use e in that case, the call to `logging.exception(...)` magically formats the most recent exception.

#### Logging exceptions with non ERROR log levels

如果你想用除 ERROR 以外的日志级别记录异常，可以使用默认记录器的exc\_info参数：

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

## 访问异常的消息

请注意，现有的库可能会抛出异常，其消息可能是任意的Unicode字符串，或者如果幸运的话，是UTF-8编码的字节字符串。如果你确实需要访问异常的文本，唯一可靠且始终有效的方法是使用`repr(e)` 或者 `%r` 字符串格式化：

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('接收到此异常: %r' % e)
...
错误:root:接收到此异常: Exception(u'f\xf6\xf6',)
追踪 (最近一次调用最后) :
文件 "<stdin>", 第2行, 在<module>
异常: f\xf6\xf6
```

If you want to log an exception with another log level than ERROR, you can use the `exc_info` argument of the default loggers:

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

## Accessing the exception's message

Be aware that libraries out there might throw exceptions with messages as any of unicode or (utf-8 if you're lucky) byte-strings. If you really need to access an exception's text, the only reliable way, that will always work, is to use `repr(e)` or the `%r` string formatting:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('received this exception: %r' % e)
...
ERROR:root:received this exception: Exception(u'f\xf6\xf6',)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xf6\xf6
```

# 第138章：Web服务器网关接口（WSGI）

参数	详细信息
start_response	一个用于处理开始的函数

## 第138.1节：服务器对象（方法）

我们的服务器对象接收一个“application”参数，该参数可以是任何可调用的应用对象（参见其他示例）。它首先写入头部，然后将应用返回的数据主体写入系统标准输出。

```
import os, sys

def run(application):
    environ['wsgi.input'] = sys.stdin
    environ['wsgi.errors'] = sys.stderr

    headers_set = []
    headers_sent = []

    def write(data):
        """
        将来自 'start_response()' 的头部数据以及来自 'response' 的主体数据
        写入系统标准输出。
        """
        if not headers_set:
            raise AssertionError("write() 在 start_response() 之前调用")

        elif not headers_sent:
            status, response_headers = headers_sent[:] = headers_setsys.stdout.write('状态: %s\\r % status')
            for header in response_headers:
                sys.stdout.write('%s: %s\\r % header)sys.stdout.
                write('\\r')

            sys.stdout.write(data)
            sys.stdout.flush()

    def start_response(status, response_headers):
        """
        设置此服务器返回响应的头部信息。
        """
        if headers_set:
            raise AssertionError("头部信息已设置！")

    headers_set[:] = [status, response_headers]
    return write

# 这是“服务器对象”中最重要的部分
# 我们的结果将由作为参数传入此方法的“应用程序”生成
result = application(environ, start_response)
尝试:
    for data in result:
        if data:
            write(data)      # 主体不为空，将其数据发送到 'write()'
            if not headers_sent:
                write("")    # 主体为空，发送空字符串到 'write()'
```

# Chapter 138: Web Server Gateway Interface (WSGI)

Parameter	Details
start_response	A function used to process the start

## Section 138.1: Server Object (Method)

Our server object is given an 'application' parameter which can be any callable application object (see other examples). It writes first the headers, then the body of data returned by our application to the system standard output.

```
import os, sys

def run(application):
    environ['wsgi.input'] = sys.stdin
    environ['wsgi.errors'] = sys.stderr

    headers_set = []
    headers_sent = []

    def write(data):
        """
        Writes header data from 'start_response()' as well as body data from 'response'
        to system standard output.
        """
        if not headers_set:
            raise AssertionError("write() before start_response()")

        elif not headers_sent:
            status, response_headers = headers_sent[:] = headers_setsys.stdout.write('Status: %s\\r\\n' % status)
            for header in response_headers:
                sys.stdout.write('%s: %s\\r\\n' % header)
                sys.stdout.write('\\r\\n')

            sys.stdout.write(data)
            sys.stdout.flush()

    def start_response(status, response_headers):
        """
        Sets headers for the response returned by this server.
        """
        if headers_set:
            raise AssertionError("Headers already set!")

    headers_set[:] = [status, response_headers]
    return write

# This is the most important piece of the 'server object'
# Our result will be generated by the 'application' given to this method as a parameter
result = application(environ, start_response)
try:
    for data in result:
        if data:
            write(data)      # Body isn't empty send its data to 'write()'
            if not headers_sent:
                write("")    # Body is empty, send empty string to 'write()'
```

# 第139章：Python 服务器发送事件

服务器发送事件（SSE）是服务器与客户端（通常是网页浏览器）之间的单向连接，允许服务器向客户端“推送”信息。它类似于WebSocket和长轮询。SSE与WebSocket的主要区别在于SSE是单向的，只有服务器可以向客户端发送信息，而WebSocket则是双方都可以互相发送信息。SSE通常被认为比WebSocket更简单易用/实现。

## 第139.1节：Flask SSE

```
@route("/stream")
def stream():
    def event_stream():
        while True:
            if message_to_send:
                产量数据：
                {} .format(message_to_send)    return Response(event_stream(
), mimetype="text/event-stream")
```

## 第139.2节：Asyncio SSE

此示例使用了asyncio SSE库：<https://github.com/brutasse/asyncio-sse>

```
import asyncio
import sse

class Handler(sse.Handler):
    @asyncio.coroutine
    def handle_request(self):
        yield from asyncio.sleep(2)
        self.send('foo')
        yield from asyncio.sleep(2)
        self.send('bar', event='wakeup')

start_server = sse.serve(Handler, 'localhost', 8888)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

# Chapter 139: Python Server Sent Events

Server Sent Events (SSE) is a unidirectional connection between a server and a client (usually a web browser) that allows the server to "push" information to the client. It is much like websockets and long polling. The main difference between SSE and websockets is that SSE is unidirectional, only the server can send info to the client, whereas with websockets, both can send info to each other. SSE is typically considered to be much simpler to use/implement than websockets.

## Section 139.1: Flask SSE

```
@route("/stream")
def stream():
    def event_stream():
        while True:
            if message_to_send:
                yield "data:
                {} \n\n".format(message_to_send)

    return Response(event_stream(), mimetype="text/event-stream")
```

## Section 139.2: Asyncio SSE

This example uses the asyncio SSE library: <https://github.com/brutasse/asyncio-sse>

```
import asyncio
import sse

class Handler(sse.Handler):
    @asyncio.coroutine
    def handle_request(self):
        yield from asyncio.sleep(2)
        self.send('foo')
        yield from asyncio.sleep(2)
        self.send('bar', event='wakeup')

start_server = sse.serve(Handler, 'localhost', 8888)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

# 视频：机器学习、数据科学和使用 Python 的深度学习

完整的动手机器学习教程，涵盖  
数据科学、Tensorflow、人工智能  
和神经网络



- ✓ 使用Tensorflow和Keras构建人工神经网络
- ✓ 使用深度学习对图像、数据和情感进行分类
- ✓ 使用线性回归、多项式回归和多元回归进行预测
- ✓ 使用Matplotlib和Seaborn进行数据可视化
- ✓ 使用 Apache Spark 的 MLLib 实现大规模机器学习
- ✓ 理解强化学习——以及如何构建一个吃豆人机器人
- ✓ 使用 K-Means 聚类、支持向量机 (SVM) 、KNN、决策树、朴素贝叶斯和主成分分析 (PCA) 对数据进行分类
- ✓ 使用训练/测试和 K 折交叉验证来选择和调整模型
- ✓ 使用基于物品和基于用户的协同过滤构建电影推荐系统

今天观看 →

# VIDEO: Machine Learning, Data Science and Deep Learning with Python

Complete hands-on machine learning tutorial with data science, Tensorflow, artificial intelligence, and neural networks



- ✓ Build artificial neural networks with Tensorflow and Keras
- ✓ Classify images, data, and sentiments using deep learning
- ✓ Make predictions using linear regression, polynomial regression, and multivariate regression
- ✓ Data Visualization with Matplotlib and Seaborn
- ✓ Implement machine learning at massive scale with Apache Spark's MLLib
- ✓ Understand reinforcement learning - and how to build a Pac-Man bot
- ✓ Classify data using K-Means clustering, Support Vector Machines (SVM), KNN, Decision Trees, Naive Bayes, and PCA
- ✓ Use train/test and K-Fold cross validation to choose and tune your models
- ✓ Build a movie recommender system using item-based and user-based collaborative filtering

Watch Today →

# 第140章：其他语言中switch语句的替代方案

## 第140.1节：使用语言提供的内容：if/else结构

好吧，如果你想要一个 switch/ case结构，最直接的方法是使用传统的 if/ else 结构：

```
def switch(value):
    if value == 1:
        return "一"
    if value == 2:
        return "二"
    if value == 42:
        return "关于生命、宇宙及一切问题的答案"
    raise Exception("未找到匹配的情况！")
```

这看起来可能多余，而且不总是美观，但这是迄今为止最高效的方法，而且能完成任务：

```
>>> switch(1)
one
>>> switch(2)
two
>>> switch(3)
...
异常: 未找到匹配项!
>>> switch(42)
关于生命、宇宙和一切问题的答案
```

## 第140.2节：使用函数字典

另一种直接的方法是创建一个函数字典：

```
switch = {
    1: lambda: 'one',
    2: lambda: '二',
    42: lambda: '生命、宇宙及一切的答案',
}
```

然后你添加一个默认函数：

```
def default_case():
    raise Exception('未找到匹配项！')
```

然后你使用字典的 get 方法，根据要检查的值获取对应的函数并执行。如果值不存在于字典中，则运行default\_case。

```
>>> switch.get(1, default_case())
一
>>> switch.get(2, default_case())
二
>>> switch.get(3, default_case())
...
异常: 未找到匹配项!
```

# Chapter 140: Alternatives to switch statement from other languages

## Section 140.1: Use what the language offers: the if/else construct

Well, if you want a switch/case construct, the most straightforward way to go is to use the good old if/else construct:

```
def switch(value):
    if value == 1:
        return "one"
    if value == 2:
        return "two"
    if value == 42:
        return "the answer to the question about life, the universe and everything"
    raise Exception("No case found!")
```

it might look redundant, and not always pretty, but that's by far the most efficient way to go, and it does the job:

```
>>> switch(1)
one
>>> switch(2)
two
>>> switch(3)
...
Exception: No case found!
>>> switch(42)
the answer to the question about life the universe and everything
```

## Section 140.2: Use a dict of functions

Another straightforward way to go is to create a dictionary of functions:

```
switch = {
    1: lambda: 'one',
    2: lambda: 'two',
    42: lambda: 'the answer of life the universe and everything',
}
```

then you add a default function:

```
def default_case():
    raise Exception('No case found!')
```

and you use the dictionary's get method to get the function given the value to check and run it. If value does not exists in dictionary, then default\_case is run.

```
>>> switch.get(1, default_case())
one
>>> switch.get(2, default_case())
two
>>> switch.get(3, default_case())
...
Exception: No case found!
```

```
>>> switch.get(42, default_case())
生命、宇宙和一切的答案
```

你也可以做一些语法糖，使得 switch 看起来更美观：

```
def run_switch(value):
    return switch.get(value, default_case())

>>> run_switch(1)
one
```

### 第140.3节：使用类的自省

你可以使用类来模拟 switch/case 结构。以下示例使用类的自省（使用 getattr() 函数将字符串解析为实例上的绑定方法）来解析“case”部分。

然后将该自省方法别名为\_\_call\_\_方法，以重载()操作符。

```
class SwitchBase:
    def switch(self, case):
        m = getattr(self, 'case_{}'.format(case), None)
        if not m:
            return self.default
        return m

    __call__ = switch
```

然后为了让代码看起来更整洁，我们继承SwitchBase类（但也可以只用一个类实现），并在其中定义所有的case作为方法：

```
class CustomSwitcher:
    def case_1(self):
        return 'one'

    def case_2(self):
        return 'two'

    def case_42(self):
        return '生命、宇宙以及一切的答案！'

    def default(self):
        raise Exception('不是一个案例！')
```

所以我们终于可以使用它了：

```
>>> switch = CustomSwitcher()
>>> print(switch(1))
one
>>> print(switch(2))
two
>>> print(switch(3))
...
Exception: 不是一个案例!
>>> print(switch(42))
生命的答案，宇宙 和 一切！
```

```
>>> switch.get(42, default_case())
the answer of life the universe and everything
```

你也可以做一些语法糖，使得 switch 看起来更美观：

```
def run_switch(value):
    return switch.get(value, default_case())

>>> run_switch(1)
one
```

### Section 140.3: Use class introspection

You can use a class to mimic the switch/case structure. The following is using introspection of a class (using the `getattr()` function that resolves a string into a bound method on an instance) to resolve the "case" part.

Then that introspecting method is aliased to the `__call__` method to overload the () operator.

```
class SwitchBase:
    def switch(self, case):
        m = getattr(self, 'case_{}'.format(case), None)
        if not m:
            return self.default
        return m

    __call__ = switch
```

Then to make it look nicer, we subclass the SwitchBase class (but it could be done in one class), and there we define all the case as methods:

```
class CustomSwitcher:
    def case_1(self):
        return 'one'

    def case_2(self):
        return 'two'

    def case_42(self):
        return 'the answer of life, the universe and everything!'

    def default(self):
        raise Exception('Not a case!')
```

so then we can finally use it:

```
>>> switch = CustomSwitcher()
>>> print(switch(1))
one
>>> print(switch(2))
two
>>> print(switch(3))
...
Exception: Not a case!
>>> print(switch(42))
the answer of life, the universe and everything!
```

## 第140.4节：使用上下文管理器

另一种方法，非常易读且优雅，但效率远低于if/else结构，是构建如下类，该类将读取并存储要比较的值，在上下文中以可调用对象的形式暴露自身，如果匹配存储的值则返回真：

```
类 Switch :
    def __init__(self, value):
        self._val = value
    定义 __enter__(self):
        返回 self
    定义 __exit__(self, type, value, traceback):
        返回 False # 允许追踪发生
    def __call__(self, cond, *mconds):
        return self._val in (cond,) + mconds
```

然后定义这些情况几乎等同于真实的 switch/case结构（下面在一个函数中展示，这样更容易演示）：

```
def run_switch(value):
    使用 Switch(value) 作为 case:
        如果 case(1):
            返回 'one'
        如果 case(2):
            返回 'two'
        如果 case(3):
            返回 '关于生命、宇宙及一切问题的答案'
        # 默认情况
        raise Exception('不是一个案例！')
```

所以执行结果将是：

```
>>> run_switch(1)
one
>>> run_switch(2)
two
>>> run_switch(3)
...
Exception: 不是一个案例!
>>> run_switch(42)
关于生命、宇宙以及一切问题的答案
```

注意事项：

- 此解决方案作为pypi上可用的 `switch` 模块提供。

## Section 140.4: Using a context manager

Another way, which is very readable and elegant, but far less efficient than an if/else structure, is to build a class such as follows, that will read and store the value to compare with, expose itself within the context as a callable that will return true if it matches the stored value:

```
class Switch:
    def __init__(self, value):
        self._val = value
    def __enter__(self):
        return self
    def __exit__(self, type, value, traceback):
        return False # Allows traceback to occur
    def __call__(self, cond, *mconds):
        return self._val in (cond,) + mconds
```

then defining the cases is almost a match to the real switch/case construct (exposed within a function below, to make it easier to show off):

```
def run_switch(value):
    with Switch(value) as case:
        if case(1):
            return 'one'
        if case(2):
            return 'two'
        if case(3):
            return 'the answer to the question about life, the universe and everything'
        # default
        raise Exception('Not a case!')
```

So the execution would be:

```
>>> run_switch(1)
one
>>> run_switch(2)
two
>>> run_switch(3)
...
Exception: Not a case!
>>> run_switch(42)
the answer to the question about life, the universe and everything
```

Nota Bene:

- This solution is being offered as the [switch module available on pypi](#).

# 第141章：列表解构（又称打包和解包）

## 第141.1节：解构赋值

在赋值中，你可以使用“解包”语法将可迭代对象拆分成多个值：

### 作为值的解构

```
a, b = (1, 2)  
print(a)  
# 输出: 1  
print(b)  
# 输出: 2
```

如果你尝试解包的元素数量超过可迭代对象的长度，会报错：

```
a, b, c = [1]  
# 抛出异常: ValueError: 解包的值不够 (预期3个, 实际1个)
```

Python 3.x 版本 > 3.0

### 作为列表的解构

你可以使用以下语法解包未知长度的列表：

```
head, *tail = [1, 2, 3, 4, 5]
```

这里，我们将第一个值提取为标量，其他值提取为列表：

```
print(head)  
# 输出: 1  
print(tail)  
# 输出: [2, 3, 4, 5]
```

这等同于：

```
l = [1, 2, 3, 4, 5]  
head = l[0]  
tail = l[1:]
```

它也适用于多个元素或列表末尾的元素：

```
a, b, *other, z = [1, 2, 3, 4, 5]  
print(a, b, z, other)  
# 输出: 1 2 5 [3, 4]
```

### 在解构赋值中忽略值

如果你只对某个值感兴趣，可以使用`_`来表示你不感兴趣。注意：这仍然会设置`_`，只是大多数人不会将其用作变量。

```
a, _ = [1, 2]  
print(a)  
# 输出: 1  
a, _, c = (1, 2, 3)  
print(a)  
# 打印次数: 1
```

# Chapter 141: List destructuring (aka packing and unpacking)

## Section 141.1: Destructuring assignment

In assignments, you can split an Iterable into values using the "unpacking" syntax:

### Destructuring as values

```
a, b = (1, 2)  
print(a)  
# Prints: 1  
print(b)  
# Prints: 2
```

If you try to unpack more than the length of the iterable, you'll get an error:

```
a, b, c = [1]  
# Raises: ValueError: not enough values to unpack (expected 3, got 1)
```

Python 3.x Version > 3.0

### Destructuring as a list

You can unpack a list of unknown length using the following syntax:

```
head, *tail = [1, 2, 3, 4, 5]
```

Here, we extract the first value as a scalar, and the other values as a list:

```
print(head)  
# Prints: 1  
print(tail)  
# Prints: [2, 3, 4, 5]
```

Which is equivalent to:

```
l = [1, 2, 3, 4, 5]  
head = l[0]  
tail = l[1:]
```

It also works with multiple elements or elements from the end of the list:

```
a, b, *other, z = [1, 2, 3, 4, 5]  
print(a, b, z, other)  
# Prints: 1 2 5 [3, 4]
```

### Ignoring values in destructuring assignments

If you're only interested in a given value, you can use `_` to indicate you aren't interested. Note: this will still set `_`, just most people don't use it as a variable.

```
a, _ = [1, 2]  
print(a)  
# Prints: 1  
a, _, c = (1, 2, 3)  
print(a)  
# Prints: 1
```

```
print(c)
# 输出: 3
```

Python 3.x 版本 > 3.0

## 在解构赋值中忽略列表

最后，你可以使用赋值中的`*_`语法来忽略许多值：

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# 输出: 1
```

这其实没什么特别的，因为你也可以通过索引访问列表。更有趣的是在一次赋值中保留第一个和最后一个值：

```
a, *_ , b = [1, 2, 3, 4, 5]
print(a, b)
# 输出: 1 5
```

或者一次提取多个值：

```
a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
print(a, b, c)
# 输出: 1 3 5
```

## 第141.2节：打包函数参数

在函数中，你可以定义若干必需参数：

```
def fun1(arg1, arg2, arg3):
    return (arg1,arg2,arg3)
```

这将使函数仅在提供三个参数时可调用：

```
fun1(1, 2, 3)
```

你也可以通过使用默认值将参数定义为可选：

```
def fun2(arg1='a', arg2='b', arg3='c'):
    return (arg1,arg2,arg3)
```

因此你可以用多种不同的方式调用该函数，例如：

```
fun2(1)           → (1, b, c)
fun2(1, 2)        → (1, 2, c)
fun2(arg2=2, arg3=3) → (a, 2, 3)
...

```

但你也可以使用解构语法来打包参数，这样你就可以使用列表或字典来赋值变量。

## 打包参数列表

假设你有一个值的列表

```
print(c)
# Prints: 3
```

Python 3.x 版本 > 3.0

## Ignoring lists in destructuring assignments

Finally, you can ignore many values using the `*_` syntax in the assignment:

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# Prints: 1
```

which is not really interesting, as you could use indexing on the list instead. Where it gets nice is to keep first and last values in one assignment:

```
a, *_ , b = [1, 2, 3, 4, 5]
print(a, b)
# Prints: 1 5
```

or extract several values at once:

```
a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
print(a, b, c)
# Prints: 1 3 5
```

## Section 141.2: Packing function arguments

In functions, you can define a number of mandatory arguments:

```
def fun1(arg1, arg2, arg3):
    return (arg1,arg2,arg3)
```

which will make the function callable only when the three arguments are given:

```
fun1(1, 2, 3)
```

and you can define the arguments as optional, by using default values:

```
def fun2(arg1='a', arg2='b', arg3='c'):
    return (arg1,arg2,arg3)
```

so you can call the function in many different ways, like:

```
fun2(1)           → (1, b, c)
fun2(1, 2)        → (1, 2, c)
fun2(arg2=2, arg3=3) → (a, 2, 3)
...

```

But you can also use the destructuring syntax to *pack* arguments up, so you can assign variables using a `list` or a `dict`.

## Packing a list of arguments

Consider you have a list of values

```
I = [1,2,3]
```

你可以使用`*`语法将值列表作为参数调用函数：

```
fun1(*I)
# 返回: (1,2,3)
fun1(*['w', 't', 'f'])
# 返回: ('w','t','f')
```

但是如果你没有提供一个长度与参数数量匹配的列表：

```
fun1(*['oops'])
# 抛出异常: TypeError: fun1() 缺少 2 个必需的位置参数: 'arg2' 和 'arg3'
```

## 打包关键字参数

现在，你也可以使用字典来打包参数。你可以使用`**`操作符告诉Python将`dict`解包为参数值：

```
d = {
    'arg1': 1,
    'arg2': 2,
    'arg3': 3
}
fun1(**d)
# 返回: (1, 2, 3)
```

当函数只有位置参数（没有默认值的参数）时，字典需要包含所有预期的参数，且不能有多余的参数，否则会报错：

```
fun1(**{'arg1':1, 'arg2':2})
# 抛出异常: TypeError: fun1() 缺少 1 个必需的位置参数: 'arg3'
fun1(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# 抛出异常: TypeError: fun1() 收到一个意外的关键字参数 'arg4'
```

对于具有可选参数的函数，你可以用同样的方式将参数打包成字典：

```
fun2(**d)
# 返回: (1, 2, 3)
```

但这里你可以省略某些值，因为它们会被默认值替代：

```
fun2(**{'arg2': 2})
# 返回: ('a', 2, 'c')
```

和之前一样，你不能传入不存在的额外参数：

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# 抛出异常: TypeError: fun2() 收到一个意外的关键字参数 'arg4'
```

在实际使用中，函数可以同时拥有位置参数和可选参数，且行为相同：

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

```
I = [1,2,3]
```

You can call the function with the list of values as an argument using the`*` syntax:

```
fun1(*I)
# Returns: (1,2,3)
fun1(*['w', 't', 'f'])
# Returns: ('w','t','f')
```

But if you do not provide a list which length matches the number of arguments:

```
fun1(*['oops'])
# Raises: TypeError: fun1() missing 2 required positional arguments: 'arg2' and 'arg3'
```

## Packing keyword arguments

Now, you can also pack arguments using a dictionary. You can use the`**` operator to tell Python to unpack the`dict` as parameter values:

```
d = {
    'arg1': 1,
    'arg2': 2,
    'arg3': 3
}
fun1(**d)
# Returns: (1, 2, 3)
```

when the function only has positional arguments (the ones without default values) you need the dictionary to be contain of all the expected parameters, and have no extra parameter, or you'll get an error:

```
fun1(**{'arg1':1, 'arg2':2})
# Raises: TypeError: fun1() missing 1 required positional argument: 'arg3'
fun1(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun1() got an unexpected keyword argument 'arg4'
```

For functions that have optional arguments, you can pack the arguments as a dictionary the same way:

```
fun2(**d)
# Returns: (1, 2, 3)
```

But there you can omit values, as they will be replaced with the defaults:

```
fun2(**{'arg2': 2})
# Returns: ('a', 2, 'c')
```

And the same as before, you cannot give extra values that are not existing parameters:

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Raises: TypeError: fun2() got an unexpected keyword argument 'arg4'
```

In real world usage, functions can have both positional and optional arguments, and it works the same:

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

你可以只用一个可迭代对象来调用函数：

```
fun3(*[1])
# 返回: (1, 'b', 'c')
fun3(*[1,2,3])
# 返回: (1, 2, 3)
```

或者只用一个字典：

```
fun3(**{'arg1':1})
# 返回: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# 返回: (1, 2, 3)
```

或者你也可以在同一次调用中同时使用两者：

```
fun3(*[1,2], **{'arg3':3})
# 返回: (1,2,3)
```

但请注意，你不能为同一个参数提供多个值：

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
# 抛出异常: TypeError: fun3() 收到参数 'arg2' 的多个值
```

## 第141.3节：函数参数的解包

当你想创建一个函数，可以接受任意数量的参数，并且不在“编译”时强制参数的位置或名称时，这是可行的，方法如下：

```
def fun1(*args, **kwargs):
    print(args, kwargs)
```

\*args 和 \*\*kwargs 参数是特殊参数，分别被设置为元组 (tuple) 和字典 (dict)：

```
fun1(1, 2, 3)
# 输出: (1, 2, 3) {}
fun1(a=1, b=2, c=3)
# 输出: () {'a': 1, 'b': 2, 'c': 3}
fun1('x', 'y', 'z', a=1, b=2, c=3)
# 输出: ('x', 'y', 'z') {'a': 1, 'b': 2, 'c': 3}
```

如果你看足够多的 Python 代码，你会很快发现它在传递参数给另一个函数时被广泛使用。例如，如果你想扩展字符串类：

```
class MyString(str):
    def __init__(self, *args, **kwargs):
        print('构造 MyString')
        super(MyString, self).__init__(*args, **kwargs)
```

you can call the function with just an iterable:

```
fun3(*[1])
# Returns: (1, 'b', 'c')
fun3(*[1,2,3])
# Returns: (1, 2, 3)
```

or with just a dictionary:

```
fun3(**{'arg1':1})
# Returns: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# Returns: (1, 2, 3)
```

or you can use both in the same call:

```
fun3(*[1,2], **{'arg3':3})
# Returns: (1,2,3)
```

Beware though that you cannot provide multiple values for the same argument:

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
# Raises: TypeError: fun3() got multiple values for argument 'arg2'
```

## Section 141.3: Unpacking function arguments

When you want to create a function that can accept any number of arguments, and not enforce the position or the name of the argument at "compile" time, it's possible and here's how:

```
def fun1(*args, **kwargs):
    print(args, kwargs)
```

The \*args and \*\*kwargs parameters are special parameters that are set to a tuple and a dict, respectively:

```
fun1(1,2,3)
# Prints: (1, 2, 3) {}
fun1(a=1, b=2, c=3)
# Prints: () {'a': 1, 'b': 2, 'c': 3}
fun1('x', 'y', 'z', a=1, b=2, c=3)
# Prints: ('x', 'y', 'z') {'a': 1, 'b': 2, 'c': 3}
```

If you look at enough Python code, you'll quickly discover that it is widely being used when passing arguments over to another function. For example if you want to extend the string class:

```
class MyString(str):
    def __init__(self, *args, **kwargs):
        print('Constructing MyString')
        super(MyString, self).__init__(*args, **kwargs)
```

# 第142章：访问Python源代码和字节码

## 第142.1节：显示函数的字节码

Python解释器在将代码执行到Python虚拟机之前会先将其编译成字节码（另见“什么是Python字节码？”）。

以下是查看Python函数字节码的方法

```
import dis

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)

# 显示该函数的反汇编字节码。
dis.dis(fib)
```

dis模块中的函数`dis.dis`将返回传入函数的反编译字节码。

## 第142.2节：显示对象的源代码

### 非内置对象

要打印Python对象的源代码，请使用`inspect`。请注意，这对内置对象或交互式定义的对象不起作用。对于这些对象，您需要稍后介绍的其他方法。

以下是如何打印random模块中`randint`方法的源代码：

```
import random
import inspect

print(inspect.getsource(random.randint))
# 输出：
#     def randint(self, a, b):
#         """返回范围[a, b]内的随机整数，包括两个端点。
#         """
#         return self.randrange(a, b+1)
```

仅打印文档字符串

```
print(inspect.getdoc(random.randint))
# 输出：
# 返回范围 [a, b] 内的随机整数，包括两个端点。
```

打印定义方法`random.randint`的文件完整路径：

```
print(inspect.getfile(random.randint))
# c:\Python35\lib\random.py
print(random.randint.__code__.co_filename) # 等同于上述
# c:\Python35\lib\random.py
```

交互式定义的对象

# Chapter 142: Accessing Python source code and bytecode

## Section 142.1: Display the bytecode of a function

The Python interpreter compiles code to bytecode before executing it on the Python's virtual machine (see also “What is python bytecode?”).

Here's how to view the bytecode of a Python function

```
import dis

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)

# Display the disassembled bytecode of the function.
dis.dis(fib)
```

The function `dis.dis` in the `dis` module will return a decompiled bytecode of the function passed to it.

## Section 142.2: Display the source code of an object

### Objects that are not built-in

To print the source code of a Python object use `inspect`. Note that this won't work for built-in objects nor for objects defined interactively. For these you will need other methods explained later.

Here's how to print the source code of the method `randint` from the `random` module:

```
import random
import inspect

print(inspect.getsource(random.randint))
# Output:
#     def randint(self, a, b):
#         """Return random integer in range [a, b], including both end points.
#         """
#         return self.randrange(a, b+1)
```

仅打印文档字符串

```
print(inspect.getdoc(random.randint))
# Output:
# Return random integer in range [a, b], including both end points.
```

Print full path of the file where the method `random.randint` is defined:

```
print(inspect.getfile(random.randint))
# c:\Python35\lib\random.py
print(random.randint.__code__.co_filename) # equivalent to the above
# c:\Python35\lib\random.py
```

Objects defined interactively

如果对象是交互式定义的，`inspect` 无法提供源代码，但你可以使用 `dill.source.getsource` 代替

```
# 在交互式命令行中定义一个新函数
def add(a, b):
    return a + b
print(add.__code__.co_filename) # 输出: <stdin>

import dill
print(dill.source.getsource(add))
# def add(a, b):
#     return a + b
```

#### 内置对象

Python内置函数的源代码是用C语言编写的，只能通过查看Python的源代码（托管在Mercurial或可从<https://www.python.org/downloads/source/>下载）来访问。

```
print(inspect.getsource(sorted)) # 会引发TypeError
type(sorted) # <class 'builtin_function_or_method'>
```

## 第142.3节：探索函数的代码对象

CPython允许访问函数对象的代码对象。

`__code__`对象包含函数的原始字节码（`co_code`）以及其他信息，如常量和变量名。

```
def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)
```

If an object is defined interactively `inspect` cannot provide the source code but you can use `dill.source.getsource` instead

```
# define a new function in the interactive shell
def add(a, b):
    return a + b
print(add.__code__.co_filename) # Output: <stdin>

import dill
print(dill.source.getsource(add))
# def add(a, b):
#     return a + b
```

#### Built-in objects

The source code for Python's built-in functions is written in C and can only be accessed by looking at the Python's source code (hosted on [Mercurial](#) or downloadable from <https://www.python.org/downloads/source/>).

```
print(inspect.getsource(sorted)) # raises a TypeError
type(sorted) # <class 'builtin_function_or_method'>
```

## Section 142.3: Exploring the code object of a function

CPython allows access to the code object for a function object.

The `__code__` object contains the raw bytecode (`co_code`) of the function as well as other information such as constants and variable names.

```
def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)
```

# 第143章：混入（Mixins）

## 第143.1节：混入（Mixin）

混入（Mixin）是一组属性和方法，可以在不同的类中使用，这些属性和方法并不来自于基类。在面向对象编程语言中，通常使用继承来赋予不同类的对象相同的功能；如果一组对象具有某种能力，就将该能力放在它们共同继承的基类中。

例如，假设你有类“汽车（Car）”、“船（Boat）”和“飞机（Plane）”。这些类的对象都有旅行的能力，因此它们拥有“travel”函数。在这种情况下，它们的旅行方式也基本相同；即获取一条路线并沿着这条路线移动。为了实现这个功能，你可以让所有类都继承自“交通工具（Vehicle）”，并将该函数放在这个共享的类中：

```
class Vehicle(object):
    """一个通用的交通工具类."""

    def __init__(self, position):
        self.position = position

    def travel(self, destination):
        route = calculate_route(from_=self.position, to=destination)
        self.move_along(route)

class Car(Vehicle):
    ...

class Boat(Vehicle):
    ...

class Plane(Vehicle):
    ...
```

使用这段代码，你可以在汽车上调用`travel (car.travel("Montana"))`、船上调用`boat.travel("Hawaii")`，以及飞机上调用`plane.travel("France")`

但是，如果你有一些功能在基类中不可用怎么办？比如，你想给Car添加一个收音机，并且能够使用`play_song_on_station`在电台播放歌曲，但你还有一个时钟（Clock）也能使用收音机。Car和Clock可以共享一个基类（Machine）。然而，并非所有机器都能播放歌曲；Boat和Plane不能（至少在这个例子中是这样）。那么如何在不重复代码的情况下实现呢？你可以使用mixin。在Python中，给类添加mixin非常简单，只需将其添加到子类列表中，如下所示

```
class Foo(main_super, mixin): ...
```

Foo将继承main\_super的所有属性和方法，同时也继承mixin的属性和方法。

所以，为了让类Car和时钟具备使用收音机的能力，你可以重写上一个示例中的Car，并写成这样：

```
class RadioUserMixin(object):
    def __init__(self):
        self.radio = Radio()
```

# Chapter 143: Mixins

## Section 143.1: Mixin

A **Mixin** is a set of properties and methods that can be used in different classes, which *don't* come from a base class. In Object Oriented Programming languages, you typically use *inheritance* to give objects of different classes the same functionality; if a set of objects have some ability, you put that ability in a base class that both objects *inherit* from.

For instance, say you have the classes Car, Boat, and Plane. Objects from all of these classes have the ability to travel, so they get the function `travel`. In this scenario, they all travel the same basic way, too; by getting a route, and moving along it. To implement this function, you could derive all of the classes from `Vehicle`, and put the function in that shared class:

```
class Vehicle(object):
    """A generic vehicle class."""

    def __init__(self, position):
        self.position = position

    def travel(self, destination):
        route = calculate_route(from_=self.position, to=destination)
        self.move_along(route)

class Car(Vehicle):
    ...

class Boat(Vehicle):
    ...

class Plane(Vehicle):
    ...
```

With this code, you can call `travel` on a car (`car.travel("Montana")`), boat (`boat.travel("Hawaii")`)，and plane (`plane.travel("France")`)

However, what if you have functionality that's not available to a base class? Say, for instance, you want to give Car a radio and the ability to use it to play a song on a radio station, with `play_song_on_station`, but you also have a clock that can use a radio too. Car and Clock could share a base class (`Machine`). However, not all machines can play songs; Boat and Plane can't (at least in this example). So how do you accomplish without duplicating code? You can use a mixin. In Python, giving a class a mixin is as simple as adding it to the list of subclasses, like this

```
class Foo(main_super, mixin): ...
```

Foo will inherit all of the properties and methods of `main_super`, but also those of `mixin` as well.

So, to give the classes Car and clock the ability to use a radio, you could override Car from the last example and write this:

```
class RadioUserMixin(object):
    def __init__(self):
        self.radio = Radio()
```

```
def play_song_on_station(self, station):
    self.radio.set_station(station)
    self.radio.play_song()
```

```
class Car(Vehicle, RadioUserMixin):
```

```
...
```

```
class Clock(Vehicle, RadioUserMixin):
```

```
...
```

现在你可以调用car.play\_song\_on\_station([98.7](#))和clock.play\_song\_on\_station([101.3](#))，但不能调用类似boat.play\_song\_on\_station([100.5](#))

mixin的重要之处在于，它们允许你为许多不同的对象添加功能，这些对象没有共享带有该功能的“主”子类，但仍然共享这段代码。没有mixin，做上面这样的例子会更困难，和/或可能需要一些重复代码。

## 第143.2节：混入中的重写方法

混入类是一种用于将额外属性和方法“混入”到类中的类。这通常没问题因为很多时候混入类不会重写彼此或基类的方法。但如果你确实在混入类中重写了方法或属性，这可能会导致意想不到的结果，因为在Python中类的继承顺序是从右到左定义的。

例如，考虑以下类

```
类 Mixin1(object):
    定义 test(self):
        打印 "Mixin1"
```

```
类 Mixin2(object):
    定义 test(self):
        打印 "Mixin2"
```

```
类 BaseClass(object):
    定义 test(self):
        打印 "Base"
```

```
类 MyClass(BaseClass, Mixin1, Mixin2):
    通过
```

在这种情况下，Mixin2 类是基类，由 Mixin1 继承，最后由 BaseClass 继承。因此，如果我们执行以下代码片段：

```
>>> x = MyClass()
>>> x.test()
Base
```

我们看到返回的结果来自基类。这可能导致代码逻辑中出现意想不到的错误，需要加以考虑并牢记。

```
def play_song_on_station(self, station):
    self.radio.set_station(station)
    self.radio.play_song()
```

```
class Car(Vehicle, RadioUserMixin):
```

```
...
```

```
class Clock(Vehicle, RadioUserMixin):
```

```
...
```

Now you can call car.play\_song\_on\_station([98.7](#)) and clock.play\_song\_on\_station([101.3](#))，but not something like boat.play\_song\_on\_station([100.5](#))

The important thing with mixins is that they allow you to add functionality to much different objects, that don't share a "main" subclass with this functionality but still share the code for it nonetheless. Without mixins, doing something like the above example would be much harder, and/or might require some repetition.

## Section 143.2: Overriding Methods in Mixins

Mixins are a sort of class that is used to "mix in" extra properties and methods into a class. This is usually fine because many times the mixin classes don't override each other's, or the base class' methods. But if you do override methods or properties in your mixins this can lead to unexpected results because in Python the class hierarchy is defined right to left.

For instance, take the following classes

```
class Mixin1(object):
    def test(self):
        print "Mixin1"
```

```
class Mixin2(object):
    def test(self):
        print "Mixin2"
```

```
class BaseClass(object):
    def test(self):
        print "Base"
```

```
class MyClass(BaseClass, Mixin1, Mixin2):
    pass
```

In this case the Mixin2 class is the base class, extended by Mixin1 and finally by BaseClass. Thus, if we execute the following code snippet:

```
>>> x = MyClass()
>>> x.test()
Base
```

We see the result returned is from the Base class. This can lead to unexpected errors in the logic of your code and needs to be accounted for and kept in mind

# 第144章：属性访问

## 第144.1节：使用点符号的基本属性访问

让我们来看一个示例类。

```
class Book:  
    def __init__(self, title, author):  
        self.title = title  
        self.author = author  
  
book1 = Book(title="Right Ho, Jeeves", author="P.G. Wodehouse")
```

在 Python 中，你可以使用点符号访问类的属性title。

```
>>> book1.title  
'P.G. Wodehouse'
```

如果属性不存在，Python 会抛出错误：

```
>>> book1.series  
回溯 (最近一次调用最后) :  
文件 "<stdin>", 第 1 行, 在 <module>  
AttributeError: 'Book' 对象没有属性 'series'
```

## 第144.2节：设置器、获取器和属性

为了数据封装，有时你希望某个属性的值来自其他属性，或者一般来说，该值应在当时计算。处理这种情况的标准方法是创建一个方法，称为获取器或设置器。

```
class Book:  
    def __init__(self, title, author):  
        self.title = title  
        self.author = author
```

在上面的例子中，很容易看出如果我们创建一个包含标题和作者的新书会发生什么。如果我们要添加到图书馆的所有书都有作者和标题，那么我们可以跳过获取器和设置器，直接使用点符号。

但是，假设有些书没有作者，我们想将作者设置为“未知”。或者如果它们有多个作者，我们计划返回作者列表。

在这种情况下，我们可以为author属性创建获取器和设置器。

```
class P:  
    def __init__(self, title, author):  
        self.title = title  
        self.setAuthor(author)  
  
    def get_author(self):  
        return self.author  
  
    def set_author(self, author):  
        if not author:  
            self.author = "Unknown"  
        else:
```

# Chapter 144: Attribute Access

## Section 144.1: Basic Attribute Access using the Dot Notation

Let's take a sample class.

```
class Book:  
    def __init__(self, title, author):  
        self.title = title  
        self.author = author  
  
book1 = Book(title="Right Ho, Jeeves", author="P.G. Wodehouse")
```

In Python you can access the attribute *title* of the class using the dot notation.

```
>>> book1.title  
'P.G. Wodehouse'
```

If an attribute doesn't exist, Python throws an error:

```
>>> book1.series  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
AttributeError: 'Book' object has no attribute 'series'
```

## Section 144.2: Setters, Getters & Properties

For the sake of data encapsulation, sometimes you want to have an attribute which value comes from other attributes or, in general, which value shall be computed at the moment. The standard way to deal with this situation is to create a method, called getter or a setter.

```
class Book:  
    def __init__(self, title, author):  
        self.title = title  
        self.author = author
```

In the example above, it's easy to see what happens if we create a new Book that contains a title and a author. If all books we're to add to our Library have authors and titles, then we can skip the getters and setters and use the dot notation. However, suppose we have some books that do not have an author and we want to set the author to "Unknown". Or if they have multiple authors and we plan to return a list of authors.

In this case we can create a getter and a setter for the *author* attribute.

```
class P:  
    def __init__(self, title, author):  
        self.title = title  
        self.setAuthor(author)  
  
    def get_author(self):  
        return self.author  
  
    def set_author(self, author):  
        if not author:  
            self.author = "Unknown"  
        else:
```

```
self.author = author
```

此方案不推荐。

一个原因是这里有一个陷阱：假设我们设计的类有一个公共属性且没有方法。人们已经大量使用它，并且他们写了如下代码：

```
>> book = Book(title = "Ancient Manuscript", author = "Some Guy")
>>> book.author = "" # 因为 Some Guy 没有写这本书！
```

现在我们有了一个问题。因为 `author` 不是一个属性！Python 提供了一个解决这个问题的方法，称为属性（properties）。获取属性的方法在其定义前用`@property`装饰。我们希望作为设置器的方法在其前面用`@属性名.setter`装饰。

牢记这一点，我们现在有了更新后的新类。

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    @property
    def author(self):
        return self.__author

    @author.setter
    def author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

注意，通常Python不允许你定义多个同名但参数数量不同的方法。然而，在这种情况下，由于使用了装饰器，Python 允许这样做。

如果我们测试代码：

```
>> book = Book(title = "Ancient Manuscript", author = "Some Guy")
>>> book.author = "" # 因为 Some Guy 没有写这本书！
>>> book.author
Unknown
```

```
self.author = author
```

This scheme is not recommended.

One reason is that there is a catch: Let's assume we have designed our class with the public attribute and no methods. People have already used it a lot and they have written code like this:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = "" #Cos Some Guy didn't write this one!
```

Now we have a problem. Because `author` is not an attribute! Python offers a solution to this problem called properties. A method to get properties is decorated with the `@property` before its header. The method that we want to function as a setter is decorated with `@attributeName.setter` before it.

Keeping this in mind, we now have our new updated class.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    @property
    def author(self):
        return self.__author

    @author.setter
    def author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Note, normally Python doesn't allow you to have multiple methods with the same name and different number of parameters. However, in this case Python allows this because of the decorators used.

If we test the code:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = "" #Cos Some Guy didn't write this one!
>>> book.author
Unknown
```

# 视频：完整的Python训练营：从零开始成为Python 3高手

像专业人士一样学习Python！从基础开始，直到创建你自己的应用程序和游戏！

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ 学习专业使用Python，掌握Python 2和Python 3！
- ✓ 用Python创建游戏，比如井字棋和二十一点！
- ✓ 学习高级Python功能，如collections模块和时间戳的使用！
- ✓ 学习使用面向对象编程和类！
- ✓ 理解复杂主题，如装饰器。
- ✓ 理解如何使用Jupyter Notebook并创建.py文件
- ✓ 了解如何在Jupyter Notebook系统中创建图形用户界面（GUI）！
- ✓ 从零开始全面掌握Python！

今天观看 →

# VIDEO: Complete Python Bootcamp: Go from zero to hero in Python 3

Learn Python like a Professional! Start from the basics and go all the way to creating your own applications and games!

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ Learn to use Python professionally, learning both Python 2 and Python 3!
- ✓ Create games with Python, like Tic Tac Toe and Blackjack!
- ✓ Learn advanced Python features, like the collections module and how to work with timestamps!
- ✓ Learn to use Object Oriented Programming with classes!
- ✓ Understand complex topics, like decorators.
- ✓ Understand how to use both the Jupyter Notebook and create .py files
- ✓ Get an understanding of how to create GUIs in the Jupyter Notebook system!
- ✓ Build a complete understanding of Python from the ground up!

Watch Today →

# 第145章：ArcPy

## 第145.1节：createDissolvedGDB 在工作空间上创建文件地理数据库

```
def createDissolvedGDB(workspace, gdbName):
    gdb_name = workspace + "/" + gdbName + ".gdb"

    if arcpy.Exists(gdb_name):
        arcpy.Delete_management(gdb_name)
        arcpy.CreateFileGDB_management(workspace, gdbName, "")
    else:
        arcpy.CreateFileGDB_management(workspace, gdbName, "")

    return gdb_name
```

## 第145.2节：使用搜索游标打印文件地理数据库中要素类所有行的某个字段值

要打印位于临时文件夹（C:\Temp）中的测试文件地理数据库（Test.gdb）中测试要素类（TestFC）的测试字段（TestField）：

```
使用 arcpy.da.SearchCursor(r"C:\Temp\Test.gdb\TestFC", ["TestField"]) 作为 cursor:
对于 cursor 中的 row:
    打印 row[0]
```

# Chapter 145: ArcPy

## Section 145.1: createDissolvedGDB to create a file gdb on the workspace

```
def createDissolvedGDB(workspace, gdbName):
    gdb_name = workspace + "/" + gdbName + ".gdb"

    if arcpy.Exists(gdb_name):
        arcpy.Delete_management(gdb_name)
        arcpy.CreateFileGDB_management(workspace, gdbName, "")
    else:
        arcpy.CreateFileGDB_management(workspace, gdbName, "")

    return gdb_name
```

## Section 145.2: Printing one field's value for all rows of feature class in file geodatabase using Search Cursor

To print a test field (TestField) from a test feature class (TestFC) in a test file geodatabase (Test.gdb) located in a temporary folder (C:\Temp):

```
with arcpy.da.SearchCursor(r"C:\Temp\Test.gdb\TestFC", ["TestField"]) as cursor:
    for row in cursor:
        print row[0]
```

# 第146章：抽象基类 (abc)

## 第146.1节：设置ABCMeta元类

抽象类是指那些旨在被继承但避免实现具体方法的类，只留下子类必须实现的方法签名。

抽象类对于在高层次上定义和强制类的抽象非常有用，类似于强类型语言中的接口概念，而无需实现方法。

定义抽象类的一种概念性方法是先为类方法设置存根，然后在访问时抛出`NotImplementedError`。这可以防止子类在未先重写父类方法的情况下访问这些方法。示例如下：

```
class Fruit:  
  
    def check_ripeness(self):  
        raise NotImplementedError("check_ripeness 方法未实现！")
```

```
class Apple(Fruit):  
    通过
```

```
a = Apple()  
a.check_ripeness() # 引发 NotImplementedError
```

以这种方式创建抽象类可以防止未重写的方法被不当使用，并且确实鼓励在子类中定义方法，但它并不强制要求定义。使用`abc`模块，当子类未能重写其父类和祖先的抽象类方法时，我们可以防止其实例化：

```
from abc import ABCMeta  
  
class AbstractClass(object):  
    # metaclass 属性必须始终作为类变量设置  
    __metaclass__ = ABCMeta  
  
    # abstractmethod 装饰器将此方法注册为未定义  
    @abstractmethod  
    def virtual_method_subclasses_must_define(self):  
        # 可以完全留空，或者提供一个基础实现  
        # 注意通常空实现会隐式返回 'None'，  
        # 但通过注册，这种行为不再被强制执行。
```

现在可以简单地继承并重写：

```
class Subclass(AbstractClass):  
    def virtual_method_subclasses_must_define(self):  
        return
```

## 第146.2节：为什么/如何使用 ABCMeta 和`@abstractmethod`

抽象基类 (ABC) 强制派生类实现基类中的特定方法。

# Chapter 146: Abstract Base Classes (abc)

## Section 146.1: Setting the ABCMeta metaclass

Abstract classes are classes that are meant to be inherited but avoid implementing specific methods, leaving behind only method signatures that subclasses must implement.

Abstract classes are useful for defining and enforcing class abstractions at a high level, similar to the concept of interfaces in typed languages, without the need for method implementation.

One conceptual approach to defining an abstract class is to stub out the class methods, and then raise a `NotImplementedError` if accessed. This prevents children classes from accessing parent methods without overriding them first. Like so:

```
class Fruit:  
  
    def check_ripeness(self):  
        raise NotImplementedError("check_ripeness method not implemented!")
```

```
class Apple(Fruit):  
    pass
```

```
a = Apple()  
a.check_ripeness() # raises NotImplementedError
```

Creating an abstract class in this way prevents improper usage of methods that are not overridden, and certainly encourages methods to be defined in child classes, but it does not enforce their definition. With the `abc` module we can prevent child classes from being instantiated when they fail to override abstract class methods of their parents and ancestors:

```
from abc import ABCMeta  
  
class AbstractClass(object):  
    # the metaclass attribute must always be set as a class variable  
    __metaclass__ = ABCMeta  
  
    # the abstractmethod decorator registers this method as undefined  
    @abstractmethod  
    def virtual_method_subclasses_must_define(self):  
        # Can be left completely blank, or a base implementation can be provided  
        # Note that ordinarily a blank interpretation implicitly returns 'None',  
        # but by registering, this behaviour is no longer enforced.
```

It is now possible to simply subclass and override:

```
class Subclass(AbstractClass):  
    def virtual_method_subclasses_must_define(self):  
        return
```

## Section 146.2: Why/How to use ABCMeta and`@abstractmethod`

Abstract base classes (ABCs) enforce what derived classes implement particular methods from the base class.

为了理解它是如何工作的以及为什么我们应该使用它，让我们来看一个范·罗斯姆（Van Rossum）会喜欢的例子。假设我们有一个基类“MontyPython”，其中有两个方法（joke 和 punchline）必须由所有派生类实现。

```
class MontyPython:  
    def joke(self):  
        raise NotImplementedError()  
  
    def punchline(self):  
        raise NotImplementedError()  
  
class ArgumentClinic(MontyPython):  
    def joke(self):  
        return "Hahahahah"
```

当我们实例化一个对象并调用它的两个方法时，调用punchline()

```
>>> sketch = ArgumentClinic()  
>>> sketch.punchline()  
NotImplementedError
```

然而，这仍然允许我们实例化ArgumentClinic类的对象而不会出错。事实上，直到我们调用punchline()方法时才会出现错误。

通过使用抽象基类（ABC）模块可以避免这种情况。让我们用同一个例子来看它是如何工作的：

```
from abc import ABCMeta, abstractmethod  
  
class MontyPython(metaclass=ABCMeta):  
    @abstractmethod  
    def joke(self):  
        pass  
  
    @abstractmethod  
    def punchline(self):  
        pass  
  
class ArgumentClinic(MontyPython):  
    def joke(self):  
        return "Hahahahah"
```

这一次，当我们尝试从不完整的类实例化对象时，会立即得到一个TypeError！

```
>>> c = ArgumentClinic()  
TypeError:  
"无法实例化包含抽象方法 punchline 的抽象类 ArgumentClinic"
```

在这种情况下，完成类定义以避免任何TypeError是很简单的：

```
class ArgumentClinic(MontyPython):  
    def joke(self):  
        return "Hahahahah"  
  
    def punchline(self):  
        return "派出警察！"
```

这一次，当你实例化对象时，它可以正常工作！

To understand how this works and why we should use it, let's take a look at an example that Van Rossum would enjoy. Let's say we have a Base class "MontyPython" with two methods (joke & punchline) that must be implemented by all derived classes.

```
class MontyPython:  
    def joke(self):  
        raise NotImplementedError()  
  
    def punchline(self):  
        raise NotImplementedError()  
  
class ArgumentClinic(MontyPython):  
    def joke(self):  
        return "Hahahahah"
```

When we instantiate an object and call its two methods, we'll get an error (as expected) with the punchline() method.

```
>>> sketch = ArgumentClinic()  
>>> sketch.punchline()  
NotImplementedError
```

However, this still allows us to instantiate an object of the ArgumentClinic class without getting an error. In fact we don't get an error until we look for the punchline().

This is avoided by using the Abstract Base Class (ABC) module. Let's see how this works with the same example:

```
from abc import ABCMeta, abstractmethod  
  
class MontyPython(metaclass=ABCMeta):  
    @abstractmethod  
    def joke(self):  
        pass  
  
    @abstractmethod  
    def punchline(self):  
        pass  
  
class ArgumentClinic(MontyPython):  
    def joke(self):  
        return "Hahahahah"
```

This time when we try to instantiate an object from the incomplete class, we immediately get a TypeError!

```
>>> c = ArgumentClinic()  
TypeError:  
"Can't instantiate abstract class ArgumentClinic with abstract methods punchline"
```

In this case, it's easy to complete the class to avoid any TypeErrors:

```
class ArgumentClinic(MontyPython):  
    def joke(self):  
        return "Hahahahah"  
  
    def punchline(self):  
        return "Send in the constable!"
```

This time when you instantiate an object it works!

# 第147章：插件和扩展类

## 第147.1节：混入（Mixins）

在面向对象编程语言中， mixin 是一个包含供其他类使用的方法的类，而不必成为那些其他类的父类。其他类如何访问 mixin 的方法取决于具体的编程语言。

它通过允许多个类使用公共功能，提供了一种多重继承的机制，但没有多重继承复杂的语义。当程序员想在不同类之间共享功能时， mixin 非常有用。与其反复编写相同的代码，不如将公共功能简单地归纳到一个 mixin 中，然后让每个需要该功能的类继承它。

当我们使用多个 mixin 时， mixin 的顺序很重要。这里有一个简单的例子：

```
类 Mixin1(object):
    定义 test(self):
        打印 "Mixin1"
```

```
类 Mixin2(object):
    定义 test(self):
        打印 "Mixin2"
```

```
class MyClass(Mixin1, Mixin2):
    通过
```

在这个例子中，我们调用了MyClass和 test方法，

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

结果必须是 Mixin1，因为顺序是从左到右。当与超类一起使用时，这可能会导致意想不到的结果。所以反向顺序更好，就像这样：

```
class MyClass(Mixin2, Mixin1):
    通过
```

结果将是：

```
>>> obj = MyClass()
>>> obj.test()
Mixin2
```

Mixin 可以用来定义自定义插件。

Python 3.x 版本 ≥ 3.0

```
class Base(object):
    定义 test(self):
        print("Base.")
```

```
class PluginA(object):
    定义 test(self):
        super().test()
        print("Plugin A.")
```

# Chapter 147: Plugin and Extension Classes

## Section 147.1: Mixins

In Object oriented programming language, a mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes. How those other classes gain access to the mixin's methods depends on the language.

It provides a mechanism for multiple inheritance by allowing multiple classes to use the common functionality, but without the complex semantics of multiple inheritance. Mixins are useful when a programmer wants to share functionality between different classes. Instead of repeating the same code over and over again, the common functionality can simply be grouped into a mixin and then inherited into each class that requires it.

When we use more than one mixins, Order of mixins are important. here is a simple example:

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class MyClass(Mixin1, Mixin2):
    pass
```

In this example we call MyClass and test method,

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

Result must be Mixin1 because Order is left to right. This could be show unexpected results when super classes add with it. So reverse order is more good just like this:

```
class MyClass(Mixin2, Mixin1):
    pass
```

Result will be:

```
>>> obj = MyClass()
>>> obj.test()
Mixin2
```

Mixins can be used to define custom plugins.

Python 3.x Version ≥ 3.0

```
class Base(object):
    def test(self):
        print("Base.")
```

```
class PluginA(object):
    def test(self):
        super().test()
        print("Plugin A.")
```

```
class PluginB(object):
    定义 test(self):
        super().test()
        print("Plugin B.")
```

```
plugins = PluginA, PluginB
```

```
class PluginSystemA(PluginA, Base):
    通过
```

```
class PluginSystemB(PluginB, Base):
    通过
```

```
PluginSystemA().test()
# Base.
# Plugin A.
```

```
PluginSystemB().test()
# Base.
# Plugin B.
```

## 第147.2节：带有自定义类的插件

在Python 3.6中，[PEP 487](#) 添加了`__init_subclass__`特殊方法，该方法简化并扩展了类的自定义，无需使用元类。因此，该功能允许创建简单插件。这里我们通过修改之前的示例来演示此功能：

Python 3.x 版本 ≥ 3.6

```
class Base:
    plugins = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.plugins.append(cls)

    定义 test(self):
        print("Base.")
```

```
class PluginA(Base):
    定义 test(self):
        super().test()
        print("Plugin A.")
```

```
class PluginB(Base):
    定义 test(self):
        super().test()
        print("Plugin B.")
```

结果：

```
PluginA().test()
# Base.
# Plugin A.
```

```
PluginB().test()
# Base.
# Plugin B.
```

```
Base.plugins
```

```
class PluginB(object):
    def test(self):
        super().test()
        print("Plugin B.")
```

```
plugins = PluginA, PluginB
```

```
class PluginSystemA(PluginA, Base):
    pass
```

```
class PluginSystemB(PluginB, Base):
    pass
```

```
PluginSystemA().test()
# Base.
# Plugin A.
```

```
PluginSystemB().test()
# Base.
# Plugin B.
```

## Section 147.2: Plugins with Customized Classes

In Python 3.6, [PEP 487](#) added the `__init_subclass__` special method, which simplifies and extends class customization without using metaclasses. Consequently, this feature allows for creating [simple plugins](#). Here we demonstrate this feature by modifying a prior example:

Python 3.x Version ≥ 3.6

```
class Base:
    plugins = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.plugins.append(cls)

    def test(self):
        print("Base.")

class PluginA(Base):
    def test(self):
        super().test()
        print("Plugin A.")

class PluginB(Base):
    def test(self):
        super().test()
        print("Plugin B.")
```

Results:

```
PluginA().test()
# Base.
# Plugin A.
```

```
PluginB().test()
# Base.
# Plugin B.
```

```
Base.plugins
```

```
# __main__.PluginA, __main__.PluginB
```

```
# __main__.PluginA, __main__.PluginB
```

# 第148章：不可变数据类型 (int、float、str、tuple 和 frozenset)

## 第148.1节：字符串的单个字符不可赋值

```
foo = "bar"  
foo[0] = "c" # 错误
```

不可变变量的值一旦创建后就不能更改。

## 第148.2节：元组的单个成员不可赋值

```
foo = ("bar", 1, "Hello!"),  
foo[1] = 2 # 错误！！
```

第二行会返回错误，因为元组的成员一旦创建后不可赋值。这是由于元组的不可变性。

## 第148.3节：frozenset 是不可变且不可赋值的

```
foo = frozenset(["bar", 1, "Hello!"])  
foo[2] = 7 # 错误  
foo.add(3) # 错误
```

第二行会返回错误，因为frozenset成员一旦创建就不能被赋值。第三行会返回错误，因为frozenset不支持可以操作成员的函数。

# Chapter 148: Immutable datatypes(int, float, str, tuple and frozensets)

## Section 148.1: Individual characters of strings are not assignable

```
foo = "bar"  
foo[0] = "c" # Error
```

Immutable variable value can not be changed once they are created.

## Section 148.2: Tuple's individual members aren't assignable

```
foo = ("bar", 1, "Hello!"),  
foo[1] = 2 # ERROR!!
```

Second line would return an error since tuple members once created aren't assignable. Because of tuple's immutability.

## Section 148.3: Frozenset's are immutable and not assignable

```
foo = frozenset(["bar", 1, "Hello!"])  
foo[2] = 7 # ERROR  
foo.add(3) # ERROR
```

Second line would return an error since frozenset members once created aren't assignable. Third line would return error as frozensets do not support functions that can manipulate members.

# 第149章：从Python 2迁移到Python 3的不兼容性

与大多数语言不同，Python支持两个主要版本。自2008年Python 3发布以来，许多人已经完成了迁移，而许多人尚未迁移。为了理解这两者，本节涵盖了Python 2和Python 3之间的重要差异。

## 第149.1节：整数除法

标准的除法符号 (/) 在Python 3和Python 2中应用于整数时表现不同。

在Python 3中，用一个整数除以另一个整数时，除法操作 `x / y` 表示真除法（使用`_truediv_`方法），并产生浮点数结果。与此同时，Python 2中的相同操作表示经典除法，会将结果向负无穷方向取整（也称为取floor）。

例如：

代码	Python 2 输出	Python 3 输出
<code>3 / 21</code>	1.5	
<code>2 / 30</code>	0.6666666666666666	
<code>-3 / 2</code>	-2	-1.5

向零舍入行为在Python 2.2中被弃用，但为了向后兼容，仍保留在Python 2.7中，并在Python 3中被移除。

注意：在Python 2中要获得float结果（无向下取整），可以将其中一个操作数指定为带小数点的数。上述`2/3`在Python 2中结果为0的例子，应写成`2 / 3.0`或`2.0 / 3`或`2.0/3.0`，才能得到 `0.6666666666666666`

代码	Python 2 输出	Python 3 输出
<code>3.0 / 2.0</code>	1.5	
<code>2 / 3.0</code>	0.6666666666666666	0.6666666666666666
<code>-3.0 / 2</code>	-1.5	-1.5

还有地板除法运算符 (//)，它在两个版本中工作方式相同：向下取整到最接近的整数。（虽然与浮点数一起使用时会返回浮点数）在两个版本中，// 运算符对应于 `_floordiv_`。

代码	Python 2 输出	Python 3 输出
<code>3 // 2</code>	1	1
<code>2 // 3</code>	0	0
<code>-3 // 2</code>	-2	-2
<code>3.0 // 2.0</code>	1.0	
<code>2.0 // 3</code>	0.0	0.0
<code>-3 // 2.0</code>	-2.0	-2.0

可以使用operator模块中的原生函数显式地执行真除法或地板除法：

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25          # 等同于 Python 3 中的 `/`
assert floordiv(10, 8) == 1             # 等同于 `//`
```

## Chapter 149: Incompatibilities moving from Python 2 to Python 3

Unlike most languages, Python supports two major versions. Since 2008 when Python 3 was released, many have made the transition, while many have not. In order to understand both, this section covers the important differences between Python 2 and Python 3.

### Section 149.1: Integer Division

The standard **division symbol** (/) operates differently in Python 3 and Python 2 when applied to integers.

When dividing an integer by another integer in Python 3, the division operation `x / y` represents a **true division** (uses `_truediv_` method) and produces a floating point result. Meanwhile, the same operation in Python 2 represents a **classic division** that rounds the result down toward negative infinity (also known as taking the `floor`).

For example:

Code	Python 2 output	Python 3 output
<code>3 / 2</code>	1	1.5
<code>2 / 3</code>	0	0.6666666666666666
<code>-3 / 2</code>	-2	-1.5

The rounding-towards-zero behavior was deprecated in [Python 2.2](#), but remains in Python 2.7 for the sake of backward compatibility and was removed in Python 3.

**Note:** To get a *float* result in Python 2 (without floor rounding) we can specify one of the operands with the decimal point. The above example of `2/3` which gives 0 in Python 2 shall be used as `2 / 3.0` or `2.0 / 3` or `2.0/3.0` to get `0.6666666666666666`

Code	Python 2 output	Python 3 output
<code>3.0 / 2.0</code>	1.5	1.5
<code>2 / 3.0</code>	0.6666666666666666	0.6666666666666666
<code>-3.0 / 2</code>	-1.5	-1.5

There is also the **floor division operator** (//), which works the same way in both versions: it rounds down to the nearest integer. (although a float is returned when used with floats) In both versions the // operator maps to `_floordiv_`.

Code	Python 2 output	Python 3 output
<code>3 // 2</code>	1	1
<code>2 // 3</code>	0	0
<code>-3 // 2</code>	-2	-2
<code>3.0 // 2.0</code>	1.0	1.0
<code>2.0 // 3</code>	0.0	0.0
<code>-3.0 // 2.0</code>	-2.0	-2.0

One can explicitly enforce true division or floor division using native functions in the `operator` module:

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25          # 等同于 `/` 在 Python 3 中
assert floordiv(10, 8) == 1             # 等同于 `//`
```

虽然使用操作符函数清晰且明确，但每次除法都使用它们会很繁琐。通常更倾向于改变 / 操作符的行为。一个常见做法是在每个模块的第一条语句中添加 `from __future__ import division`：

```
# 必须是模块中的第一条语句
from __future__ import division

代码 Python 2 输出 Python 3输出
3 / 21.5 1.5
2 / 30.6666666666666666 0.6666666666666666
-3 / 2 -1.5 -1.5
```

`from __future__ import division` 保证了 / 运算符表示真正的除法，并且仅在包含 `__future__` 导入的模块中生效，因此没有充分理由不在所有新模块中启用它。

注意：一些其他编程语言使用 向零舍入（截断），而不是像 Python 那样 向负无穷舍入（即在那些语言中  $-3 / 2 == -1$ ）。这种行为在移植或比较代码时可能会引起混淆。

关于浮点操作数的说明：作为 `from __future__ import division` 的替代方案，可以使用常规除法符号 / 并确保至少一个操作数是浮点数： $3 / 2.0 == 1.5$ 。然而，这被认为是不好的做法。因为很容易写成 `average = sum(items) / len(items)` 而忘记将其中一个参数转换为浮点数。此外，这种情况在测试时可能经常被忽视，例如，如果你在包含 float 的数组上测试，但在生产环境中收到的是 int 数组。另外，如果相同代码用于 Python 3，期望  $3 / 2 == 1$  为真的程序将无法正常工作。

请参见 [PEP 238](#) 了解更多关于为什么在 Python 3 中更改除法运算符以及为什么应避免使用旧式除法的详细理由。

请参见 简单数学 主题，了解更多关于除法的内容。

## 第149.2节：解包可迭代对象

Python 3.x 版本  $\geq 3.0$

在 Python 3 中，你可以在不知道可迭代对象确切元素数量的情况下进行解包，甚至可以用一个变量来保存可迭代对象的剩余部分。为此，你需要提供一个变量来收集值列表。这是通过在变量名前加星号来实现的。例如，解包一个 列表：

```
first, second, *tail, last = [1, 2, 3, 4, 5]
print(first)
# 输出: 1
print(second)
# 输出: 2
print(tail)
# 输出: [3, 4]
print(last)
# 输出: 5
```

注意：使用`*variable`语法时，`variable`始终是一个列表，即使原始类型不是列表。它可能包含零个或多个元素，具体取决于原始列表中的元素数量。

```
first, second, *tail, last = [1, 2, 3, 4]
print(tail)
```

While clear and explicit, using operator functions for every division can be tedious. Changing the behavior of the / operator will often be preferred. A common practice is to eliminate typical division behavior by adding `from __future__ import division` as the first statement in each module:

```
# needs to be the first statement in a module
from __future__ import division

Code Python 2 output Python 3 output
3 / 2 1.5 1.5
2 / 3 0.6666666666666666 0.6666666666666666
-3 / 2 -1.5 -1.5
```

`from __future__ import division` guarantees that the / operator represents true division and only within the modules that contain the `__future__` import, so there are no compelling reasons for not enabling it in all new modules.

**Note:** Some other programming languages use *rounding toward zero* (truncation) rather than *rounding down toward negative infinity* as Python does (i.e. in those languages  $-3 / 2 == -1$ ). This behavior may create confusion when porting or comparing code.

**Note on float operands:** As an alternative to `from __future__ import division`, one could use the usual division symbol / and ensure that at least one of the operands is a float:  $3 / 2.0 == 1.5$ . However, this can be considered bad practice. It is just too easy to write `average = sum(items) / len(items)` and forget to cast one of the arguments to float. Moreover, such cases may frequently evade notice during testing, e.g., if you test on an array containing `floats` but receive an array of `ints` in production. Additionally, if the same code is used in Python 3, programs that expect  $3 / 2 == 1$  to be True will not work correctly.

See [PEP 238](#) for more detailed rationale why the division operator was changed in Python 3 and why old-style division should be avoided.

See the *Simple Math* topic for more about division.

## Section 149.2: Unpacking Iterables

Python 3.x Version  $\geq 3.0$

In Python 3, you can unpack an iterable without knowing the exact number of items in it, and even have a variable hold the end of the iterable. For that, you provide a variable that may collect a list of values. This is done by placing an asterisk before the name. For example, unpacking a `list`:

```
first, second, *tail, last = [1, 2, 3, 4, 5]
print(first)
# Out: 1
print(second)
# Out: 2
print(tail)
# Out: [3, 4]
print(last)
# Out: 5
```

**Note:** When using the `*variable` syntax, the variable will always be a list, even if the original type wasn't a list. It may contain zero or more elements depending on the number of elements in the original list.

```
first, second, *tail, last = [1, 2, 3, 4]
print(tail)
```

```
# 输出: [3]
```

```
first, second, *tail, last = [1, 2, 3]
print(tail)
# 输出: []
print(last)
# 输出: 3
```

类似地，解包一个str:

```
begin, *tail = "Hello"
print(begin)
# 输出: 'H'
print(tail)
# 输出: ['e', 'l', 'l', 'o']
```

解包日期的示例；本例中使用\_作为丢弃变量（我们只关心年份的值）：

```
person = ('John', 'Doe', (10, 16, 2016))
*, (_, year_of_birth) = person
print(year_of_birth)
# 输出: 2016
```

值得一提的是，由于\*可以接收可变数量的元素，赋值时不能对同一个可迭代对象使用两个\*——这样会导致无法判断第一个解包有多少元素，第二个又有多少元素：

```
*head, *tail = [1, 2]
# 输出: 语法错误：赋值中有两个带星号的表达式
```

Python 3.x 版本 ≥ 3.5

到目前为止，我们已经讨论了赋值中的解包。Python 3.5 中扩展了\*和\*\*。现在可以在一个表达式中进行多个解包操作：

```
{*range(4), 4, *(5, 6, 7)}
# 输出: {0, 1, 2, 3, 4, 5, 6, 7}
```

Python 2.x 版本 ≥ 2.0

也可以将可迭代对象拆包为函数参数：

```
iterable = [1, 2, 3, 4, 5]
print(iterable)
# 输出: [1, 2, 3, 4, 5]
print(*iterable)
# 输出: 1 2 3 4 5
```

Python 3.x 版本 ≥ 3.5

拆包字典使用两个相邻的星号 \*\* (PEP 448)：

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}
# 输出: {'x': 1, 'y': 2, 'z': 3}
```

这既允许覆盖旧值，也允许合并字典。

```
dict1 = {'x': 1, 'y': 1}
```

```
# Out: [3]
```

```
first, second, *tail, last = [1, 2, 3]
print(tail)
# Out: []
print(last)
# Out: 3
```

Similarly, unpacking a str:

```
begin, *tail = "Hello"
print(begin)
# Out: 'H'
print(tail)
# Out: ['e', 'l', 'l', 'o']
```

Example of unpacking a date; \_ is used in this example as a throwaway variable (we are interested only in year value):

```
person = ('John', 'Doe', (10, 16, 2016))
*, (_, year_of_birth) = person
print(year_of_birth)
# Out: 2016
```

It is worth mentioning that, since \* eats up a variable number of items, you cannot have two \*s for the same iterable in an assignment - it wouldn't know how many elements go into the first unpacking, and how many in the second:

```
*head, *tail = [1, 2]
# Out: SyntaxError: two starred expressions in assignment
```

Python 3.x 版本 ≥ 3.5

So far we have discussed unpacking in assignments. \* and \*\* were [extended in Python 3.5](#). It's now possible to have several unpacking operations in one expression:

```
{*range(4), 4, *(5, 6, 7)}
# Out: {0, 1, 2, 3, 4, 5, 6, 7}
```

Python 2.x 版本 ≥ 2.0

It is also possible to unpack an iterable into function arguments:

```
iterable = [1, 2, 3, 4, 5]
print(iterable)
# Out: [1, 2, 3, 4, 5]
print(*iterable)
# Out: 1 2 3 4 5
```

Python 3.x 版本 ≥ 3.5

Unpacking a dictionary uses two adjacent stars \*\* (PEP 448):

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}
# Out: {'x': 1, 'y': 2, 'z': 3}
```

This allows for both overriding old values and merging dictionaries.

```
dict1 = {'x': 1, 'y': 1}
```

```
dict2 = {'y': 2, 'z': 3}
{**dict1, **dict2}
# 输出: {'x': 1, 'y': 2, 'z': 3}
```

Python 3.x 版本 ≥ 3.0

Python 3 移除了函数中的元组解包。因此，以下代码在 Python 3 中无法运行

```
# 在 Python 2 中可用，但在 Python 3 中语法错误：
map(lambda (x, y): x + y, zip(range(5), range(5)))
# 非 lambda 函数同理：
```

```
def example((x, y)):
    通过
```

```
# 在 Python 2 和 Python 3 中均可用：
map(lambda x: x[0] + x[1], zip(range(5), range(5)))
# 非 lambda 函数也一样：
```

```
def working_example(x_y):
    x, y = x_y
    通过
```

详见PEP 3113 以了解详细理由。

## 第149.3节：字符串：字节与Unicode

Python 2.x 版本 ≤ 2.7

在Python 2中，字符串有两种变体：由字节组成的类型为`str`的字符串和由文本组成的类型为`unicode`的字符串。

在Python 2中，类型为`str`的对象始终是字节序列，但通常用于文本和二进制数据。

字符串字面量被解释为字节字符串。

```
s = 'Café'    # type(s) == str
```

有两个例外：你可以通过在字面量前加上l显式定义一个Unicode（文本）字面量：

```
s = u'Café'    # type(s) == unicode
b = 'Lorem ipsum' # type(b) == str
```

或者，你可以指定整个模块的字符串字面量应创建Unicode（文本）字面量：

```
from __future__ import unicode_literals

s = 'Café'    # type(s) == unicode
b = 'Lorem ipsum' # type(b) == unicode
```

为了检查你的变量是否是字符串（无论是Unicode还是字节字符串），你可以使用：

```
isinstance(s, basestring)
```

Python 3.x 版本 ≥ 3.0

在Python 3中，`str`类型是Unicode文本类型。

```
s = 'Café'      # type(s) == str
```

```
dict2 = {'y': 2, 'z': 3}
{**dict1, **dict2}
# Out: {'x': 1, 'y': 2, 'z': 3}
```

Python 3.x 版本 ≥ 3.0

Python 3 移除了函数中的元组解包。因此，以下代码在 Python 3 中无法运行

```
# Works in Python 2, but syntax error in Python 3:
map(lambda (x, y): x + y, zip(range(5), range(5)))
# Same is true for non-lambdas:
```

```
def example((x, y)):
    pass
```

```
# Works in both Python 2 and Python 3:
map(lambda x: x[0] + x[1], zip(range(5), range(5)))
# And non-lambdas, too:
```

```
def working_example(x_y):
    x, y = x_y
    pass
```

See PEP 3113 for detailed rationale.

## Section 149.3: Strings: Bytes versus Unicode

Python 2.x 版本 ≤ 2.7

In Python 2 there are two variants of string: those made of bytes with type (`str`) and those made of text with type (`unicode`).

In Python 2, an object of type `str` is always a byte sequence, but is commonly used for both text and binary data.

A string literal is interpreted as a byte string.

```
s = 'Café'    # type(s) == str
```

There are two exceptions: You can define a *Unicode (text) literal* explicitly by prefixing the literal with `u`:

```
s = u'Café'    # type(s) == unicode
b = 'Lorem ipsum' # type(b) == str
```

Alternatively, you can specify that a whole module's string literals should create Unicode (text) literals:

```
from __future__ import unicode_literals

s = 'Café'    # type(s) == unicode
b = 'Lorem ipsum' # type(b) == unicode
```

In order to check whether your variable is a string (either Unicode or a byte string), you can use:

```
isinstance(s, basestring)
```

Python 3.x 版本 ≥ 3.0

In Python 3, the `str` type is a Unicode text type.

```
s = 'Café'      # type(s) == str
```

```
s = 'Café'      # type(s) == str (注意带重音符的结尾e)
```

此外，Python 3增加了一个bytes对象，适用于二进制“数据块”或写入与编码无关的文件。要创建bytes对象，可以在字符串字面量前加b前缀，或调用字符串的encode方法：

```
# 或者，如果你确实需要一个字节字符串：  
s = b'Café'      # type(s) == bytes  
s = 'Café'.encode()  # type(s) == bytes
```

要测试一个值是否为字符串，使用：

```
isinstance(s, str)
```

Python 3.x 版本 ≥ 3.3

也可以在字符串字面量前加上 u 前缀，以便在 Python 2 和 Python 3 代码库之间兼容。由于在 Python 3 中，所有字符串默认都是 Unicode，给字符串字面量加上 u 前缀没有效果：

```
u'Café' == 'Café'
```

Python 2的原始 Unicode 字符串前缀 ur 不被支持：

```
>>> ur'Café'  
文件 "<stdin>", 第 1 行  
ur'Café'
```

语法错误: 无效的语法

注意，必须对 Python 3 的文本 (str) 对象进行 encode 操作，才能将其转换为该文本的 bytes 表示形式。该方法的默认编码是 UTF-8。

你可以使用decode来询问一个bytes对象表示的Unicode文本内容：

```
>>> b.decode()  
'Café'
```

Python 2.x 版本 ≥ 2.6

虽然bytes类型在Python 2和3中都存在，unicode类型仅存在于Python 2中。要在Python 2中使用Python 3的隐式Unicode字符串，请在代码文件顶部添加以下内容：

```
from __future__ import unicode_literals  
print(repr("hi"))  
# u'hi'
```

Python 3.x 版本 ≥ 3.0

另一个重要区别是，在Python 3中对bytes进行索引会得到一个int类型的输出，如下所示：

```
b"abc"[0] == 97
```

而切片大小为1时，结果是一个长度为1的bytes对象：

```
b"abc"[0:1] == b"a"
```

此外，Python 3修复了一些与Unicode相关的异常行为，例如Python 2中反转字节字符串的问题。例如，以下问题已被解决：

```
s = 'Café'      # type(s) == str (note the accented trailing e)
```

Additionally, Python 3 added a [bytes object](#), suitable for binary "blobs" or writing to encoding-independent files. To create a bytes object, you can prefix b to a string literal or call the string's encode method:

```
# Or, if you really need a byte string:  
s = b'Café'      # type(s) == bytes  
s = 'Café'.encode()  # type(s) == bytes
```

To test whether a value is a string, use:

```
isinstance(s, str)
```

Python 3.x 版本 ≥ 3.3

It is also possible to prefix string literals with a u prefix to ease compatibility between Python 2 and Python 3 code bases. Since, in Python 3, all strings are Unicode by default, prepending a string literal with u has no effect:

```
u'Café' == 'Café'
```

Python 2's raw Unicode string prefix ur is not supported, however:

```
>>> ur'Café'  
File "<stdin>", line 1  
    ur'Café'  
          ^  
SyntaxError: invalid syntax
```

Note that you must [encode](#) a Python 3 text ([str](#)) object to convert it into a [bytes](#) representation of that text. The default encoding of this method is [UTF-8](#).

You can use [decode](#) to ask a [bytes](#) object for what Unicode text it represents:

```
>>> b.decode()  
'Café'
```

Python 2.x 版本 ≥ 2.6

While the [bytes](#) type exists in both Python 2 and 3, the [unicode](#) type only exists in Python 2. To use Python 3's implicit Unicode strings in Python 2, add the following to the top of your code file:

```
from __future__ import unicode_literals  
print(repr("hi"))  
# u'hi'
```

Python 3.x 版本 ≥ 3.0

Another important difference is that indexing bytes in Python 3 results in an [int](#) output like so:

```
b"abc"[0] == 97
```

Whilst slicing in a size of one results in a length 1 bytes object:

```
b"abc"[0:1] == b"a"
```

In addition, Python 3 [fixes some unusual behavior](#) with unicode, i.e. reversing byte strings in Python 2. For example, the [following issue](#) is resolved:

```

# -*- coding: utf8 -*-
print("嗨, 我的名字是Łukasz Langa.")
print(u"嗨, 我的名字是Łukasz Langa."[:-1])
print("嗨, 我的名字是Łukasz Langa."[:-1])

# Python 2 中的输出
# 嗨, 我的名字是Łukasz Langa.
# .agnaL zsakuł 是我的名字, 嗨
# .agnaL zsaku♦♦ 是我的名字, 嗨

# Python 3 中的输出
# 嗨, 我的名字是Łukasz Langa.
# .agnaL zsakuł 是我的名字, 嗨
# .agnaL zsakuł 是我的名字, 嗨

```

## 第149.4节：print语句与print函数

在Python 2中，`print`是一个语句：

```

Python 2.x 版本 ≤ 2.7
print "Hello World"
print          # 打印一个换行符
print "无换行",      # 添加尾随逗号以去除换行符
print >>sys.stderr, "错误"  # 打印到标准错误输出
print("hello")        # 打印 "hello", 因为 ("hello") == "hello"
print()              # 打印一个空元组 "()"
print 1, 2, 3         # 打印以空格分隔的参数："1 2 3"
print(1, 2, 3)        # 打印元组 "(1, 2, 3)"

```

在 Python 3 中，`print()` 是一个函数，带有用于常见用途的关键字参数：

```

Python 3.x 版本 ≥ 3.0
print "Hello World"      # 语法错误
print("Hello World")
print()                  # 打印换行符 (必须使用括号)
print("No newline", end="") # end 指定追加内容 (默认为换行符)
print("Error", file=sys.stderr) # file 指定输出缓冲区
print("Comma", "separated", "output", sep=",") # sep 指定分隔符
print("A", "B", "C", sep="") # sep 为空字符串：打印为 ABC
print("Flush this", flush=True) # 刷新输出缓冲区, Python 3.3 新增
print(1, 2, 3)            # 打印以空格分隔的参数："1 2 3"
print((1, 2, 3))          # 打印元组 "(1, 2, 3)"

```

`print` 函数具有以下参数：

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`sep` 是分隔传递给 `print` 的对象的分隔符。例如：

```

print('foo', 'bar', sep='~') # 输出: foo~bar
print('foo', 'bar', sep='.') # 输出: foo.bar

```

`end` 是打印语句后面跟随的内容。例如：

```
print('foo', 'bar', end='!') # 输出: foo bar!
```

```

# -*- coding: utf8 -*-
print("Hi, my name is Łukasz Langa.")
print(u"Hi, my name is Łukasz Langa."[:-1])
print("Hi, my name is Łukasz Langa."[:-1])

# Output in Python 2
# Hi, my name is Łukasz Langa.
# .agnaL zsakuł si eman ym ,iH
# .agnaL zsaku♦♦ si eman ym ,iH

# Output in Python 3
# Hi, my name is Łukasz Langa.
# .agnaL zsakuł si eman ym ,iH
# .agnaL zsakuł si eman ym ,iH

```

## Section 149.4: Print statement vs. Print function

In Python 2, `print` is a statement:

```

Python 2.x Version ≤ 2.7
print "Hello World"
print          # print a newline
print "No newline",      # add trailing comma to remove newline
print >>sys.stderr, "Error"  # print to stderr
print("hello")        # print "hello", since ("hello") == "hello"
print()              # print an empty tuple "()"
print 1, 2, 3         # print space-separated arguments: "1 2 3"
print(1, 2, 3)        # print tuple "(1, 2, 3)"

```

In Python 3, `print()` is a function, with keyword arguments for common uses:

```

Python 3.x Version ≥ 3.0
print "Hello World"      # SyntaxError
print("Hello World")
print()                  # print a newline (must use parentheses)
print("No newline", end="") # end specifies what to append (defaults to newline)
print("Error", file=sys.stderr) # file specifies the output buffer
print("Comma", "separated", "output", sep=",") # sep specifies the separator
print("A", "B", "C", sep="") # null string for sep: prints as ABC
print("Flush this", flush=True) # flush the output buffer, added in Python 3.3
print(1, 2, 3)            # print space-separated arguments: "1 2 3"
print((1, 2, 3))          # print tuple "(1, 2, 3)"

```

The `print` function has the following parameters:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`sep` is what separates the objects you pass to `print`. For example:

```

print('foo', 'bar', sep='~') # out: foo~bar
print('foo', 'bar', sep='.') # out: foo.bar

```

`end` is what the end of the `print` statement is followed by. For example:

```
print('foo', 'bar', end='!') # out: foo bar!
```

紧跟非换行结尾的打印语句再次打印将打印在同一行：

```
print('foo', end='~')
print('bar')
# 输出: foo~bar
```

注意：为了未来兼容性，print 函数从 Python 2.6 开始也可用；但除非禁用对print 语句的解析，否则不能使用

```
from __future__ import print_function
```

该函数格式与 Python 3 的完全相同，只是缺少了flush参数。

详见 PEP 3105 的理由说明。

## 第149.5节：range和xrange函数的区别

在Python 2中，range函数返回一个列表，而 xrange创建一个特殊的 xrange对象，这是一个不可变的序列，与其他内置序列类型不同，它不支持切片，也没有 index和 count方法：

```
Python 2.x 版本 ≥ 2.3
print(range(1, 10))
# 输出: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print(isinstance(range(1, 10), list))
# 输出: True
```

```
print(xrange(1, 10))
# 输出: xrange(1, 10)
```

```
print(isinstance(xrange(1, 10), xrange))
# 输出: True
```

在Python 3中，xrange被扩展为range序列，因此现在创建的是一个range对象。不存在 xrange类型：

```
Python 3.x 版本 ≥ 3.0
```

```
print(range(1, 10))
# 输出: range(1, 10)
```

```
print(isinstance(range(1, 10), range))
# 输出: True
```

```
# print(xrange(1, 10))
# 输出将是：
#Traceback (most recent call last):
# 文件 "<stdin>", 第 1 行, 位于 <module>
#NameError: 名称 'xrange' 未定义
```

此外，自 Python 3.2 起，range 也支持切片、index 和 count：

```
print(range(1, 10)[3:7])
# 输出: range(3, 7)
print(range(1, 10).count(5))
# 输出: 1
```

Printing again following a non-newline ending print statement will print to the same line:

```
print('foo', end='~')
print('bar')
# out: foo~bar
```

**Note:** For future compatibility, `print` function is also available in Python 2.6 onwards; however it cannot be used unless parsing of the `print` statement is disabled with

```
from __future__ import print_function
```

This function has exactly same format as Python 3's, except that it lacks the flush parameter.

See PEP 3105 for rationale.

## Section 149.5: Differences between range and xrange functions

In Python 2, `range` function returns a list while `xrange` creates a special `xrange` object, which is an immutable sequence, which unlike other built-in sequence types, doesn't support slicing and has neither `index` nor `count` methods:

```
Python 2.x 版本 ≥ 2.3
```

```
print(range(1, 10))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print(isinstance(range(1, 10), list))
# Out: True
```

```
print(xrange(1, 10))
# Out: xrange(1, 10)
```

```
print(isinstance(xrange(1, 10), xrange))
# Out: True
```

In Python 3, `xrange` was expanded to the `range` sequence, which thus now creates a `range` object. There is no `xrange` type:

```
Python 3.x 版本 ≥ 3.0
```

```
print(range(1, 10))
# Out: range(1, 10)
```

```
print(isinstance(range(1, 10), range))
# Out: True
```

```
# print(xrange(1, 10))
# The output will be:
#Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
#NameError: name 'xrange' is not defined
```

Additionally, since Python 3.2, `range` also supports slicing, `index` and `count`:

```
print(range(1, 10)[3:7])
# Out: range(3, 7)
print(range(1, 10).count(5))
# Out: 1
```

```
print(range(1, 10).index(7))
# 输出: 6
```

使用特殊序列类型而非列表的优点是解释器不必为列表分配内存并填充它：

```
Python 2.x 版本 ≥ 2.3
# range(1000000000000000)
# 输出将是:
# Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
# MemoryError

print(xrange(0000000000000000))
# 输出: xrange(1000000000000000)
```

由于后者行为通常是期望的，前者在Python 3中被移除了。如果你仍然想在Python 3中拥有一个列表，可以简单地对一个 `range` 对象使用 `list()` 构造函数：

```
Python 3.x 版本 ≥ 3.0
print(list(range(1, 10)))
# 输出: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## 兼容性

为了保持Python 2.x和Python 3.x版本之间的兼容性，你可以使用外部包 `future` 中的 `builtins` 模块来实现向前兼容和向后兼容：

```
Python 2.x 版本 ≥ 2.0
# 向前兼容
from builtins import range

for i in range(10**8):
    通过
```

```
Python 3.x 版本 ≥ 3.0
# 向后兼容
from past.builtins import xrange

for i in xrange(10**8):
    通过
```

`future` 库中的 `range` 支持切片、`index` 和 `count`，在所有 Python 版本中都可用，就像 Python 3.2+ 中的内置方法一样。

## 第149.6节：引发和处理异常

这是 Python 2 语法，注意在 `raise` 和 `except` 行上的逗号，：

```
Python 2.x 版本 ≥ 2.3
尝试:
    raise IOError, "输入/输出错误"
except IOError, exc:
    print exc
```

在 Python 3 中，，语法被废弃，取而代之的是使用括号和 `as` 关键字：

```
print(range(1, 10).index(7))
# Out: 6
```

The advantage of using a special sequence type instead of a list is that the interpreter does not have to allocate memory for a list and populate it:

```
Python 2.x 版本 ≥ 2.3
# range(1000000000000000)
# The output would be:
# Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
# MemoryError

print(xrange(1000000000000000))
# Out: xrange(1000000000000000)
```

Since the latter behaviour is generally desired, the former was removed in Python 3. If you still want to have a list in Python 3, you can simply use the `list()` constructor on a `range` object:

```
Python 3.x 版本 ≥ 3.0
print(list(range(1, 10)))
# Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## Compatibility

In order to maintain compatibility between both Python 2.x and Python 3.x versions, you can use the `builtins` module from the external package `future` to achieve both *forward-compatibility* and *backward-compatibility*:

```
Python 2.x 版本 ≥ 2.0
# forward-compatible
from builtins import range

for i in range(10**8):
    pass

Python 3.x 版本 ≥ 3.0
# backward-compatible
from past.builtins import xrange

for i in xrange(10**8):
    pass
```

The `range` in `future` library supports slicing, `index` and `count` in all Python versions, just like the built-in method on Python 3.2+.

## Section 149.6: Raising and handling Exceptions

This is the Python 2 syntax, note the commas, on the `raise` and `except` lines:

```
Python 2.x 版本 ≥ 2.3
try:
    raise IOError, "input/output error"
except IOError, exc:
    print exc
```

In Python 3, the , syntax is dropped and replaced by parenthesis and the `as` keyword:

尝试:

```
try:  
    raise IOError("输入/输出错误")  
except IOError as exc:  
    print(exc)
```

为了向后兼容, Python 3 的语法从 Python 2.6 开始也可用, 因此应在所有不需要兼容旧版本的新代码中使用该语法。

Python 3.x 版本  $\geq$  3.0

Python 3 还增加了异常链功能, 可以表明某个其他异常是此异常的原因。例如

尝试:

```
try:  
    file = open('database.db')  
except FileNotFoundError as e:  
    raise DatabaseError('无法打开 {}') from e
```

在 `except` 语句中引发的异常类型是 `DatabaseError`, 但原始异常被标记为该异常的 `__cause__` 属性。当显示追踪信息时, 原始异常也会在追踪信息中显示:

```
追踪 (最近一次调用最后) :  
文件 "", 第 2 行, 位于  
FileNotFoundError
```

上述异常是以下异常的直接原因:

```
追踪 (最近一次调用最后) :  
文件 "", 第 4 行, 位于  
DatabaseError('无法打开 database.db')
```

如果在 `except` 块中抛出异常时 没有 显式链式调用:

```
try:  
    file = open('database.db')  
except FileNotFoundError as e:  
    raise DatabaseError('无法打开 {}')
```

追踪信息为

```
追踪 (最近一次调用最后) :  
文件 "", 第 2 行, 位于  
FileNotFoundError
```

在处理上述异常时, 又发生了另一个异常:

```
追踪 (最近一次调用最后) :  
文件 "", 第 4 行, 位于  
DatabaseError('无法打开 database.db')
```

Python 2.x 版本  $\geq$  2.0

在 Python 2.x 中, 两者都不被支持;如果在 `except` 块中引发另一个异常, 原始异常及其回溯信息将会丢失。以下代码可用于兼容性处理:

try:

```
try:  
    raise IOError("input/output error")  
except IOError as exc:  
    print(exc)
```

For backwards compatibility, the Python 3 syntax is also available in Python 2.6 onwards, so it should be used for all new code that does not need to be compatible with previous versions.

Python 3.x 版本  $\geq$  3.0

Python 3 也增加了异常链功能, 其中你可以指示某些其他异常是此异常的 *cause*。例如

```
try:  
    file = open('database.db')  
except FileNotFoundError as e:  
    raise DatabaseError('Cannot open {}') from e
```

The exception raised in the `except` statement is of type `DatabaseError`, but the original exception is marked as the `__cause__` attribute of that exception. When the traceback is displayed, the original exception will also be displayed in the traceback:

```
Traceback (most recent call last):  
File "", line 2, in  
FileNotFoundException
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):  
File "", line 4, in  
DatabaseError('Cannot open database.db')
```

If you throw in an `except` block without explicit chaining:

```
try:  
    file = open('database.db')  
except FileNotFoundError as e:  
    raise DatabaseError('Cannot open {}')
```

The traceback is

```
Traceback (most recent call last):  
File "", line 2, in  
FileNotFoundException
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):  
File "", line 4, in  
DatabaseError('Cannot open database.db')
```

Python 2.x 版本  $\geq$  2.0

Neither one is supported in Python 2.x; the original exception and its traceback will be lost if another exception is raised in the `except` block. The following code can be used for compatibility:

```

import sys
import traceback

try:
    funcWithError()
except:
    sys_vers = getattr(sys, 'version_info', (0,))
    if sys_vers < (3, 0):
        traceback.print_exc()
    raise Exception("new exception")

```

Python 3.x 版本 ≥ 3.3

要“忘记”之前抛出的异常，请使用 `raise from None`

```

尝试:
file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}'.format(e)) from None

```

现在回溯信息将简单显示为

```

追踪 (最近一次调用最后) :
文件 "", 第 4 行, 位于
DatabaseError('无法打开 database.db')

```

或者为了兼容 Python 2 和 3，你可以使用 `six` 包，方法如下：

```

import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('无法打开 {}'.format(e)), None)

```

## 第149.7节：列表推导式中的变量泄漏

Python 2.x 版本 ≥ 2.3

```

x = 'hello world!'
vowels = [x for x in 'AEIOU']

```

```

print(vowels)
# 输出: ['A', 'E', 'I', 'O', 'U']
print(x)
# 输出: 'U'

```

Python 3.x 版本 ≥ 3.0

```

x = 'hello world!'
vowels = [x for x in 'AEIOU']

```

```

print(vowels)
# 输出: ['A', 'E', 'I', 'O', 'U']
print(x)
# 输出: 'hello world!'

```

从示例中可以看出，在 Python 2 中，变量 `x` 的值被泄漏了：它覆盖了 `hello world!` 并打印出了 `U`，因为这是循环结束时 `x` 的最后一个值。

然而，在 Python 3 中，`x` 会打印最初定义的 `hello world!`，因为它来自列表的局部变量

```

import sys
import traceback

try:
    funcWithError()
except:
    sys_vers = getattr(sys, 'version_info', (0,))
    if sys_vers < (3, 0):
        traceback.print_exc()
    raise Exception("new exception")

```

Python 3.x Version ≥ 3.3

To "forget" the previously thrown exception, use `raise from None`

```

try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}'.format(e)) from None

```

Now the traceback would simply be

```

Traceback (most recent call last):
File "", line 4, in
DatabaseError('无法打开 database.db')

```

Or in order to make it compatible with both Python 2 and 3 you may use the `six` package like so:

```

import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('无法打开 {}'.format(e)), None)

```

## Section 149.7: Leaked variables in list comprehension

Python 2.x Version ≥ 2.3

```

x = 'hello world!'
vowels = [x for x in 'AEIOU']

```

```

print(vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'U'

```

Python 3.x Version ≥ 3.0

```

x = 'hello world!'
vowels = [x for x in 'AEIOU']

```

```

print(vowels)
# Out: ['A', 'E', 'I', 'O', 'U']
print(x)
# Out: 'hello world!'

```

As can be seen from the example, in Python 2 the value of `x` was leaked: it masked `hello world!` and printed out `U`, since this was the last value of `x` when the loop ended.

However, in Python 3 `x` prints the originally defined `hello world!`, since the local variable from the list

推导式不会屏蔽外部作用域的变量。

此外，无论是生成器表达式（自Python 2.5起可用），还是字典或集合推导式（这些从Python 3回移植到Python 2.7），在Python 2中都不会导致变量泄漏。

请注意，在Python 2和Python 3中，使用for循环时变量会泄漏到外部作用域：

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
    vowels.append(x)
print(x)
# 输出: 'U'
```

## 第149.8节：True、False和None

在Python 2中，True、False和None是内置常量。这意味着它们可以被重新赋值。

```
Python 2.x 版本 ≥ 2.0
True, False = False, True
True # False
False # True
```

从 Python 2.4 开始，不能对 None 进行此操作。

```
Python 2.x 版本 ≥ 2.4
None = None # 语法错误：不能给 None 赋值
```

在 Python 3 中，True、False 和 None 现在是关键字。

```
Python 3.x 版本 ≥ 3.0
True, False = False, True # 语法错误：不能给关键字赋值
None = None # 语法错误：不能给关键字赋值
```

## 第149.9节：用户输入

在 Python 2 中，使用 `raw_input` 函数接受用户输入，

```
Python 2.x 版本 ≥ 2.3
user_input = raw_input()
```

而在 Python 3 中，用户输入是通过 `input` 函数接受的。

```
Python 3.x 版本 ≥ 3.0
user_input = input()
```

在 Python 2 中，`input` 函数会接受输入并对其进行解释。虽然这很有用，但存在一些安全考虑，因此在 Python 3 中被移除。要实现相同功能，可以使用 `eval(input())`。

为了使脚本在两个版本中都能移植，可以将以下代码放在 Python 脚本的顶部附近：

```
尝试:
    input = raw_input
except NameError:
```

comprehension does not mask variables from the surrounding scope.

Additionally, neither generator expressions (available in Python since 2.5) nor dictionary or set comprehensions (which were backported to Python 2.7 from Python 3) leak variables in Python 2.

Note that in both Python 2 and Python 3, variables will leak into the surrounding scope when using a for loop:

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
    vowels.append(x)
print(x)
# Out: 'U'
```

## Section 149.8: True, False and None

In Python 2, `True`, `False` and `None` are built-in constants. Which means it's possible to reassign them.

```
Python 2.x 版本 ≥ 2.0
True, False = False, True
True # False
False # True
```

You can't do this with `None` since Python 2.4.

```
Python 2.x 版本 ≥ 2.4
None = None # SyntaxError: cannot assign to None
```

In Python 3, `True`, `False`, and `None` are now keywords.

```
Python 3.x 版本 ≥ 3.0
True, False = False, True # SyntaxError: can't assign to keyword
None = None # SyntaxError: can't assign to keyword
```

## Section 149.9: User Input

In Python 2, user input is accepted using the `raw_input` function,

```
Python 2.x 版本 ≥ 2.3
user_input = raw_input()
```

而在 Python 3 中，用户输入是通过 `input` 函数接受的。

```
Python 3.x 版本 ≥ 3.0
user_input = input()
```

In Python 2, the `input` function will accept input and *interpret* it. While this can be useful, it has several security considerations and was removed in Python 3. To access the same functionality, `eval(input())` can be used.

To keep a script portable across the two versions, you can put the code below near the top of your Python script:

```
try:
    input = raw_input
except NameError:
```

## 第 149.10 节：不同类型的比较

Python 2.x 版本 ≥ 2.3

不同类型的对象可以进行比较。结果是任意的，但一致的。它们的顺序是 `None` 小于任何其他类型，数值类型小于非数值类型，其他所有类型按类型的字典序排列。因此，`int` 小于 `str`, `tuple` 大于 `list` :

```
[1, 2] > 'foo'  
# 输出: False  
(1, 2) > 'foo'  
# 输出: True  
[1, 2] > (1, 2)  
# 输出: False  
100 < [1, 'x'] < 'xyz' < (1, 'x')  
# 输出: True
```

最初这样做是为了能够对混合类型的列表进行排序，并且对象会按类型分组：

```
l = [7, 'x', (1, 2), [5, 6], 5, 8.0, 'y', 1.2, [7, 8], 'z']  
sorted(l)  
# 输出: [1.2, 5, 7, 8.0, [5, 6], [7, 8], 'x', 'y', 'z', (1, 2)]
```

Python 3.x 版本 ≥ 3.0

比较不同（非数字）类型时会引发异常：

```
1 < 1.5  
# 输出: True  
  
[1, 2] > 'foo'  
# TypeError: unorderable types: list() > str()  
(1, 2) > 'foo'  
# TypeError: 不可比较的类型: tuple() > str()  
[1, 2] > (1, 2)  
# TypeError: 不可比较的类型: list() > tuple()
```

要在 Python 3 中按类型对混合列表进行排序，并实现版本间的兼容性，必须为 `sorted` 函数提供一个 `key` :

```
>>> list = [1, 'hello', [3, 4], {'python': 2}, 'stackoverflow', 8, {'python': 3}, [5, 6]]  
>>> sorted(list, key=str)  
# 输出: [1, 8, [3, 4], [5, 6], 'hello', 'stackoverflow', {'python': 2}, {'python': 3}]
```

使用 `str` 作为 `key` 函数会暂时将每个元素转换为字符串，仅用于比较目的。然后它会根据字符串表示的首字符是 `[`、`{` 或 `0-9` 来排序这些元素（以及后续所有字符）。

## 第 149.11 节：迭代器上的 `.next()` 方法重命名

在 Python 2 中，可以通过迭代器自身的 `next` 方法遍历迭代器：

```
Python 2.x 版本 ≥ 2.3  
g = (i for i in range(0, 3))  
g.next() # 返回 0
```

## Section 149.10: Comparison of different types

Python 2.x 版本 ≥ 2.3

Objects of different types can be compared. The results are arbitrary, but consistent. They are ordered such that `None` is less than anything else, numeric types are smaller than non-numeric types, and everything else is ordered lexicographically by type. Thus, an `int` is less than a `str` and a `tuple` is greater than a `list`:

```
[1, 2] > 'foo'  
# Out: False  
(1, 2) > 'foo'  
# Out: True  
[1, 2] > (1, 2)  
# Out: False  
100 < [1, 'x'] < 'xyz' < (1, 'x')  
# Out: True
```

This was originally done so a list of mixed types could be sorted and objects would be grouped together by type:

```
l = [7, 'x', (1, 2), [5, 6], 5, 8.0, 'y', 1.2, [7, 8], 'z']  
sorted(l)  
# Out: [1.2, 5, 7, 8.0, [5, 6], [7, 8], 'x', 'y', 'z', (1, 2)]
```

Python 3.x 版本 ≥ 3.0

An exception is raised when comparing different (non-numeric) types:

```
1 < 1.5  
# Out: True  
  
[1, 2] > 'foo'  
# TypeError: unorderable types: list() > str()  
(1, 2) > 'foo'  
# TypeError: unorderable types: tuple() > str()  
[1, 2] > (1, 2)  
# TypeError: unorderable types: list() > tuple()
```

To sort mixed lists in Python 3 by types and to achieve compatibility between versions, you have to provide a key to the sorted function:

```
>>> list = [1, 'hello', [3, 4], {'python': 2}, 'stackoverflow', 8, {'python': 3}, [5, 6]]  
>>> sorted(list, key=str)  
# Out: [1, 8, [3, 4], [5, 6], 'hello', 'stackoverflow', {'python': 2}, {'python': 3}]
```

Using `str` as the key function temporarily converts each item to a string only for the purposes of comparison. It then sees the string representation starting with either `[`、`{` 或 `0-9` and it's able to sort those (and all the following characters).

## Section 149.11: `.next()` method on iterators renamed

In Python 2, an iterator can be traversed by using a method called `next` on the iterator itself:

```
Python 2.x 版本 ≥ 2.3  
g = (i for i in range(0, 3))  
g.next() # Yields 0
```

```
g.next() # 生成 1  
g.next() # 生成 2
```

在 Python 3 中, `.next` 方法被重命名为 `__next__`, 以体现其“魔法”角色, 因此调用 `.next` 会引发 `AttributeError`。正确的做法是在 Python 2 和 Python 3 中都使用带有迭代器作为参数的 `next` 函数 来访问此功能。

```
Python 3.x 版本 ≥ 3.0  
g = (i for i in range(0, 3))  
next(g) # 生成 0  
next(g) # 生成 1  
next(g) # 生成 2
```

这段代码在 2.6 版本到当前版本之间均可移植使用。

## 第 149.12 节 : filter()、map() 和 zip() 返回迭代器而非序列

```
Python 2.x 版本 ≤ 2.7
```

在 Python 2 中, `filter`、`map` 和 `zip` 内置函数返回的是序列。`map` 和 `zip` 总是返回列表, 而 `filter` 的返回类型取决于传入参数的类型 :

```
>>> s = filter(lambda x: x.isalpha(), 'a1b2c3')  
>>> s  
'abc'  
>>> s = map(lambda x: x * x, [0, 1, 2])  
>>> s  
[0, 1, 4]  
>>> s = zip([0, 1, 2], [3, 4, 5])  
>>> s  
[(0, 3), (1, 4), (2, 5)]
```

```
Python 3.x 版本 ≥ 3.0
```

在 Python 3 中, `filter`、`map` 和 `zip` 返回的是迭代器 :

```
>>> it = filter(lambda x: x.isalpha(), 'a1b2c3')  
>>> it  
<filter object at 0x00000098A55C2518>  
>>> ''.join(it)  
'abc'  
>>> it = map(lambda x: x * x, [0, 1, 2])  
>>> it  
<map object at 0x000000E0763C2D30>  
>>> list(it)  
[0, 1, 4]  
>>> it = zip([0, 1, 2], [3, 4, 5])  
>>> it  
<zip object at 0x000000E0763C52C8>  
>>> list(it)  
[(0, 3), (1, 4), (2, 5)]
```

自 Python 2 起, `itertools` 中的 `izip` 相当于 Python 3 中的 `zip`, Python 3 中已移除 `izip`。

## 第 149.13 节 : 重命名的模块

标准库中的一些模块已被重命名 :

```
g.next() # Yields 1  
g.next() # Yields 2
```

In Python 3 the `.next` method has been renamed to `__next__`, acknowledging its “magic” role, so calling `.next` will raise an `AttributeError`. The correct way to access this functionality in both Python 2 and Python 3 is to call the `next` function with the iterator as an argument.

```
Python 3.x 版本 ≥ 3.0  
g = (i for i in range(0, 3))  
next(g) # Yields 0  
next(g) # Yields 1  
next(g) # Yields 2
```

This code is portable across versions from 2.6 through to current releases.

## Section 149.12: filter(), map() and zip() return iterators instead of sequences

```
Python 2.x 版本 ≤ 2.7
```

In Python 2 `filter`, `map` and `zip` built-in functions return a sequence. `map` and `zip` always return a list while with `filter` the return type depends on the type of given parameter:

```
>>> s = filter(lambda x: x.isalpha(), 'a1b2c3')  
>>> s  
'abc'  
>>> s = map(lambda x: x * x, [0, 1, 2])  
>>> s  
[0, 1, 4]  
>>> s = zip([0, 1, 2], [3, 4, 5])  
>>> s  
[(0, 3), (1, 4), (2, 5)]
```

```
Python 3.x 版本 ≥ 3.0
```

In Python 3 `filter`, `map` and `zip` return iterator instead:

```
>>> it = filter(lambda x: x.isalpha(), 'a1b2c3')  
>>> it  
<filter object at 0x00000098A55C2518>  
>>> ''.join(it)  
'abc'  
>>> it = map(lambda x: x * x, [0, 1, 2])  
>>> it  
<map object at 0x000000E0763C2D30>  
>>> list(it)  
[0, 1, 4]  
>>> it = zip([0, 1, 2], [3, 4, 5])  
>>> it  
<zip object at 0x000000E0763C52C8>  
>>> list(it)  
[(0, 3), (1, 4), (2, 5)]
```

Since Python 2 `itertools.izip` is equivalent of Python 3 `zip` `izip` has been removed on Python 3.

## Section 149.13: Renamed modules

A few modules in the standard library have been renamed:

旧名称	新名称
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
_markupbase	markupbase
repr	reprlib
test.test_support	test.support
Tkinter	tkinter
tkFileDialog	tkinter.filedialog
urllib / urllib2	urllib, urllib.parse, urllib.error, urllib.response, urllib.request, urllib.robotparser

有些模块甚至已经从文件转换为库。以上面的 `tkinter` 和 `urllib` 为例。

## 兼容性

在维护 Python 2.x 和 3.x 版本兼容性时，可以使用[future外部包](#)来实现在 Python 2.x 版本中以 Python 3.x 名称导入顶级标准库包。

## 第149.14节：移除操作符 `<>` 和 ````，它们与 `!=` 和 `repr()` 同义

在 Python 2 中，`<>` 是 `!=` 的同义词；同样，`foo` 是 `repr(foo)` 的同义词。

Python 2.x 版本 ≤ 2.7

```
>>> 1 <> 2
True
>>> 1 <> 1
False
>>> foo = 'hello world'
>>> repr(foo)
"hello world"
>>> `foo`
"hello world"
```

Python 3.x 版本 ≥ 3.0

```
>>> 1 <> 2
文件 "<stdin>", 第 1 行
 1 <> 2
^
语法错误: 无效的语法
>>> `foo`
文件 "<stdin>", 第 1 行
  `foo`
^
语法错误: 无效的语法
```

## 第149.15节：long与int的区别

在 Python 2 中，任何大于 C 语言中 `ssize_t` 类型的整数都会被转换为 `long` 数据类型，字面量后会带有 L 后缀。  
例如，在 32 位的 Python 构建环境中：

Python 2.x 版本 ≤ 2.7

Old name	New name
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
_markupbase	markupbase
repr	reprlib
test.test_support	test.support
Tkinter	tkinter
tkFileDialog	tkinter.filedialog
urllib / urllib2	urllib, urllib.parse, urllib.error, urllib.response, urllib.request, urllib.robotparser

Some modules have even been converted from files to libraries. Take `tkinter` and `urllib` from above as an example.

## Compatibility

When maintaining compatibility between both Python 2.x and 3.x versions, you can use the [future external package](#) to enable importing top-level standard library packages with Python 3.x names on Python 2.x versions.

## Section 149.14: Removed operators `<>` and ````, synonymous with `!=` and `repr()`

In Python 2, `<>` is a synonym for `!=`; likewise, `foo` is a synonym for `repr(foo)`.

Python 2.x Version ≤ 2.7

```
>>> 1 <> 2
True
>>> 1 <> 1
False
>>> foo = 'hello world'
>>> repr(foo)
"hello world"
>>> `foo`
"hello world"
```

Python 3.x Version ≥ 3.0

```
>>> 1 <> 2
File "<stdin>", line 1
  1 <> 2
^
SyntaxError: invalid syntax
>>> `foo`
File "<stdin>", line 1
  `foo`
^
SyntaxError: invalid syntax
```

## Section 149.15: long vs. int

In Python 2, any integer larger than a C `ssize_t` would be converted into the `long` data type, indicated by an L suffix on the literal. For example, on a 32 bit build of Python:

Python 2.x Version ≤ 2.7

```
>>> 2**31
2147483648L
>>> type(2**31)
<type 'long'>
>>> 2**30
1073741824
>>> type(2**30)
<type 'int'>
>>> 2**31 - 1 # 2**31 是 long, long - int 是 long
2147483647L
```

然而，在Python 3中，long数据类型被移除；无论整数多大，它都会是一个int类型。

Python 3.x 版本  $\geq$  3.0

```
2**1024
# 输出:
```

```
>>> 2**31
2147483648L
>>> type(2**31)
<type 'long'>
>>> 2**30
1073741824
>>> type(2**30)
<type 'int'>
>>> 2**31 - 1 # 2**31 is long and long - int is long
2147483647L
```

However, in Python 3, the `long` data type was removed; no matter how big the integer is, it will be an `int`.

Python 3.x 版本  $\geq$  3.0

```
2**1024
# Output:
17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847732240753602
1120113879871393357658789768814416224928474306394741243777678934248654852763022196012460941194530829
52085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224
137216
print(-(2**1024))
# Output:
-17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847732240753602
21120113879871393357658789768814416224928474306394741243777678934248654852763022196012460941194530829
952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224
4137216
type(2**1024)
# Output: <class 'int'>
```

## Section 149.16: All classes are "new-style classes" in Python 3

In Python 3.x all classes are *new-style classes*; when defining a new class python implicitly makes it inherit from `object`. As such, specifying `object` in a `class` definition is a completely optional:

Python 3.x 版本  $\geq$  3.0

```
class X: pass
class Y(object): pass
```

Both of these classes now contain `object` in their `mro` (method resolution order):

Python 3.x 版本  $\geq$  3.0

```
>>> X.__mro__
(__main__.X, object)

>>> Y.__mro__
(__main__.Y, object)
```

In Python 2.x classes are, by default, old-style classes; they do not implicitly inherit from `object`. This causes the semantics of classes to differ depending on if we explicitly add `object` as a base `class`:

Python 2.x 版本  $\geq$  2.3

```
class X: pass
class Y(object): pass
```

In this case, if we try to print the `__mro__` of `Y`, similar output as that in the Python 3.x case will appear:

Python 2.x 版本  $\geq$  2.3

```
>>> Y.__mro__
(<class '__main__.Y', <type 'object'>)
```

这是因为我们在定义时显式让Y继承了object：`class Y(object): pass`。对于不继承object的类X，`__mro__`属性不存在，尝试访问会导致`AttributeError`。

为了确保兼容性两个Python版本，类可以定义为以object作为基类  
class:

```
class mycls(object):
    "我完全兼容 Python 2/3"
```

或者，如果在全局作用域中将`_metaclass_`变量设置为`type`，则在给定模块中所有随后定义的类都会隐式成为新式类，无需显式继承object：

```
_metaclass_ = type

class mycls:
    "我也完全兼容 Python 2/3"
```

## 第149.17节：Reduce 不再是内置函数

在 Python 2 中，reduce既可以作为内置函数使用，也可以从functools包中导入（2.6版本及以上），而在 Python 3 中，reduce只能从functools中导入。不过 Python 2 和 Python 3 中reduce的语法相同，形式为`reduce(function_to_reduce, list_to_reduce)`。

举个例子，假设我们通过除法将列表中的相邻数字归约为单个值。这里我们使用operator库中的truediv函数。

在 Python 2.x 中，写法非常简单：

```
Python 2.x 版本 ≥ 2.3
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

在 Python 3.x 中，示例变得稍微复杂一些：

```
Python 3.x 版本 ≥ 3.0
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

我们也可以使用`from functools import reduce`来避免使用命名空间调用`reduce`。

## 第 149.18 节：绝对/相对导入

在 Python 3 中，PEP 404 改变了导入的方式，不再允许在包中使用隐式相对导入，且`from ... import *`仅允许在模块级代码中使用。

要在 Python 2 中实现 Python 3 的行为：

```
>>> Y.__mro__
(<class '__main__.Y', <type 'object'>)
```

This happens because we explicitly made Y inherit from object when defining it: `class Y(object): pass`. For class X which does *not* inherit from object the `__mro__` attribute does not exist, trying to access it results in an `AttributeError`.

In order to **ensure compatibility** between both versions of Python, classes can be defined with `object` as a base class:

```
class mycls(object):
    """I am fully compatible with Python 2/3"""


```

Alternatively, if the `_metaclass_` variable is set to `type` at global scope, all subsequently defined classes in a given module are implicitly new-style without needing to explicitly inherit from `object`:

```
_metaclass_ = type

class mycls:
    """I am also fully compatible with Python 2/3"""


```

## Section 149.17: Reduce is no longer a built-in

In Python 2, `reduce` is available either as a built-in function or from the `functools` package (version 2.6 onwards), whereas in Python 3 `reduce` is available only from `functools`. However the syntax for `reduce` in both Python2 and Python3 is the same and is `reduce(function_to_reduce, list_to_reduce)`.

As an example, let us consider reducing a list to a single value by dividing each of the adjacent numbers. Here we use `truediv` function from the `operator` library.

In Python 2.x it is as simple as:

```
Python 2.x 版本 ≥ 2.3
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

In Python 3.x the example becomes a bit more complicated:

```
Python 3.x 版本 ≥ 3.0
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

We can also use `from functools import reduce` to avoid calling `reduce` with the namespace name.

## Section 149.18: Absolute/Relative Imports

In Python 3, PEP 404 changes the way imports work from Python 2. *Implicit relative* imports are no longer allowed in packages and `from ... import *` imports are only allowed in module level code.

To achieve Python 3 behavior in Python 2:

- 可以通过 `from __future__ import absolute_import` 启用 绝对导入 功能，显式相对 导入被鼓励
- 鼓励 显式相对 导入

为澄清，在 Python 2 中，模块可以如下导入位于同一目录下的另一个模块的内容：

```
import foo
```

仅从import语句来看，foo的位置是不明确的。这种隐式相对导入因此不被推荐，建议使用显式相对导入，形式如下：

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
from ...package import bar
from ...sys import path
```

点号.允许显式声明模块在目录树中的位置。

## 关于相对导入的更多内容

考虑一个名为shapes的用户自定义包。目录结构如下：

```
shapes
├── __init__.py
│
├── circle.py
│
├── square.py
│
└── triangle.py
```

circle.py、square.py 和 triangle.py 都将 util.py 作为模块导入。它们如何引用同级别的模块？

```
from . import util # 使用 util.PI、util.sq(x) 等
```

或者

```
from .util import * # 使用 PI、sq(x) 等调用函数
```

这里的 . 用于同级相对导入。

现在，考虑 shapes 模块的另一种布局：

```
shapes
├── __init__.py
│
├── circle
│   ├── __init__.py
│   └── circle.py
```

- the [absolute imports](#) feature can be enabled with `from __future__ import absolute_import`
- *explicit relative imports* are encouraged in place of *implicit relative imports*

For clarification, in Python 2, a module can import the contents of another module located in the same directory as follows:

```
import foo
```

Notice the location of foo is ambiguous from the import statement alone. This type of implicit relative import is thus discouraged in favor of [explicit relative imports](#), which look like the following:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
from ...package import bar
from ...sys import path
```

The dot . allows an explicit declaration of the module location within the directory tree.

## More on Relative Imports

Consider some user defined package called shapes. The directory structure is as follows:

```
shapes
├── __init__.py
│
├── circle.py
│
├── square.py
│
└── triangle.py
```

circle.py, square.py and triangle.py all import util.py as a module. How will they refer to a module in the same level?

```
from . import util # use util.PI, util.sq(x), etc
```

OR

```
from .util import * #use PI, sq(x), etc to call functions
```

The . is used for same-level relative imports.

Now, consider an alternate layout of the shapes module:

```
shapes
├── __init__.py
│
├── circle
│   ├── __init__.py
│   └── circle.py
```

```
└── square
    ├── __init__.py
    └── square.py

└── triangle
    ├── __init__.py
    └── triangle.py

└── util.py
```

现在，这三个类将如何引用 util.py？

```
from .. import util # 使用 util.PI, util.sq(x) 等
```

或者

```
from ..util import * # 使用 PI, sq(x) 等调用函数
```

.. 用于父级相对导入。根据父子层级之间的层数，添加更多的 ..。

## 第149.19节：map()

map() 是一个内置函数，适用于对可迭代对象的元素应用函数。在 Python 2 中，map 返回一个列表。在 Python 3 中，map 返回一个 map 对象，即生成器。

```
# Python 2.X
>>> map(str, [1, 2, 3, 4, 5])
['1', '2', '3', '4', '5']
>>> type(_)
>>> <class 'list'>

# Python 3.X
>>> map(str, [1, 2, 3, 4, 5])
<map object at 0x...>
>>> type(_)
<class 'map'>

# 我们需要再次应用 map，因为之前的 map 已经“被消费”了.....
>>> map(str, [1, 2, 3, 4, 5])
>>> list(_)
['1', '2', '3', '4', '5']
```

在 Python 2 中，你可以传入 None 作为恒等函数。但这在 Python 3 中不再适用。

```
Python 2.x 版本 ≥ 2.3
>>> map(None, [0, 1, 2, 3, 0, 4])
[0, 1, 2, 3, 0, 4]
```

```
Python 3.x 版本 ≥ 3.0
>>> list(map(None, [0, 1, 2, 3, 0, 5]))
Traceback (most recent call last):
文件 "<stdin>", 第 1 行, 在 <module>
类型错误: 'NoneType' 对象不可调用
```

此外，在 Python 2 中传入多个可迭代对象作为参数时，map 会用 None  
(类似于 `itertools.zip_longest`) 填充较短的可迭代对象。在 Python 3 中，迭代会在最短的可迭代对象结束后停止。

```
└── square
    ├── __init__.py
    └── square.py

└── triangle
    ├── __init__.py
    └── triangle.py

└── util.py
```

Now, how will these 3 classes refer to util.py?

```
from .. import util # use util.PI, util.sq(x), etc
```

OR

```
from ..util import * # use PI, sq(x), etc to call functions
```

The .. is used for parent-level relative imports. Add more .s with number of levels between the parent and child.

## Section 149.19: map()

map() is a builtin that is useful for applying a function to elements of an iterable. In Python 2, map returns a list. In Python 3, map returns a map object, which is a generator.

```
# Python 2.X
>>> map(str, [1, 2, 3, 4, 5])
['1', '2', '3', '4', '5']
>>> type(_)
>>> <class 'list'>

# Python 3.X
>>> map(str, [1, 2, 3, 4, 5])
<map object at 0x...>
>>> type(_)
<class 'map'>

# We need to apply map again because we "consumed" the previous map....
>>> map(str, [1, 2, 3, 4, 5])
>>> list(_)
['1', '2', '3', '4', '5']
```

In Python 2, you can pass None to serve as an identity function. This no longer works in Python 3.

```
Python 2.x 版本 ≥ 2.3
>>> map(None, [0, 1, 2, 3, 0, 4])
[0, 1, 2, 3, 0, 4]
```

```
Python 3.x 版本 ≥ 3.0
>>> list(map(None, [0, 1, 2, 3, 0, 5]))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

Moreover, when passing more than one iterable as argument in Python 2, map pads the shorter iterables with None  
(similar to `itertools.zip_longest`). In Python 3, iteration stops after the shortest iterable.

在 Python 2 中：

```
Python 2.x 版本 ≥ 2.3
>>> map(None, [1, 2, 3], [1, 2], [1, 2, 3, 4, 5])
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

在 Python 3 中：

```
Python 3.x 版本 ≥ 3.0
>>> list(map(lambda x, y, z: (x, y, z), [1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2)]
```

```
# 若要获得与 Python 2 中相同的填充效果, 请使用 itertools 中的 zip_longest
>>> import itertools
>>> list(itertools.zip_longest([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

注意：与其使用map，不如考虑使用列表推导式，它兼容Python 2和3。将map(str, [1, 2, 3, 4, 5]):

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

## 第149.20节：round()函数的平局处理和返回类型

### round()的平局处理

在Python 2中，使用round()对一个与两个整数等距的数字进行取整时，会返回距离0更远的那个整数。例如：

```
Python 2.x 版本 ≤ 2.7
round(1.5) # 输出: 2.0
round(0.5) # 输出: 1.0
round(-0.5) # 输出: -1.0
round(-1.5) # 输出: -2.0
```

然而在Python 3中，round()会返回偶数整数（也称为银行家舍入）。例如：

```
Python 3.x 版本 ≥ 3.0
round(1.5) # 输出: 2
round(0.5) # 输出: 0
round(-0.5) # 输出: 0
round(-1.5) # 输出: -2
```

round()函数遵循向偶数舍入策略，即将处于半途的数字舍入到最近的偶数整数（例如，round(2.5)现在返回2而不是3.0）。

根据维基百科参考，这也被称为无偏舍入、收敛舍入、统计学舍入、荷兰舍入、高斯舍入或奇偶舍入。

四舍六入五成双舍入是IEEE 754标准的一部分，也是微软.NET中的默认舍入模式。

这种舍入策略倾向于减少总的舍入误差。由于平均来看，向上舍入的数字数量与向下舍入的数字数量相同，舍入误差相互抵消。

In Python 2:

```
Python 2.x Version ≥ 2.3
>>> map(None, [1, 2, 3], [1, 2], [1, 2, 3, 4, 5])
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

In Python 3:

```
Python 3.x Version ≥ 3.0
>>> list(map(lambda x, y, z: (x, y, z), [1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2)]
```

```
# to obtain the same padding as in Python 2 use zip_longest from itertools
>>> import itertools
>>> list(itertools.zip_longest([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

**Note:** instead of `map` consider using list comprehensions, which are Python 2/3 compatible. Replacing `map(str, [1, 2, 3, 4, 5])`:

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

## Section 149.20: The round() function tie-breaking and return type

### round() tie breaking

In Python 2, using `round()` on a number equally close to two integers will return the one furthest from 0. For example:

```
Python 2.x Version ≤ 2.7
round(1.5) # Out: 2.0
round(0.5) # Out: 1.0
round(-0.5) # Out: -1.0
round(-1.5) # Out: -2.0
```

In Python 3 however, `round()` will return the even integer (aka *bankers' rounding*). For example:

```
Python 3.x Version ≥ 3.0
round(1.5) # Out: 2
round(0.5) # Out: 0
round(-0.5) # Out: 0
round(-1.5) # Out: -2
```

The `round()` function follows the [half to even rounding](#) strategy that will round half-way numbers to the nearest even integer (for example, `round(2.5)` now returns 2 rather than 3.0).

As per [reference in Wikipedia](#), this is also known as *unbiased rounding*, *convergent rounding*, *statistician's rounding*, *Dutch rounding*, *Gaussian rounding*, or *odd-even rounding*.

Half to even rounding is part of the [IEEE 754](#) standard and it's also the default rounding mode in Microsoft's .NET.

This rounding strategy tends to reduce the total rounding error. Since on average the amount of numbers that are rounded up is the same as the amount of numbers that are rounded down, rounding errors cancel out. Other

其他舍入方法则倾向于在平均误差上存在向上或向下的偏差。

### round() 返回类型

round()函数在Python 2.7中返回float类型

Python 2.x 版本 ≤ 2.7

```
round(4.8)
# 5.0
```

从Python 3.0开始，如果省略第二个参数（小数位数），则返回int类型。

Python 3.x 版本 ≥ 3.0

```
round(4.8)
# 5
```

## 第149.21节：文件输入输出

file 在3.x中不再是内置名称（open 仍然有效）。

文件I/O的内部细节已移至标准库的io模块，该模块也是StringIO的新归属。

```
import io
assert io.open 是 open # 内置函数的别名
buffer = io.StringIO()
buffer.write('hello, ')
# 返回写入的字符数 buffer.write('world!')
buffer.getvalue() # 'hello, world!'
```

文件模式（文本模式与二进制模式）现在决定了读取文件时产生的数据类型（以及写入时所需的数据类型）：

```
with open('data.txt') as f:
    first_line = next(f)
        assert type(first_line) is str
    with open('data.bin', 'rb') as f:
        first_kb = f.read(1024)
            assert type(first_kb) is bytes
```

文本文件的编码默认为 locale.getpreferredencoding(False) 返回的编码。要显式指定编码，请使用 encoding 关键字参数：

```
with open('old_japanese_poetry.txt', 'shift_jis') as text:
    haiku = text.read()
```

## 第149.22节：Python 3中移除了cmp函数

在Python 3中，内置函数 cmp 被移除，同时 \_\_cmp\_\_ 特殊方法也被移除。

摘自文档：

函数 cmp() 应视为已废弃，且不再支持 \_\_cmp\_\_() 特殊方法。

rounding methods instead tend to have an upwards or downwards bias in the average error.

### round() return type

The round() function returns a float type in Python 2.7

Python 2.x Version ≤ 2.7

```
round(4.8)
# 5.0
```

Starting from Python 3.0, if the second argument (number of digits) is omitted, it returns an int.

Python 3.x Version ≥ 3.0

```
round(4.8)
# 5
```

## Section 149.21: File I/O

file is no longer a builtin name in 3.x (open still works).

Internal details of file I/O have been moved to the standard library io module, which is also the new home of StringIO:

```
import io
assert io.open is open # the builtin is an alias
buffer = io.StringIO()
buffer.write('hello, ')
# returns number of characters written
buffer.write('world!\n')
buffer.getvalue() # 'hello, world!\n'
```

The file mode (text vs binary) now determines the type of data produced by reading a file (and type required for writing):

```
with open('data.txt') as f:
    first_line = next(f)
        assert type(first_line) is str
    with open('data.bin', 'rb') as f:
        first_kb = f.read(1024)
            assert type(first_kb) is bytes
```

The encoding for text files defaults to whatever is returned by locale.getpreferredencoding(False). To specify an encoding explicitly, use the encoding keyword parameter:

```
with open('old_japanese_poetry.txt', 'shift_jis') as text:
    haiku = text.read()
```

## Section 149.22: cmp function removed in Python 3

In Python 3 the cmp built-in function was removed, together with the \_\_cmp\_\_ special method.

From the documentation:

The cmp() function should be treated as gone, and the \_\_cmp\_\_() special method is no longer supported.

使用`_lt_()`进行排序，`_eq_()`配合`_hash_()`使用，以及根据需要使用其他丰富的比较方法。（如果你确实需要`cmp()`功能，可以使用表达式`(a > b) - (a < b)`作为`cmp(a, b)`的等价替代。）

此外，所有接受`cmp`参数的内置函数现在只接受`key`关键字参数。

在`functools`模块中还有一个有用的函数`cmp_to_key(func)`，允许你将`cmp`风格的函数转换为`key`风格的函数：

将旧式比较函数转换为键函数。用于接受键函数的工具（例如`sorted()`、`min()`、`max()`、`heapq.nlargest()`、`heapq.nsmallest()`、`itertools.groupby()`）。此函数主要作为从支持比较函数的Python 2转换程序的过渡工具。

Use `_lt_()` for sorting, `_eq_()` with `_hash_()`, and other rich comparisons as needed. (If you really need the `cmp()` functionality, you could use the expression `(a > b) - (a < b)` as the equivalent for `cmp(a, b)`.)

Moreover all built-in functions that accepted the `cmp` parameter now only accept the `key` keyword only parameter.

In the `functools` module there is also useful function `cmp_to_key(func)` that allows you to convert from a `cmp`-style function to a `key`-style function:

Transform an old-style comparison function to a key function. Used with tools that accept key functions (such as `sorted()`, `min()`, `max()`, `heapq.nlargest()`, `heapq.nsmallest()`, `itertools.groupby()`). This function is primarily used as a transition tool for programs being converted from Python 2 which supported the use of comparison functions.

## 第149.23节：八进制常量

在Python 2中，八进制字面量可以定义为

```
>>> 0755 # 仅Python 2
```

为确保跨版本兼容，使用

```
0o755 # 适用于Python 2和Python 3
```

## 第149.24节：写入文件对象时的返回值

在Python 2中，直接写入文件句柄返回`None`：

```
Python 2.x 版本 ≥ 2.3
hi = sys.stdout.write('hello world\n')
# 输出: hello world
type(hi)
# 输出: <type 'NoneType'>
```

在 Python 3 中，向句柄写入文本时会返回写入的字符数，写入字节时会返回写入的字节数：

```
Python 3.x 版本 ≥ 3.0
import sys

char_count = sys.stdout.write('hello world ?\n')
# 输出: hello world ?
char_count
# 输出: 14

byte_count = sys.stdout.buffer.write(b'hello world \ud83d\udcbb\n')
# 输出: hello world ?
byte_count
# 输出: 17
```

## Section 149.23: Octal Constants

In Python 2, an octal literal could be defined as

```
>>> 0755 # only Python 2
```

To ensure cross-compatibility, use

```
0o755 # both Python 2 and Python 3
```

## Section 149.24: Return value when writing to a file object

In Python 2, writing directly to a file handle returns `None`:

```
Python 2.x 版本 ≥ 2.3
hi = sys.stdout.write('hello world\n')
# Out: hello world
type(hi)
# Out: <type 'NoneType'>
```

In Python 3, writing to a handle will return the number of characters written when writing text, and the number of bytes written when writing bytes:

```
Python 3.x 版本 ≥ 3.0
import sys

char_count = sys.stdout.write('hello world ?\n')
# Out: hello world ?
char_count
# Out: 14

byte_count = sys.stdout.buffer.write(b'hello world \xf0\x9f\x90\x8d\n')
# Out: hello world ?
byte_count
# Out: 17
```

## 第149.25节：exec语句在Python 3中是一个函数

在 Python 2 中，`exec` 是一个语句，具有特殊语法：`exec code [in globals[, locals]]`。在 Python 3 中，`exec` 现在

## Section 149.25: exec statement is a function in Python 3

In Python 2, `exec` is a statement, with special syntax: `exec code [in globals[, locals]]` . In Python 3 `exec` is now





```
my_instance = MyClass()
print bool(MyClass)      # True
print bool(my_instance)   # False
```

Python 3.x 版本 ≥ 3.0

在 Python 3 中, `__bool__` 取代了 `__nonzero__`

```
class MyClass:
    def __bool__(self):
        return False
```

```
my_instance = MyClass()
print(bool(MyClass))      # True
print(bool(my_instance))   # False
```

## 第149.29节：Python 2 中 `hasattr` 函数的错误

在 Python 2 中, 当属性引发错误时, `hasattr` 会忽略该属性, 返回 `False`.

```
class A(object):
    @property
    def get(self):
        raise IOError
```

```
class B(object):
    @property
    def get(self):
        return 'get in b'
```

```
a = A()
b = B()
```

```
print 'a hasattr get: ', hasattr(a, 'get')
# 在 Python 2 中输出 False (已修复, Python 3 中为 True)
print 'b hasattr get', hasattr(b, 'get')
# 在 Python 2 和 Python 3 中输出 True
```

此错误已在 Python3 中修复。因此如果你使用 Python 2, 请使用

```
尝试:
    a.get
except AttributeError:
    print("没有 get 属性!")
```

或者使用 `getattr` 代替

```
p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("没有 get 属性!")
```

```
my_instance = MyClass()
print bool(MyClass)      # True
print bool(my_instance)   # False
```

Python 3.x Version ≥ 3.0

In Python 3, `__bool__` is used instead of `__nonzero__`

```
class MyClass:
    def __bool__(self):
        return False
```

```
my_instance = MyClass()
print(bool(MyClass))      # True
print(bool(my_instance))   # False
```

## Section 149.29: `hasattr` function bug in Python 2

In Python 2, when a property raise an error, `hasattr` will ignore this property, returning `False`.

```
class A(object):
    @property
    def get(self):
        raise IOError
```

```
class B(object):
    @property
    def get(self):
        return 'get in b'
```

```
a = A()
b = B()
```

```
print 'a hasattr get: ', hasattr(a, 'get')
# output False in Python 2 (fixed, True in Python 3)
print 'b hasattr get', hasattr(b, 'get')
# output True in Python 2 and Python 3
```

This bug is fixed in Python3. So if you use Python 2, use

```
try:
    a.get
except AttributeError:
    print("no get property!")
```

or use `getattr` instead

```
p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("no get property!")
```

# 视频：Python数据科学与机器学习训练营

学习如何使用NumPy、Pandas、Seaborn、Matplotlib、Plotly、Scikit-Learn、机器学习、Tensorflow等！



- ✓ 使用Python进行数据科学和机器学习
- ✓ 使用Spark进行大数据分析
- ✓ 实现机器学习算法
- ✓ 学习使用NumPy进行数值数据处理
- ✓ 学习使用Pandas进行数据分析
- ✓ 学习使用Matplotlib进行Python绘图
- ✓ 学习使用Seaborn进行统计图表绘制
- ✓ 使用Plotly进行交互式动态可视化
- ✓ 使用SciKit-Learn完成机器学习任务
- ✓ K均值聚类
- ✓ 逻辑回归
- ✓ 线性回归
- ✓ 随机森林和决策树
- ✓ 神经网络
- ✓ 支持向量机今天观看→

# VIDEO: Python for Data Science and Machine Learning Bootcamp

Learn how to use NumPy, Pandas, Seaborn, Matplotlib , Plotly, Scikit-Learn , Machine Learning, Tensorflow, and more!



- ✓ Use Python for Data Science and Machine Learning
- ✓ Use Spark for Big Data Analysis
- ✓ Implement Machine Learning Algorithms
- ✓ Learn to use NumPy for Numerical Data
- ✓ Learn to use Pandas for Data Analysis
- ✓ Learn to use Matplotlib for Python Plotting
- ✓ Learn to use Seaborn for statistical plots
- ✓ Use Plotly for interactive dynamic visualizations
- ✓ Use SciKit-Learn for Machine Learning Tasks
- ✓ K-Means Clustering
- ✓ Logistic Regression
- ✓ Linear Regression
- ✓ Random Forest and Decision Trees
- ✓ Neural Networks
- ✓ Support Vector Machines

**Watch Today →**

# 第150章：2to3工具

参数	描述
文件名 / 目录名	2to3接受一个文件或目录列表作为参数，对其进行转换。目录会递归遍历以查找Python源代码。
选项	<b>选项说明</b>
-f FIX, --fix=FIX	指定要应用的转换；默认：全部。使用--list-fixes列出可用的转换
-j PROCESSES, --processes=PROCESSES	同时运行2to3
-x NOFIX, --nofix=NOFIX	排除某个转换
-l, --list-fixes	列出可用的转换
-p, --print-function	更改语法，使print()被视为函数
-v, --verbose	更详细的输出
--no-diffs	不输出重构的差异
-w	写回修改后的文件
-n, --nobackups	不创建修改文件的备份
-o OUTPUT_DIR, --output-dir=OUTPUT_DIR	将输出文件放置在此目录中，而不是覆盖输入文件。 需要使用-n标志，因为当输入文件未被修改时，不需要备份文件。
-W, --write-unchanged-files	即使没有必要更改，也写出输出文件。与-o一起使用时非常有用，以便完整的源代码树被翻译和复制。隐含-w。
--add-suffix=ADD_SUFFIX	指定要附加到所有输出文件名的字符串。如果非空，则需要使用-n。例如： --add-suffix='3'将生成.py3文件。

## 第150.1节：基本用法

考虑以下Python2.x代码。将文件保存为example.py

```
Python 2.x 版本 ≥ 2.0
def greet(name):
    print "Hello, {0}!".format(name)
print "你叫什么名字？"
name = raw_input()
greet(name)
```

在上述文件中，有几行代码不兼容。Python 3.x中，raw\_input()方法被替换为input()，且print不再是语句，而是函数。可以使用2to3工具将此代码转换为Python 3.x代码。

### Unix

```
$ 2to3 example.py
```

### Windows

```
> path/to/2to3.py example.py
```

运行上述代码将输出与原始源文件的差异，如下所示。

```
RefactoringTool: 跳过隐式修复器：buffer
RefactoringTool: 跳过隐式修复器：idioms
RefactoringTool: 跳过隐式修复器：set_literal
RefactoringTool: 跳过隐式修复器：ws_comma
```

# Chapter 150: 2to3 tool

Parameter	Description
filename / directory_name	2to3 accepts a list of files or directories which is to be transformed as its argument. The directories are recursively traversed for Python sources.
Option	<b>Option Description</b>
-f FIX, --fix=FIX	Specify transformations to be applied; default: all. List available transformations with --list-fixes
-j PROCESSES, --processes=PROCESSES	Run 2to3 concurrently
-x NOFIX, --nofix=NOFIX	Exclude a transformation
-l, --list-fixes	List available transformations
-p, --print-function	Change the grammar so that <b>print()</b> is considered a function
-v, --verbose	More verbose output
--no-diffs	Do not output diffs of the refactoring
-w	Write back modified files
-n, --nobackups	Do not create backups of modified files
-o OUTPUT_DIR, --output-dir=OUTPUT_DIR	Place output files in this directory instead of overwriting input files. Requires the -n flag, as backup files are unnecessary when the input files are not modified.
-W, --write-unchanged-files	Write output files even if no changes were required. Useful with -o so that a complete source tree is translated and copied. Implies -w.
--add-suffix=ADD_SUFFIX	Specify a string to be appended to all output filenames. Requires -n if non-empty. Ex.: --add-suffix='3' will generate .py3 files.

## Section 150.1: Basic Usage

Consider the following Python2.x code. Save the file as example.py

```
Python 2.x Version ≥ 2.0
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

In the above file, there are several incompatible lines. The **raw\_input()** method has been replaced with **input()** in Python 3.x and **print** is no longer a statement, but a function. This code can be converted to Python 3.x code using the 2to3 tool.

### Unix

```
$ 2to3 example.py
```

### Windows

```
> path/to/2to3.py example.py
```

Running the above code will output the differences against the original source file as shown below.

```
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
```

```
RefactoringTool: 已重构 example.py
--- example.py      (原始)
+++ example.py      (重构后)
@@ -1,5 +1,5 @@
 def greet(name):
-    print "Hello, {0}!".format(name)
-print "What's your name?"
-name = raw_input()
+    print("Hello, {0}!".format(name))
+print("What's your name?")
+name = input()
 greet(name)
RefactoringTool: 需要修改的文件：
RefactoringTool: example.py
```

可以使用 `-w` 参数将修改写回源文件。除非使用 `-n` 参数，否则会创建一个名为 `example.py.bak` 的原始文件备份。

#### Unix

```
$ 2to3 -w example.py
```

#### Windows

```
> path/to/2to3.py -w example.py
```

现在 `example.py` 文件已从 Python 2.x 转换为 Python 3.x 代码。

完成后，`example.py` 将包含以下有效的 Python3.x 代码：

Python 3.x 版本 ≥ 3.0

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

```
RefactoringTool: Refactored example.py
--- example.py      (original)
+++ example.py      (refactored)
@@ -1,5 +1,5 @@
 def greet(name):
-    print "Hello, {0}!".format(name)
-print "What's your name?"
-name = raw_input()
+    print("Hello, {0}!".format(name))
+print("What's your name?")
+name = input()
 greet(name)
RefactoringTool: Files that need to be modified:
RefactoringTool: example.py
```

The modifications can be written back to the source file using the `-w` flag. A backup of the original file called `example.py.bak` is created, unless the `-n` flag is given.

#### Unix

```
$ 2to3 -w example.py
```

#### Windows

```
> path/to/2to3.py -w example.py
```

Now the `example.py` file has been converted from Python 2.x to Python 3.x code.

Once finished, `example.py` will contain the following valid Python3.x code:

Python 3.x 版本 ≥ 3.0

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

# 第151章：非官方的Python实现

## 第151.1节：IronPython

基于C#编写的.NET和Mono开源实现，采用Apache许可证2.0授权。它依赖于DLR（动态语言运行时）。目前仅支持2.7版本，3.x版本正在开发中。

与CPython的区别：

- 与.NET框架紧密集成。
- 字符串默认是Unicode编码。
- 不支持用C语言编写的CPython扩展。
- 不存在全局解释器锁（GIL）问题。
- 性能通常较低，但取决于具体测试。

### Hello World

```
print "Hello World!"
```

你也可以使用.NET函数：

```
import clr
from System import Console
Console.WriteLine("Hello World!")
```

### 外部链接

- [官方网站](#)
- [GitHub仓库](#)

## 第 151.2 节：Jython

基于Java编写的JVM开源实现，采用Python软件基金会许可证。它仅支持版本2.7，版本3正在开发中。

与CPython的区别：

- 与JVM紧密集成。
- 字符串为Unicode。
- 不支持用C语言编写的CPython扩展。
- 不存在全局解释器锁（GIL）问题。
- 性能通常较低，但取决于具体测试。

### Hello World

```
print "Hello World!"
```

你也可以使用Java函数：

```
from java.lang import System
System.out.println("Hello World!")
```

### 外部链接

- [官方网站](#)
- [Mercurial仓库](#)

# Chapter 151: Non-official Python implementations

## Section 151.1: IronPython

Open-source implementation for .NET and Mono written in C#, licensed under Apache License 2.0. It relies on DLR (Dynamic Language Runtime). It supports only version 2.7, version 3 is currently being developed.

Differences with CPython:

- Tight integration with .NET Framework.
- Strings are Unicode by default.
- Does not support extensions for CPython written in C.
- Does not suffer from Global Interpreter Lock.
- Performance is usually lower, though it depends on tests.

### Hello World

```
print "Hello World!"
```

You can also use .NET functions:

```
import clr
from System import Console
Console.WriteLine("Hello World!")
```

### External links

- [Official website](#)
- [GitHub repository](#)

## Section 151.2: Jython

Open-source implementation for JVM written in Java, licensed under Python Software Foundation License. It supports only version 2.7, version 3 is currently being developed.

Differences with CPython:

- Tight integration with JVM.
- Strings are Unicode.
- Does not support extensions for CPython written in C.
- Does not suffer from Global Interpreter Lock.
- Performance is usually lower, though it depends on tests.

### Hello World

```
print "Hello World!"
```

You can also use Java functions:

```
from java.lang import System
System.out.println("Hello World!")
```

### External links

- [Official website](#)
- [Mercurial repository](#)

## 第151.3节：Transcrypt

Transcrypt 是一个将相当广泛的 Python 子集预编译成紧凑、可读的 JavaScript 的工具。它具有以下特点：

- 允许使用纯 Python 语法进行经典的面向对象编程和多重继承，由 CPython 的原生解析器解析
- 与高质量的面向网络的 JavaScript 库无缝集成，而非面向桌面的 Python 库
- 基于层级 URL 的模块系统，允许通过 PyPi 分发模块
- Python 源代码与生成的 JavaScript 代码之间的简单对应，便于调试
- 多级源码映射和可选的目标代码源引用注释
- 下载文件体积紧凑，单位为 KB 而非 MB
- 优化的 JavaScript 代码，使用记忆化（调用缓存）来选择性地绕过原型查找链
- 操作符重载可在局部开启或关闭，以便于编写可读的数值计算代码

### 代码大小和速度

经验表明，650 kB 的 Python 源代码大致相当于同等大小的 JavaScript 源代码。其速度与手写的 JavaScript 相当，如果开启调用记忆功能，速度甚至可以超过手写代码。

### 与HTML的集成

```
<script src="__javascript__/hello.js"></script>
<h2>你好演示</h2>

<p>
<div id = "greet">...</div>
<button onclick="hello.solarSystem.greet ()>反复点击我！</button>

<p>
<div id = "explain">...</div>
<button onclick="hello.solarSystem.explain ()>我也请你反复点击！</button>
```

### 与JavaScript和DOM的集成

```
from itertools import chain

class 太阳系:
    planets = [list(chain(planet, (index + 1,))) for index, planet in enumerate((
        ('水星', '炎热', 2240),
        ('金星', '含硫', 6052),
        ('地球', '肥沃', 6378),
        ('火星', '红色', 3397),
        ('木星', '多风暴', 71492),
        ('土星', '有环', 60268),
        ('天王星', '寒冷', 25559),
        ('海王星', '非常寒冷', 24766)
    ))]

    lines = (
        '{} 是一颗 {} 行星',
        '{} 的半径是 {} 公里',
        '{} 是从太阳数起的第 {} 颗行星'
    )

    def __init__(self):
        self.lineIndex = 0
```

## Section 151.3: Transcrypt

Transcrypt is a tool to precompile a fairly extensive subset of Python into compact, readable Javascript. It has the following characteristics:

- Allows for classical OO programming with multiple inheritance using pure Python syntax, parsed by CPython's native parser
- Seamless integration with the universe of high-quality web-oriented JavaScript libraries, rather than the desktop-oriented Python ones
- Hierarchical URL based module system allowing module distribution via PyPi
- Simple relation between Python source and generated JavaScript code for easy debugging
- Multi-level sourcemaps and optional annotation of target code with source references
- Compact downloads, kB's rather than MB's
- Optimized JavaScript code, using memoization (call caching) to optionally bypass the prototype lookup chain
- Operator overloading can be switched on and off locally to facilitate readable numerical math

### Code size and speed

Experience has shown that 650 kB of Python sourcecode roughly translates in the same amount of JavaScript source code. The speed matches the speed of handwritten JavaScript and can surpass it if call memoizing is switched on.

### Integration with HTML

```
<script src="__javascript__/hello.js"></script>
<h2>Hello demo</h2>

<p>
<div id = "greet">...</div>
<button onclick="hello.solarSystem.greet ()>Click me repeatedly!</button>

<p>
<div id = "explain">...</div>
<button onclick="hello.solarSystem.explain ()>And click me repeatedly too!</button>
```

### Integration with JavaScript and DOM

```
from itertools import chain

class SolarSystem:
    planets = [list(chain(planet, (index + 1,))) for index, planet in enumerate((
        ('Mercury', 'hot', 2240),
        ('Venus', 'sulphurous', 6052),
        ('Earth', 'fertile', 6378),
        ('Mars', 'reddish', 3397),
        ('Jupiter', 'stormy', 71492),
        ('Saturn', 'ringed', 60268),
        ('Uranus', 'cold', 25559),
        ('Neptune', 'very cold', 24766)
    ))]

    lines = (
        '{} is a {} planet',
        'The radius of {} is {} km',
        '{} is planet nr. {} counting from the sun'
    )

    def __init__(self):
        self.lineIndex = 0
```

```

def greet (self):
    self.planet = self.planets [int (Math.random () * len (self.planets))]
        document.getElementById ('greet') .innerHTML = '你好 {}'.format (self.planet [0])
    self.explain ()

def explain (self):
    document.getElementById ('explain').innerHTML = (
        self.lines [self.lineIndex] .format (self.planet [0], self.planet [self.lineIndex + 1])
    )
    self.lineIndex = (self.lineIndex + 1) % 3
    solarSystem = SolarSystem ()

```

## 与其他JavaScript库的集成

Transcrypt可以与任何JavaScript库结合使用，无需特殊措施或语法。文档中给出了例如react.js、riot.js、fabric.js和node.js的示例。

## Python和JavaScript代码之间的关系

### Python

```

class A:
    def __init__ (self, x):
        self.x = x

    def show (self, label):
        print ('A.show', label, self.x)

```

```

类 B:
    def __init__ (self, y):
        alert ('In B constructor')
        self.y = y

    def show (self, label):
        print ('B.show', label, self.y)

```

```

class C (A, B):
    def __init__ (self, x, y):
        alert ('In C constructor')
        A.__init__ (self, x)
        B.__init__ (self, y)
        self.show ('constructor')

    def show (self, label):
        B.show (self, label)
        print ('C.show', label, self.x, self.y)

```

```

a = A (1001)
a.show ('america')

```

```

b = B (2002)
b.show ('俄罗斯')

```

```

c = C (3003, 4004)
c.show ('netherlands')

```

```

show2 = c.show
show2 ('copy')

```

### JavaScript

```

def greet (self):
    self.planet = self.planets [int (Math.random () * len (self.planets))]
        document.getElementById ('greet') .innerHTML = 'Hello {}'.format (self.planet [0])
    self.explain ()

def explain (self):
    document.getElementById ('explain').innerHTML = (
        self.lines [self.lineIndex] .format (self.planet [0], self.planet [self.lineIndex + 1])
    )
    self.lineIndex = (self.lineIndex + 1) % 3
    solarSystem = SolarSystem ()

```

## Integration with other JavaScript libraries

Transcrypt can be used in combination with any JavaScript library without special measures or syntax. In the documentation examples are given for a.o. react.js, riot.js, fabric.js and node.js.

## Relation between Python and JavaScript code

### Python

```

class A:
    def __init__ (self, x):
        self.x = x

    def show (self, label):
        print ('A.show', label, self.x)

```

```

class B:
    def __init__ (self, y):
        alert ('In B constructor')
        self.y = y

    def show (self, label):
        print ('B.show', label, self.y)

```

```

class C (A, B):
    def __init__ (self, x, y):
        alert ('In C constructor')
        A.__init__ (self, x)
        B.__init__ (self, y)
        self.show ('constructor')

```

```

def show (self, label):
    B.show (self, label)
    print ('C.show', label, self.x, self.y)

```

```

a = A (1001)
a.show ('america')

```

```

b = B (2002)
b.show ('russia')

```

```

c = C (3003, 4004)
c.show ('netherlands')

```

```

show2 = c.show
show2 ('copy')

```

### JavaScript

```

var A = __class__ ('A', [object], {
    get __init__ () {return __get__ (this, function (self, x) {
        self.x = x;
    })},
    get show () {return __get__ (this, function (self, label) {
        print ('A.show', label, self.x);
    });}
});
var B = __class__ ('B', [object], {
    get __init__ () {return __get__ (this, function (self, y) {
        alert ('In B constructor');
        self.y = y;
    })},
    get show () {return __get__ (this, function (self, label) {
        print ('B.show', label, self.y);
    });}
});
var C = __class__ ('C', [A, B], {
    get __init__ () {return __get__ (this, function (self, x, y) {
        alert ('In C constructor');
        A.__init__ (self, x);
        B.__init__ (self, y);
        self.show ('constructor');
    })},
    get show () {return __get__ (this, function (self, label) {
        B.show (self, label);
        print ('C.show', label, self.x, self.y);
    });}
});
var a = A (1001);
a.show ('america');
var b = B (2002);
b.show ('russia');
var c = C (3003, 4004);
c.show ('netherlands');
var show2 = c.show;
show2 ('copy');

```

## 外部链接

- 官方网站: <http://www.transcript.org/>
- 代码仓库: <https://github.com/JdeH/Transcript>

```

var A = __class__ ('A', [object], {
    get __init__ () {return __get__ (this, function (self, x) {
        self.x = x;
    })},
    get show () {return __get__ (this, function (self, label) {
        print ('A.show', label, self.x);
    });}
});
var B = __class__ ('B', [object], {
    get __init__ () {return __get__ (this, function (self, y) {
        alert ('In B constructor');
        self.y = y;
    })},
    get show () {return __get__ (this, function (self, label) {
        print ('B.show', label, self.y);
    });}
});
var C = __class__ ('C', [A, B], {
    get __init__ () {return __get__ (this, function (self, x, y) {
        alert ('In C constructor');
        A.__init__ (self, x);
        B.__init__ (self, y);
        self.show ('constructor');
    })},
    get show () {return __get__ (this, function (self, label) {
        B.show (self, label);
        print ('C.show', label, self.x, self.y);
    });}
});
var a = A (1001);
a.show ('america');
var b = B (2002);
b.show ('russia');
var c = C (3003, 4004);
c.show ('netherlands');
var show2 = c.show;
show2 ('copy');

```

## External links

- Official website: <http://www.transcript.org/>
- Repository: <https://github.com/JdeH/Transcript>

# 第152章：抽象语法树

## 第152.1节：分析Python脚本中的函数

该程序分析一个Python脚本，并针对每个定义的函数，报告函数开始的行号、函数签名结束的位置、文档字符串结束的位置以及函数定义结束的位置。

```
#!/usr/local/bin/python3

import ast
import sys

""" 我们收集的数据。每个键是函数名；每个值是一个字典
，字典包含键：firstline、sigend、docend 和 lastline，值为对应的行号
表示这些事件发生的位置。 """
functions = {}

def process(functions):
    """ 处理存储在 functions 中的函数数据。 """
    for funcname,data in functions.items():
        print("函数：" ,funcname)
        print("开始于第几行：" ,data['firstline'])
        print("签名结束于第几行：" ,data['sigend'])
        if ( data['sigend'] < data['docend'] ):
            print("文档字符串结束于第几行：" ,data['docend'])
        else:
            print("无文档字符串")
        print("函数结束于第几行：" ,data['lastline'])
        print()

class FuncLister(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        """ 递归访问所有函数，确定每个函数的起始位置、签名结束位置、文档字符串结束位置以及
        函数结束位置。 """
        functions[node.name] = {'firstline':node.lineno}
        sigend = max(node.lineno,lastline(node.args))
        functions[node.name]['sigend'] = sigend
        docstring = ast.get_docstring(node)      docstring
        length = len(docstring.split("\n")) if docstring else -1      functions[node.name]['docend']
        = sigend+docstringlength
        functions[node.name]['lastline'] = lastline(node)
        self.generic_visit(node)

    def lastline(node):
        """ 递归查找节点的最后一行 """
        return max( [ node.lineno if hasattr(node,'lineno') else -1 , ]
        +[lastline(child) for child in ast.iter_child_nodes(node)] )

    def readin(pythonfilename):
        """ 读取文件名并将函数数据存储到 functions 中。 """
        with open(pythonfilename) as f:
            code = f.read()
        FuncLister().visit(ast.parse(code))

    def analyze(file,process):
        """ 读取文件并处理函数数据。 """
        readin(file)
        process(functions)
```

# Chapter 152: Abstract syntax tree

## Section 152.1: Analyze functions in a python script

This analyzes a python script and, for each defined function, reports the line number where the function began, where the signature ends, where the docstring ends, and where the function definition ends.

```
#!/usr/local/bin/python3

import ast
import sys

""" The data we collect. Each key is a function name; each value is a dict
with keys: firstline, sigend, docend, and lastline and values of line numbers
where that happens. """
functions = {}

def process(functions):
    """ Handle the function data stored in functions. """
    for funcname,data in functions.items():
        print("function:" ,funcname)
        print("\tstarts at line:" ,data['firstline'])
        print("\tsignature ends at line:" ,data['sigend'])
        if ( data['sigend'] < data['docend'] ):
            print("\t\tdocstring ends at line:" ,data['docend'])
        else:
            print("\tno docstring")
        print("\tfunction ends at line:" ,data['lastline'])
        print()

class FuncLister(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        """ Recursively visit all functions, determining where each function
        starts, where its signature ends, where the docstring ends, and where
        the function ends. """
        functions[node.name] = {'firstline':node.lineno}
        sigend = max(node.lineno,lastline(node.args))
        functions[node.name]['sigend'] = sigend
        docstring = ast.get_docstring(node)
        docstringlength = len(docstring.split('\n')) if docstring else -1
        functions[node.name]['docend'] = sigend+docstringlength
        functions[node.name]['lastline'] = lastline(node)
        self.generic_visit(node)

    def lastline(node):
        """ Recursively find the last line of a node """
        return max( [ node.lineno if hasattr(node,'lineno') else -1 , ]
        +[lastline(child) for child in ast.iter_child_nodes(node)] )

    def readin(pythonfilename):
        """ Read the file name and store the function data into functions. """
        with open(pythonfilename) as f:
            code = f.read()
        FuncLister().visit(ast.parse(code))

    def analyze(file,process):
        """ Read the file and process the function data. """
        readin(file)
        process(functions)
```

```
if __name__ == '__main__':
    if len(sys.argv)>1:
        for file in sys.argv[1:]:
            analyze(file,process)
    else:
        analyze(sys.argv[0],process)
```

```
if __name__ == '__main__':
    if len(sys.argv)>1:
        for file in sys.argv[1:]:
            analyze(file,process)
    else:
        analyze(sys.argv[0],process)
```

# 第153章：Unicode与字节

参数	详细信息
编码	要使用的编码，例如'ascii'、'utf8'等...
错误	错误模式，例如'replace'用问号替换错误字符，'ignore'忽略错误字符等...

## 第153.1节：编码/解码错误处理

.encode和.decode都支持错误模式。

默认是'strict'，会在错误时抛出异常。其他模式则更宽容。

### 编码

```
>>> "£13.55".encode('ascii', errors='replace')
b'?13.55'
>>> "£13.55".encode('ascii', errors='ignore')
b'13.55'
>>> "£13.55".encode('ascii', errors='namereplace')
b'\u00a313.55'
>>> "£13.55".encode('ascii', errors='xmlcharrefreplace')
b'&#163;13.55'
>>> "£13.55".encode('ascii', errors='backslashreplace')
b'\xa313.55'
```

### 解码

```
>>> b = "£13.55".encode('utf8')
>>> b.decode('ascii', errors='replace')
'◆◆13.55'
>>> b.decode('ascii', errors='ignore')
'13.55'
>>> b.decode('ascii', errors='backslashreplace')
'\xc2\x313.55'
```

### 士气

从以上内容可以清楚看出，在处理unicode和字节时，保持编码的一致性至关重要。

## 第153.2节：文件输入输出

以非二进制模式打开的文件（例如'r'或'w'）处理的是字符串。默认编码是'utf8'。

```
open(fn, mode='r')           # 以utf8模式打开文件进行读取
open(fn, mode='r', encoding='utf16') # 以utf16模式打开文件进行读取

# 错误：当期望字符串时，不能写入字节：
open("foo.txt", "w").write(b"foo")
```

以二进制模式打开的文件（例如'rb'或'wb'）处理的是字节。不能指定encoding参数，因为没有编码。

```
open(fn, mode='wb') # 以写入字节模式打开文件

# 错误：当期望字节时，不能写入字符串：
open(fn, mode='wb').write("hi")
```

# Chapter 153: Unicode and bytes

Parameter	Details
encoding	The encoding to use, e.g. 'ascii', 'utf8', etc...
errors	The errors mode, e.g. 'replace' to replace bad characters with question marks, 'ignore' to ignore bad characters, etc...

## Section 153.1: Encoding/decoding error handling

.encode and .decode both have error modes.

The default is 'strict'，which raises exceptions on error. Other modes are more forgiving.

### Encoding

```
>>> "£13.55".encode('ascii', errors='replace')
b'?13.55'
>>> "£13.55".encode('ascii', errors='ignore')
b'13.55'
>>> "£13.55".encode('ascii', errors='namereplace')
b'\u00a313.55'
>>> "£13.55".encode('ascii', errors='xmlcharrefreplace')
b'&#163;13.55'
>>> "£13.55".encode('ascii', errors='backslashreplace')
b'\xa313.55'
```

### Decoding

```
>>> b = "£13.55".encode('utf8')
>>> b.decode('ascii', errors='replace')
'◆◆13.55'
>>> b.decode('ascii', errors='ignore')
'13.55'
>>> b.decode('ascii', errors='backslashreplace')
'\xc2\x313.55'
```

### Morale

It is clear from the above that it is vital to keep your encodings straight when dealing with unicode and bytes.

## Section 153.2: File I/O

Files opened in a non-binary mode (e.g. 'r' or 'w') deal with strings. The default encoding is 'utf8'.

```
open(fn, mode='r')           # opens file for reading in utf8
open(fn, mode='r', encoding='utf16') # opens file for reading utf16

# ERROR: cannot write bytes when a string is expected:
open("foo.txt", "w").write(b"foo")
```

Files opened in a binary mode (e.g. 'rb' or 'wb') deal with bytes. No encoding argument can be specified as there is no encoding.

```
open(fn, mode='wb') # open file for writing bytes

# ERROR: cannot write string when bytes is expected:
open(fn, mode='wb').write("hi")
```

## 第153.3节：基础知识

在Python 3中，str是支持Unicode的字符串类型，而bytes是原始字节序列的类型。

```
type("f") == type(u"f") # True, <class 'str'>
type(b"f") # <class 'bytes'>
```

在 Python 2 中普通字符串默认是原始字节序列，带有 "u" 前缀的字符串是 unicode 字符串。

```
type("f") == type(b"f") # True, <type 'str'>
type(u"f") # <type 'unicode'>
```

### Unicode 转换为字节

Unicode 字符串可以通过 .encode(encoding) 转换为字节。

#### Python 3

```
>>> "£13.55".encode('utf8')
b'\xc2\xa313.55'
>>> "£13.55".encode('utf16')
b'\xff\xfe\x00\x01\x00\x03\x00.\x005\x005\x00'
```

#### Python 2

在 py2 中默认的控制台编码是 `sys.getdefaultencoding() == 'ascii'`，而不是 py3 中的 `utf-8`，因此无法像前面的例子那样直接打印。

```
>>> print type(u"£13.55".encode('utf8'))
<type 'str'>
>>> print u"£13.55".encode('utf8')
SyntaxError: 非 ASCII 字符 'Ã' 出现在...
# 在文件内设置编码

# -*- coding: utf-8 -*-
>>> 打印 u"£13.55".encode('utf8')
└─ú13.55
```

如果编码无法处理该字符串，将引发 `UnicodeEncodeError`：

```
>>> "£13.55".encode('ascii')
追踪 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <module>
UnicodeEncodeError: 'ascii' 编码器无法编码位置 0 处的字符 '£' : 序号不在
范围(128)内
```

### 字节转为 Unicode

字节可以通过 .decode(encoding) 转换为 Unicode 字符串。

**字节序列只能通过相应的编码转换为 Unicode 字符串！**

```
>>> b'£13.55'.decode('utf8')
```

## Section 153.3: Basics

In Python 3 str is the type for unicode-enabled strings, while bytes is the type for sequences of raw bytes.

```
type("f") == type(u"f") # True, <class 'str'>
type(b"f") # <class 'bytes'>
```

In Python 2 a casual string was a sequence of raw bytes by default and the unicode string was every string with "u" prefix.

```
type("f") == type(b"f") # True, <type 'str'>
type(u"f") # <type 'unicode'>
```

### Unicode to bytes

Unicode strings can be converted to bytes with .encode(encoding).

#### Python 3

```
>>> "£13.55".encode('utf8')
b'\xc2\x0313.55'
>>> "£13.55".encode('utf16')
b'\xff\xfe\x00\x01\x00\x03\x00.\x005\x005\x00'
```

#### Python 2

in py2 the default console encoding is `sys.getdefaultencoding() == 'ascii'` and not `utf-8` as in py3, therefore printing it as in the previous example is not directly possible.

```
>>> print type(u"£13.55".encode('utf8'))
<type 'str'>
>>> print u"£13.55".encode('utf8')
SyntaxError: Non-ASCII character '\xc2' in...
# with encoding set inside a file

# -*- coding: utf-8 -*-
>>> print u"£13.55".encode('utf8')
└─ú13.55
```

If the encoding can't handle the string, a `UnicodeEncodeError` is raised:

```
>>> "£13.55".encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xa3' in position 0: ordinal not in
range(128)
```

### Bytes to unicode

Bytes can be converted to unicode strings with .decode(encoding).

**A sequence of bytes can only be converted into a unicode string via the appropriate encoding!**

```
>>> b'\xc2\x0313.55'.decode('utf8')
```

'£13.55'

如果编码无法处理该字符串，将引发 `UnicodeDecodeError`：

```
>>> b'\xc2\xa313.55'.decode('utf16')
Traceback (most recent call last):
文件 "<stdin>", 第 1 行, 在 <module>
文件 "/Users/csaftoiu/csaftoiu-github/yahoo-groups-
backup/.virtualenv/bin/../lib/python3.5/encodings/utf_16.py", 第 16 行, 在 decode
    返回 codecs.utf_16_decode(input, errors, True)
UnicodeDecodeError: 'utf-16-le' 编解码器无法解码位置6的字节0x35：数据截断
```

'£13.55'

If the encoding can't handle the string, a `UnicodeDecodeError` is raised:

```
>>> b'\xc2\xa313.55'.decode('utf16')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/csaftoiu/csaftoiu-github/yahoo-groups-
backup/.virtualenv/bin/../lib/python3.5/encodings/utf_16.py", line 16, in decode
    return codecs.utf_16_decode(input, errors, True)
UnicodeDecodeError: 'utf-16-le' codec can't decode byte 0x35 in position 6: truncated data
```

# 第154章：Python串口通信 (pyserial)

参数	详细信息
端口	设备名称，例如GNU/Linux上的/dev/ttyUSB0或Windows上的COM3。
波特率	类型：整数 默认值：9600 标准值：50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200

## 第154.1节：初始化串口设备

```
import serial
#串口接收这两个参数：串口设备和波特率
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

## 第154.2节：从串口读取数据

初始化串口设备

```
import serial
#串口接收两个参数：串口设备和波特率
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

从串口设备读取单个字节

```
data = ser.read()
```

从串口设备读取指定数量的字节

```
data = ser.read(size=5)
```

从串口设备读取一行数据。

```
data = ser.readline()
```

在串口设备上有数据写入时读取数据。

```
#适用于python2.7
data = ser.read(ser.inWaiting())

#适用于python3
ser.read(ser.inWaiting)
```

## 第154.3节：检查你的

机器上有哪些串口可用

要获取可用串口列表，请使用

```
python -m serial.tools.list_ports
```

在命令提示符下或

```
from serial.tools import list_ports
```

# Chapter 154: Python Serial Communication (pyserial)

parameter	details
port	Device name e.g. /dev/ttyUSB0 on GNU/Linux or COM3 on Windows.
baudrate	baudrate type: int default: 9600 standard values: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200

## Section 154.1: Initialize serial device

```
import serial
#Serial takes these two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

## Section 154.2: Read from serial port

Initialize serial device

```
import serial
#Serial takes two parameters: serial device and baudrate
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

to read single byte from serial device

```
data = ser.read()
```

to read given number of bytes from the serial device

```
data = ser.read(size=5)
```

to read one line from serial device.

```
data = ser.readline()
```

to read the data from serial device while something is being written over it.

```
#for python2.7
data = ser.read(ser.inWaiting())

#for python3
ser.read(ser.inWaiting)
```

## Section 154.3: Check what serial ports are available on your machine

To get a list of available serial ports use

```
python -m serial.tools.list_ports
```

at a command prompt or

```
from serial.tools import list_ports
```

```
list_ports.comports() # 输出可用串口列表
```

from the Python shell.

```
list_ports.comports() # Outputs list of available serial ports
```

from the Python shell.

# 视频：机器学习 A-Z：动手 Python数据科学

学习如何从两位数据科学专家那里用Python创建机器学习算法。包含代码模板。



- ✓ 精通Python上的机器学习
- ✓ 对多种机器学习模型有良好的直觉
- ✓ 做出准确的预测
- ✓ 进行强有力的分析
- ✓ 制作稳健的机器学习模型
- ✓ 为您的业务创造强大的附加价值
- ✓ 将机器学习用于个人目的
- ✓ 处理强化学习、自然语言处理和深度学习等特定主题
- ✓ 掌握降维等高级技术
- ✓ 了解针对每种问题应选择哪种机器学习模型
- ✓ 构建一支强大的机器学习模型军团，并知道如何组合它们以解决任何问题

今天观看 →

# VIDEO: Machine Learning A-Z: Hands-On Python In Data Science

Learn to create Machine Learning Algorithms in Python from two Data Science experts. Code templates included.



- ✓ Master Machine Learning on Python
- ✓ Have a great intuition of many Machine Learning models
- ✓ Make accurate predictions
- ✓ Make powerful analysis
- ✓ Make robust Machine Learning models
- ✓ Create strong added value to your business
- ✓ Use Machine Learning for personal purpose
- ✓ Handle specific topics like Reinforcement Learning, NLP and Deep Learning
- ✓ Handle advanced techniques like Dimensionality Reduction
- ✓ Know which Machine Learning model to choose for each type of problem
- ✓ Build an army of powerful Machine Learning models and know how to combine them to solve any problem

Watch Today →

# 第155章：使用Py2Neo的Neo4j和Cypher

## 第155.1节：向Neo4j图添加节点

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    article.properties["title"] = results[r]['news_title']
    article.properties["timestamp"] = results[r]['news_timestamp']
    article.push()
    [...]
```

向图中添加节点非常简单，`graph.merge_one` 很重要，因为它可以防止重复项。（如果你运行脚本两次，第二次它会更新标题，而不会为相同的文章创建新节点）`timestamp` 应该是整数，而不是日期字符串，因为neo4j实际上没有日期

数据类型。当你将日期存储为 '05-06-1989' 时，会导致排序问题。

`article.push()` 是实际将操作提交到neo4j的调用。不要忘记这一步。

## 第155.2节：导入和认证

```
from py2neo import authenticate, Graph, Node, Relationship
authenticate("localhost:7474", "neo4j", "<pass>")
graph = Graph()
```

你必须确保你的Neo4j数据库存在于localhost:7474，并且使用了正确的凭据。

`graph`对象是你在其余Python代码中与neo4j实例交互的接口。你不应该将其作为全局变量，而应将其保存在类的`__init__`方法中。

## 第155.3节：向Neo4j图添加关系

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    if 'LOCATION' in results[r].keys():
        for loc in results[r]['LOCATION']:
            loc = graph.merge_one("Location", "name", loc)
            尝试:
    rel = graph.create_unique(Relationship(article, "about_place", loc))
    except Exception, e:
        print e
```

`create_unique` 对避免重复非常重要。但除此之外，它是一个相当直接的操作。关系名称也很重要，因为在高级用例中会用到它。

## 第155.4节：查询1：新闻标题自动完成

```
def get_autocomplete(text):
    query = """
    start n = node(*) where n.name =~ '(?i)%s.*' return n.name,labels(n) limit 10;
    """
```

# Chapter 155: Neo4j and Cypher using Py2Neo

## Section 155.1: Adding Nodes to Neo4j Graph

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    article.properties["title"] = results[r]['news_title']
    article.properties["timestamp"] = results[r]['news_timestamp']
    article.push()
    [...]
```

Adding nodes to the graph is pretty simple, `graph.merge_one` is important as it prevents duplicate items. (If you run the script twice, then the second time it would update the title and not create new nodes for the same articles)

`timestamp` should be an integer and not a date string as neo4j doesn't really have a date datatype. This causes sorting issues when you store date as '05-06-1989'

`article.push()` is an the call that actually commits the operation into neo4j. Don't forget this step.

## Section 155.2: Importing and Authenticating

```
from py2neo import authenticate, Graph, Node, Relationship
authenticate("localhost:7474", "neo4j", "<pass>")
graph = Graph()
```

You have to make sure your Neo4j Database exists at localhost:7474 with the appropriate credentials.

the `graph` object is your interface to the neo4j instance in the rest of your python code. Rather than making this a global variable, you should keep it in a class's `__init__` method.

## Section 155.3: Adding Relationships to Neo4j Graph

```
results = News.objects.todays_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    if 'LOCATION' in results[r].keys():
        for loc in results[r]['LOCATION']:
            loc = graph.merge_one("Location", "name", loc)
            try:
                rel = graph.create_unique(Relationship(article, "about_place", loc))
            except Exception, e:
                print e
```

`create_unique` is important for avoiding duplicates. But otherwise it's a pretty straightforward operation. The relationship name is also important as you would use it in advanced cases.

## Section 155.4: Query 1: Autocomplete on News Titles

```
def get_autocomplete(text):
    query = """
    start n = node(*) where n.name =~ '(?i)%s.*' return n.name,labels(n) limit 10;
    """
```

```

query = query % (text)
obj = []
for res in graph.cypher.execute(query):
    # print res[0],res[1]
obj.append({'name':res[0],'entity_type':res[1]})
return res

```

这是一个示例的 Cypher 查询，用于获取所有属性名 (name) 以参数 text 开头的节点。

## 第155.5节：查询2：按特定日期和地点获取新闻文章

```

def search_news_by_entity(location,timestamp):
query = """
MATCH (n)-[]->(l)
where l.name='%s' and n.timestamp='%s'
RETURN n.news_id limit 10
"""

query = query % (location,timestamp)

news_ids = []
for res in graph.cypher.execute(query):
news_ids.append(str(res[0]))

return news_ids

```

您可以使用此查询查找所有通过关系连接到某个地点(l)的新闻文章(n)

## 第155.6节：Cypher查询示例

统计与特定人物随时间连接的文章数量

```

MATCH (n)-[]->(l)
where l.name='唐纳德·特朗普'
RETURN n.date,count(*) order by n.date

```

搜索与特朗普相同新闻文章连接且总关系节点数至少为5的其他人物/地点

```

MATCH (n:NewsArticle)-[]->(l)
where l.name='唐纳德·特朗普'
MATCH (n:NewsArticle)-[]->(m)
with m, count(n) as num where num>5
return labels(m)[0],(m.name), num order by num desc limit 10

```

```

query = query % (text)
obj = []
for res in graph.cypher.execute(query):
    # print res[0],res[1]
obj.append({'name':res[0],'entity_type':res[1]})
return res

```

This is a sample cypher query to get all nodes with the property name that starts with the argument text.

## Section 155.5: Query 2 : Get News Articles by Location on a particular date

```

def search_news_by_entity(location,timestamp):
query = """
MATCH (n)-[]->(l)
where l.name='%s' and n.timestamp='%s'
RETURN n.news_id limit 10
"""

query = query % (location,timestamp)

news_ids = []
for res in graph.cypher.execute(query):
news_ids.append(str(res[0]))

return news_ids

```

You can use this query to find all news articles (n) connected to a location (1) by a relationship.

## Section 155.6: Cypher Query Samples

Count articles connected to a particular person over time

```

MATCH (n)-[]->(l)
where l.name='Donald Trump'
RETURN n.date,count(*) order by n.date

```

Search for other People / Locations connected to the same news articles as Trump with at least 5 total relationship nodes.

```

MATCH (n:NewsArticle)-[]->(l)
where l.name='Donald Trump'
MATCH (n:NewsArticle)-[]->(m)
with m, count(n) as num where num>5
return labels(m)[0],(m.name), num order by num desc limit 10

```

# 第156章：使用Python的基本curses

## 第156.1节：wrapper()辅助函数

虽然上述基本调用已经足够简单，curses包还提供了wrapper(func, ...)辅助函数。下面的示例包含了上述的等价内容：

```
main(scr, *args):
    # -- 使用屏幕执行操作 --
    scr.border(0)
    scr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    scr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # 在用户按下 'q' 之前保持在此循环中
        ch = scr.getch()
        if ch == ord('q'):
            break

curses.wrapper(main)
```

这里，wrapper 会初始化 curses，创建 stdscr，一个窗口对象，并将 stdscr 以及任何后续参数传递给 func。当 func 返回时，wrapper 会在程序退出前恢复终端。

## 第156.2节：基本调用示例

```
import curses
import traceback

try:
    # -- 初始化 --
    stdscr = curses.initscr() # 初始化 curses 屏幕curses.noecho()
    # 关闭按键自动回显到屏幕curses.cbreak()          # 进入中断模式，按键后无需按回车
    # 即可注册      stdscr.keypad(1)           # 启用特殊键值，如 curses.KEY_LEFT 等

    # -- 使用屏幕执行操作 --
    stdscr.border(0)
    stdscr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    stdscr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # 在用户按下 'q' 之前保持在此循环中
        ch = stdscr.getch()
        if ch == ord('q'):
            break

    # -- 用户代码结束 --

except:
    traceback.print_exc() # 打印错误的追踪日志

finally:
    # --- 退出时清理 ---
    stdscr.keypad(0)
    curses.echo()
    curses.nocbreak()
    curses.endwin()
```

# Chapter 156: Basic Curses with Python

## Section 156.1: The wrapper() helper function

While the basic invocation above is easy enough, the curses package provides the wrapper(func, ...) helper function. The example below contains the equivalent of above:

```
main(scr, *args):
    # -- Perform an action with Screen --
    scr.border(0)
    scr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    scr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # stay in this loop till the user presses 'q'
        ch = scr.getch()
        if ch == ord('q'):

curses.wrapper(main)
```

Here, wrapper will initialize curses, create stdscr, a WindowObject and pass both stdscr, and any further arguments to func. When func returns, wrapper will restore the terminal before the program exits.

## Section 156.2: Basic Invocation Example

```
import curses
import traceback

try:
    # -- Initialize --
    stdscr = curses.initscr() # initialize curses screen
    curses.noecho()          # turn off auto echoing of keypress on to screen
    curses.cbreak()          # enter break mode where pressing Enter key
    # after keystroke is not required for it to register
    stdscr.keypad(1)          # enable special Key values such as curses.KEY_LEFT etc

    # -- Perform an action with Screen --
    stdscr.border(0)
    stdscr.addstr(5, 5, 'Hello from Curses!', curses.A_BOLD)
    stdscr.addstr(6, 5, 'Press q to close this screen', curses.A_NORMAL)

    while True:
        # stay in this loop till the user presses 'q'
        ch = stdscr.getch()
        if ch == ord('q'):
            break

    # -- End of user code --

except:
    traceback.print_exc() # print trace back log of the error

finally:
    # --- Cleanup on exit ---
    stdscr.keypad(0)
    curses.echo()
    curses.nocbreak()
    curses.endwin()
```

# 第157章：Python中的模板

## 第157.1节：使用模板的简单数据输出程序

```
from string import Template

data = dict(item = "candy", price = 8, qty = 2)

# 定义模板
t = Template("Simon bought $qty $item for $price dollar")
print(t.substitute(data))
```

输出：

西蒙买了2块糖，花了8美元

模板支持基于\$的替换，而非基于%的替换。**Substitute**（映射，关键字）执行模板替换，返回一个新字符串。

映射是任何类似字典的对象，其键与模板占位符匹配。在此示例中，price和 qty 是占位符。关键字参数也可以用作占位符。如果两者同时存在，关键字中的占位符优先。

## 第157.2节：更改分隔符

您可以将"\$"分隔符更改为其他任意字符。以下示例：

```
from string import Template

class MyOtherTemplate(Template):
    delimiter = "#"

data = dict(id = 1, name = "Ricardo")
t = MyOtherTemplate("My name is #name and I have the id: #id")
print(t.substitute(data))
```

您可以在此处阅读文档 [here](#)

# Chapter 157: Templates in python

## Section 157.1: Simple data output program using template

```
from string import Template

data = dict(item = "candy", price = 8, qty = 2)

# define the template
t = Template("Simon bought $qty $item for $price dollar")
print(t.substitute(data))
```

Output:

Simon bought 2 candy for 8 dollar

Templates support \$-based substitutions instead of %-based substitution. **Substitute** (mapping, keywords) performs template substitution, returning a new string.

Mapping is any dictionary-like object with keys that match with the template placeholders. In this example, price and qty are placeholders. Keyword arguments can also be used as placeholders. Placeholders from keywords take precedence if both are present.

## Section 157.2: Changing delimiter

You can change the "\$" delimiter to any other. The following example:

```
from string import Template

class MyOtherTemplate(Template):
    delimiter = "#"

data = dict(id = 1, name = "Ricardo")
t = MyOtherTemplate("My name is #name and I have the id: #id")
print(t.substitute(data))
```

You can read de docs [here](#)

# 第158章：枕头

## 第158.1节：读取图像文件

```
from PIL import Image  
  
im = Image.open("Image.bmp")
```

## 第158.2节：将文件转换为JPEG格式

```
from __future__ import print_function  
import os, sys  
from PIL import Image  
  
for infile in sys.argv[1:]:  
    f, e = os.path.splitext(infile)  
    outfile = f + ".jpg"  
    if infile != outfile:  
        尝试:  
            Image.open(infile).save(outfile)  
        except IOError:  
            print("无法转换", infile)
```

# Chapter 158: Pillow

## Section 158.1: Read Image File

```
from PIL import Image  
  
im = Image.open("Image.bmp")
```

## Section 158.2: Convert files to JPEG

```
from __future__ import print_function  
import os, sys  
from PIL import Image  
  
for infile in sys.argv[1:]:  
    f, e = os.path.splitext(infile)  
    outfile = f + ".jpg"  
    if infile != outfile:  
        尝试:  
            Image.open(infile).save(outfile)  
        except IOError:  
            print("cannot convert", infile)
```

# 第159章：pass语句

## 第159.1节：忽略异常

尝试:

```
metadata = metadata['properties']
except KeyError:
    pass
```

## 第159.2节：创建可捕获的新异常

```
class CompileError(Exception):
    pass
```

# Chapter 159: The pass statement

## Section 159.1: Ignore an exception

try:

```
    metadata = metadata[ 'properties' ]
except KeyError:
    pass
```

## Section 159.2: Create a new Exception that can be caught

```
class CompileError(Exception):
    pass
```

# 视频：机器学习、数据科学和使用 Python 的深度学习

完整的动手机器学习教程，涵盖  
数据科学、Tensorflow、人工智能  
和神经网络



- ✓ 使用Tensorflow和Keras构建人工神经网络
- ✓ 使用深度学习对图像、数据和情感进行分类
- ✓ 使用线性回归、多项式回归和多元回归进行预测
- ✓ 使用Matplotlib和Seaborn进行数据可视化
- ✓ 使用 Apache Spark 的 MLLib 实现大规模机器学习
- ✓ 理解强化学习——以及如何构建一个吃豆人机器人
- ✓ 使用 K-Means 聚类、支持向量机 (SVM) 、KNN、决策树、朴素贝叶斯和主成分分析 (PCA) 对数据进行分类
- ✓ 使用训练/测试和 K 折交叉验证来选择和调整模型
- ✓ 使用基于物品和基于用户的协同过滤构建电影推荐系统

今天观看 →

# VIDEO: Machine Learning, Data Science and Deep Learning with Python

Complete hands-on machine learning tutorial with data science, Tensorflow, artificial intelligence, and neural networks



- ✓ Build artificial neural networks with Tensorflow and Keras
- ✓ Classify images, data, and sentiments using deep learning
- ✓ Make predictions using linear regression, polynomial regression, and multivariate regression
- ✓ Data Visualization with Matplotlib and Seaborn
- ✓ Implement machine learning at massive scale with Apache Spark's MLLib
- ✓ Understand reinforcement learning - and how to build a Pac-Man bot
- ✓ Classify data using K-Means clustering, Support Vector Machines (SVM), KNN, Decision Trees, Naive Bayes, and PCA
- ✓ Use train/test and K-Fold cross validation to choose and tune your models
- ✓ Build a movie recommender system using item-based and user-based collaborative filtering

Watch Today →

# 第160章：带有精确帮助输出的CLI子命令

创建子命令的不同方式，如在hg或svn中，具有与命令行界面和帮助输出完全相同的效果，详见备注部分。

命令行参数解析涵盖了更广泛的参数解析主题。

## 第160.1节：原生方式（无库）

```
"""
用法：sub <命令>

命令：

status - 显示状态
list   - 打印列表
"""

import sys

def check():
    print("status")
    return 0

if sys.argv[1:] == ['status']:
    sys.exit(check())
elif sys.argv[1:] == ['list']:
    print("list")
else:
    print(__doc__.strip())
```

无参数输出：

```
用法：sub

子命令：

status - 显示状态
list   - 打印列表
```

优点：

- 无依赖
- 每个人都应该能够阅读该内容
- 完全控制帮助格式

## 第160.2节：argparse（默认帮助格式器）

```
import argparse
import sys

def check():
    print("status")
    return 0
```

# Chapter 160: CLI subcommands with precise help output

Different ways to create subcommands like in hg or svn with the exact command line interface and help output as shown in Remarks section.

Parsing Command Line arguments covers broader topic of arguments parsing.

## Section 160.1: Native way (no libraries)

```
"""
usage: sub <command>

commands:

    status - show status
    list   - print list
"""

import sys

def check():
    print("status")
    return 0

if sys.argv[1:] == ['status']:
    sys.exit(check())
elif sys.argv[1:] == ['list']:
    print("list")
else:
    print(__doc__.strip())
```

Output without arguments:

```
usage: sub

commands:

    status - show status
    list   - print list
```

Pros:

- no deps
- everybody should be able to read that
- complete control over help formatting

## Section 160.2: argparse (default help formatter)

```
import argparse
import sys

def check():
    print("status")
    return 0
```

```

parser = argparse.ArgumentParser(prog="sub", add_help=False)
subparser = parser.add_subparsers(dest="cmd")

subparser.add_parser('status', help='显示状态')
subparser.add_parser('list', help='打印列表')

# 当未提供参数时显示帮助的技巧
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())

```

无参数输出：

用法: sub {status,list} ...

位置参数:

{status,list}
status 显示状态
list 打印列表

优点：

- 附带 Python
- 包含选项解析

## 第160.3节：argparse（自定义帮助格式化器）

```

import argparse
import sys

类 CustomHelpFormatter(argparse.HelpFormatter):
    定义 _format_action(self, action):
        如果 type(action) == argparse._SubParsersAction:
            # 注入新的类变量用于子命令格式化
    subactions = action._get_subactions()
        invocations = [self._format_action_invocation(a) for a in subactions]
        self._subcommand_max_length = max(len(i) for i in invocations)

    如果 type(action) == argparse._SubParsersAction._ChoicesPseudoAction:
        # 格式化子命令帮助行
    subcommand = self._format_action_invocation(action) # 类型: str
        width = self._subcommand_max_length
    help_text = ""
        if action.help:
    help_text = self._expand_help(action)
        return " {:width} - {}".format(subcommand, help_text, width=width)

    elif type(action) == argparse._SubParsersAction:
        # 处理子命令帮助部分
    msg = ""
        for subaction in action._get_subactions():
    msg += self._format_action(subaction)
        return msg

```

```

parser = argparse.ArgumentParser(prog="sub", add_help=False)
subparser = parser.add_subparsers(dest="cmd")

subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')

# hack to show help when no arguments supplied
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())

```

Output without arguments:

```

usage: sub {status,list} ...

positional arguments:
  {status,list}
    status      show status
    list       print list

```

Pros:

- comes with Python
- option parsing is included

## Section 160.3: argparse (custom help formatter)

```

import argparse
import sys

class CustomHelpFormatter(argparse.HelpFormatter):
    def _format_action(self, action):
        if type(action) == argparse._SubParsersAction:
            # inject new class variable for subcommand formatting
        subactions = action._get_subactions()
            invocations = [self._format_action_invocation(a) for a in subactions]
            self._subcommand_max_length = max(len(i) for i in invocations)

        if type(action) == argparse._SubParsersAction._ChoicesPseudoAction:
            # format subcommand help line
        subcommand = self._format_action_invocation(action) # type: str
            width = self._subcommand_max_length
        help_text = ""
            if action.help:
                help_text = self._expand_help(action)
            return " {:width} - {}\n".format(subcommand, help_text, width=width)

        elif type(action) == argparse._SubParsersAction:
            # process subcommand help section
        msg = '\n'
            for subaction in action._get_subactions():
                msg += self._format_action(subaction)
            return msg

```

```

else:
    return super(CustomHelpFormatter, self).format_action(action)

def check():
    print("status")
    return 0

parser = argparse.ArgumentParser(usage="sub <command>", add_help=False,
                                formatter_class=CustomHelpFormatter)

subparser = parser.add_subparsers(dest="cmd")
subparser.add_parser('status', help='显示状态')
subparser.add_parser('list', help='打印列表')

# 自定义帮助信息
parser._positionals.title = "命令"

# 当未提供参数时显示帮助的技巧
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())

```

无参数输出：

```

usage: sub <command>

命令:

status - 显示状态
list   - 打印列表

```

```

else:
    return super(CustomHelpFormatter, self).format_action(action)

def check():
    print("status")
    return 0

parser = argparse.ArgumentParser(usage="sub <command>", add_help=False,
                                formatter_class=CustomHelpFormatter)

subparser = parser.add_subparsers(dest="cmd")
subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')

# custom help message
parser._positionals.title = "commands"

# hack to show help when no arguments supplied
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)

args = parser.parse_args()

if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())

```

Output without arguments:

```

usage: sub <command>

commands:

status - show status
list   - print list

```

# 第161章：数据库访问

## 第161.1节：SQLite

SQLite 是一个轻量级的基于磁盘的数据库。由于它不需要单独的数据库服务器，因此常用于原型设计或小型应用，这些应用通常由单个用户或在某一时间由一个用户使用。

```
import sqlite3

conn = sqlite3.connect("users.db")
c = conn.cursor()

c.execute("CREATE TABLE user (name text, age integer)")

c.execute("INSERT INTO user VALUES ('User A', 42)")
c.execute("INSERT INTO user VALUES ('User B', 43)")

conn.commit()

c.execute("SELECT * FROM user")
print(c.fetchall())

conn.close()
```

上述代码连接到名为users.db的数据库文件，如果该文件不存在则先创建。你可以通过 SQL 语句与数据库进行交互。

此示例的结果应为：

```
[(u'User A', 42), (u'User B', 43)]
```

### SQLite 语法：深入分析

#### 入门指南

1. 使用以下方式导入 sqlite 模块

```
>>> import sqlite3
```

2. 要使用该模块，必须先创建一个表示数据库的 Connection 对象。这里的数据将存储在 example.db 文件中：

```
>>> conn = sqlite3.connect('users.db')
```

或者，你也可以提供特殊名称 :memory: 来创建一个驻留在内存中的临时数据库，方法如下：

```
>>> conn = sqlite3.connect(':memory:')
```

3. 一旦拥有了 Connection 对象，就可以创建一个 Cursor 对象，并调用其 execute() 方法来执行 SQL 命令：

```
c = conn.cursor()
```

# Chapter 161: Database Access

## Section 161.1: SQLite

SQLite is a lightweight, disk-based database. Since it does not require a separate database server, it is often used for prototyping or for small applications that are often used by a single user or by one user at a given time.

```
import sqlite3

conn = sqlite3.connect("users.db")
c = conn.cursor()

c.execute("CREATE TABLE user (name text, age integer)")

c.execute("INSERT INTO user VALUES ('User A', 42)")
c.execute("INSERT INTO user VALUES ('User B', 43)")

conn.commit()

c.execute("SELECT * FROM user")
print(c.fetchall())

conn.close()
```

The code above connects to the database stored in the file named users.db, creating the file first if it doesn't already exist. You can interact with the database via SQL statements.

The result of this example should be:

```
[(u'User A', 42), (u'User B', 43)]
```

### The SQLite Syntax: An in-depth analysis

#### Getting started

1. Import the sqlite module using

```
>>> import sqlite3
```

2. To use the module, you must first create a Connection object that represents the database. Here the data will be stored in the example.db file:

```
>>> conn = sqlite3.connect('users.db')
```

Alternatively, you can also supply the special name :memory: to create a temporary database in RAM, as follows:

```
>>> conn = sqlite3.connect(':memory:')
```

3. Once you have a Connection, you can create a Cursor object and call its execute() method to perform SQL commands:

```
c = conn.cursor()
```

```

# 创建表
c.execute("创建表 stocks
(日期 文本, 交易类型 文本, 股票代码 文本, 数量 实数, 价格 实数)")

# 插入一行数据
c.execute("插入到 stocks 值 ('2006-01-05','买入','RHAT',100,35.14)")

# 保存 (提交) 更改
conn.commit()

# 如果完成操作, 也可以关闭连接。
# 只要确保所有更改已提交, 否则将丢失。
conn.close()

```

## 连接的重要属性和功能

### 1. isolation\_level

这是一个用于获取或设置当前隔离级别的属性。自动提交模式为 None, 或者为以下之一DEFERRED, 立即或独占。

### 2. 光标

游标对象用于执行SQL命令和查询。

### 3. commit()

提交当前事务。

### 4. rollback()

回滚自上次调用commit()以来所做的所有更改

### 5. close()

关闭数据库连接。它不会自动调用commit()。如果在未先调用commit()的情况下调用close()（假设您未处于自动提交模式），则所有所做的更改将会丢失。

### 6. total\_changes

一个属性，记录自数据库打开以来被修改、删除或插入的总行数。

### 7. execute, executemany, 和 executescript

这些函数的行为与游标对象的相同。这是一个快捷方式，因为通过连接对象调用这些函数会导致创建一个中间游标对象，并调用游标对象的相应方法。

```

# Create table
c.execute('''CREATE TABLE stocks
            (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()

```

## Important Attributes and Functions of Connection

### 1. isolation\_level

It is an attribute used to get or set the current isolation level. None for autocommit mode or one of DEFERRED, IMMEDIATE or EXCLUSIVE.

### 2. cursor

The cursor object is used to execute SQL commands and queries.

### 3. commit()

Commits the current transaction.

### 4. rollback()

Rolls back any changes made since the previous call to commit()

### 5. close()

Closes the database connection. It does not call commit() automatically. If close() is called without first calling commit() (assuming you are not in autocommit mode) then all changes made will be lost.

### 6. total\_changes

An attribute that logs the total number of rows modified, deleted or inserted since the database was opened.

### 7. execute, executemany, and executescript

These functions perform the same way as those of the cursor object. This is a shortcut since calling these functions through the connection object results in the creation of an intermediate cursor object and calls the corresponding method of the cursor object

## 8.row\_factory

您可以将此属性更改为一个可调用对象，该对象接受游标和原始行作为元组，并返回真实的结果行。

```
def dict_factory(cursor, row):
    d = {}
    for i, col in enumerate(cursor.description):
        d[col[0]] = row[i]
    return d

conn = sqlite3.connect(":memory:")
conn.row_factory = dict_factory
```

## Cursor 的重要函数

### 1. execute(sql[, parameters])

执行单条 SQL 语句。该 SQL 语句可以是参数化的（即使用占位符代替 SQL 字面量）。sqlite3 模块支持两种占位符：问号 ? (“qmark 风格”) 和命名占位符 :name (“named 风格”）。

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.execute("create table people (name, age)")

who = "Sophia"
age = 37
# 这是 qmark 风格：
cur.execute("insert into people values (?, ?)",
            (who, age))

# 而且这是命名风格：
cur.execute("select * from people where name=:who and age=:age",
            {"who": who, "age": age}) # 键对应SQL中的占位符

print(cur.fetchone())
```

注意：不要使用 %s 将字符串插入SQL命令，因为这可能使程序容易受到 SQL注入攻击（参见SQL注入）。

### 2. executemany(sql, 参数序列)

对序列sql中所有参数序列或映射执行SQL命令。  
sqlite3模块也允许使用生成参数的迭代器代替序列。

```
L = [(1, 'abcd', 'dfj', 300),    # 一个要插入数据库的元组列表
      (2, 'cfgd', 'dyfj', 400),
      (3, 'sdd', 'dfjh', 300.50)]

conn = sqlite3.connect("test1.db")
conn.execute("create table if not exists book (id int, name text, author text, price real)")
conn.executemany("insert into book values (?, ?, ?, ?)", L)
```

## 8. row\_factory

You can change this attribute to a callable that accepts the cursor and the original row as a tuple and will return the real result row.

```
def dict_factory(cursor, row):
    d = {}
    for i, col in enumerate(cursor.description):
        d[col[0]] = row[i]
    return d

conn = sqlite3.connect(":memory:")
conn.row_factory = dict_factory
```

## Important Functions of Cursor

### 1. execute(sql[, parameters])

Executes a *single* SQL statement. The SQL statement may be parametrized (i. e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks ? (“qmark style”) and named placeholders :name (“named style”).

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.execute("create table people (name, age)")

who = "Sophia"
age = 37
# This IS the qmark STYLE:
cur.execute("insert into people values (?, ?)",
            (who, age))

# AND this IS the named STYLE:
cur.execute("select * from people where name=:who and age=:age",
            {"who": who, "age": age}) # the KEYS correspond TO the placeholders IN SQL

print(cur.fetchone())
```

Beware: don't use %s for inserting strings into SQL commands as it can make your program vulnerable to an SQL injection attack (see SQL Injection).

### 2. executemany(sql, seq\_of\_parameters)

Executes an SQL command against all parameter sequences or mappings found in the sequence sql. The sqlite3 module also allows using an iterator yielding parameters instead of a sequence.

```
L = [(1, 'abcd', 'dfj', 300),    # A list OF tuples TO be inserted INTO the DATABASE
      (2, 'cfgd', 'dyfj', 400),
      (3, 'sdd', 'dfjh', 300.50)]

conn = sqlite3.connect("test1.db")
conn.execute("create table if not exists book (id int, name text, author text, price real)")
conn.executemany("insert into book values (?, ?, ?, ?)", L)
```

```
FOR ROW IN conn.execute("select * from book"):
    print(ROW)
```

你也可以将迭代器对象作为参数传递给executemany，函数会遍历迭代器返回的每个值元组。迭代器必须返回一个值元组。

```
import sqlite3

class IterChars:
    def __init__(SELF):
        SELF.count = ord('a')

    def __iter__(SELF):
        返回自身

    def __next__(SELF):          # (Python 2中使用 NEXT(SELF) )
        如果 SELF.count > ord('z'):
            抛出 StopIteration
        SELF.count += 1
        返回 (chr(SELF.count - 1),)

conn = sqlite3.connect("abc.db")
cur = conn.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

ROWS = cur.execute("select c from characters")
对于 ROW 在 ROWS 中:
    打印(ROW[0]),
```

### 3. executescript(sql\_script)

这是一个非标准的便捷方法，用于一次执行多条SQL语句。它首先发出一个COMMIT语句，然后执行作为参数传入的SQL脚本。

sql\_script可以是str或bytes的实例。

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.executescript("""
    创建表 person(
        firstname,
        lastname,
        age
    );
    创建表 book(
        title,
        author,
        published
    );
    向 book(title, author, published) 插入数据
    值为 (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    )""")
```

```
FOR ROW IN conn.execute("select * from book"):
    print(ROW)
```

You can also pass iterator objects as a parameter to executemany, and the function will iterate over the each tuple of values that the iterator returns. The iterator must return a tuple of values.

```
import sqlite3

class IterChars:
    def __init__(SELF):
        SELF.count = ord('a')

    def __iter__(SELF):
        RETURN SELF

    def __next__(SELF):          # (USE NEXT(SELF) FOR Python 2)
        IF SELF.count > ord('z'):
            raise StopIteration
        SELF.count += 1
        RETURN (chr(SELF.count - 1),)

conn = sqlite3.connect("abc.db")
cur = conn.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

ROWS = cur.execute("select c from characters")
FOR ROW IN ROWS:
    print(ROW[0]),
```

### 3. executescript(sql\_script)

This is a nonstandard convenience method for executing multiple SQL statements at once. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter.

sql\_script can be an instance of str or bytes.

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );
    create table book(
        title,
        author,
        published
    );
    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    )""")
```

```
);  
    ");
```

下一组函数用于配合SQL中的SELECT语句使用。执行SELECT语句后，要检索数据，可以将游标视为迭代器，调用游标的fetchone()方法来获取单条匹配行，或者调用fetchall()来获取所有匹配行的列表。

迭代器形式的示例：

```
import sqlite3  
stocks = [('2006-01-05', 'BUY', 'RHAT', 100, 35.14),  
          ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),  
          ('2006-04-06', 'SELL', 'IBM', 500, 53.0),  
          ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]  
conn = sqlite3.connect(":memory:")  
conn.execute("create table stocks (date text, buysell text, symb text, amount int, price real)")  
conn.executemany("insert into stocks values (?, ?, ?, ?, ?)", stocks)  
cur = conn.cursor()
```

对于行在 cur.execute('SELECT \* FROM stocks ORDER BY price'):  
print(ROW)

```
# 输出：  
# ('2006-01-05', '买入', 'RHAT', 100, 35.14)  
# ('2006-03-28', '买入', 'IBM', 1000, 45.0)  
# ('2006-04-06', '卖出', 'IBM', 500, 53.0)  
# ('2006-04-05', '买入', 'MSFT', 1000, 72.0)
```

#### 4. fetchone()

获取查询结果集的下一行，返回单个序列；如果没有更多数据，则返回None。

```
cur.execute('SELECT * FROM stocks ORDER BY price')  
i = cur.fetchone()  
while(i):  
    print(i)  
    i = cur.fetchone()  
  
# 输出：  
# ('2006-01-05', '买入', 'RHAT', 100, 35.14)  
# ('2006-03-28', '买入', 'IBM', 1000, 45.0)  
# ('2006-04-06', '卖出', 'IBM', 500, 53.0)  
# ('2006-04-05', '买入', 'MSFT', 1000, 72.0)
```

#### 5. fetchmany(size=cursor.arraysize)

获取查询结果的下一组行（由 size 指定），返回一个列表。如果省略 size，fetchmany 将返回单行。当没有更多行可用时，返回空列表。

```
cur.execute('SELECT * FROM stocks ORDER BY price')  
print(cur.fetchmany(2))  
  
# 输出：  
# [ ('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0) ]
```

```
);  
    ");
```

The next set of functions are used in conjunction with **SELECT** statements in SQL. To retrieve data after executing a **SELECT** statement, you can either treat the cursor as an iterator, call the cursor's **fetchone()** method to retrieve a single matching row, or call **fetchall()** to get a list of the matching rows.

Example of the iterator form:

```
import sqlite3  
stocks = [('2006-01-05', 'BUY', 'RHAT', 100, 35.14),  
          ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),  
          ('2006-04-06', 'SELL', 'IBM', 500, 53.0),  
          ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]  
conn = sqlite3.connect(":memory:")  
conn.execute("create table stocks (date text, buysell text, symb text, amount int, price real)")  
conn.executemany("insert into stocks values (?, ?, ?, ?, ?)", stocks)  
cur = conn.cursor()  
  
FOR ROW IN cur.execute('SELECT * FROM stocks ORDER BY price'):  
    print(ROW)  
  
# Output:  
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)  
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)  
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)  
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

#### 4. fetchone()

Fetches the next row of a query result set, returning a single sequence, or None when no more data is available.

```
cur.execute('SELECT * FROM stocks ORDER BY price')  
i = cur.fetchone()  
while(i):  
    print(i)  
    i = cur.fetchone()  
  
# Output:  
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)  
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)  
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)  
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

#### 5. fetchmany(size=cursor.arraysize)

Fetches the next set of rows of a query result (specified by size), returning a list. If size is omitted, fetchmany returns a single row. An empty list is returned when no more rows are available.

```
cur.execute('SELECT * FROM stocks ORDER BY price')  
print(cur.fetchmany(2))  
  
# Output:  
# [ ('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0) ]
```

## 6. fetchall()

获取查询结果的所有（剩余）行，返回一个列表。

```
cur.execute('SELECT * FROM stocks ORDER BY price')
print(cur.fetchall())

# 输出:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0), ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
```

## SQLite 和 Python 数据类型

SQLite 原生支持以下类型：NULL、INTEGER、REAL、TEXT、BLOB。

这是从 SQL 到 Python 或反之的数据类型的转换方式。

None	<->	NULL
int	<->	INTEGER/INT
float	<->	REAL/FLOAT
str	<->	TEXT/VARCHAR(n)
bytes	<->	BLOB

## 6. fetchall()

Fetches all (remaining) rows of a query result, returning a list.

```
cur.execute('SELECT * FROM stocks ORDER BY price')
print(cur.fetchall())

# Output:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0), ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
```

## SQLite and Python data types

SQLite natively supports the following types: NULL, INTEGER, REAL, TEXT, BLOB.

This is how the data types are converted when moving from SQL to Python or vice versa.

None	<->	NULL
int	<->	INTEGER/INT
float	<->	REAL/FLOAT
str	<->	TEXT/VARCHAR(n)
bytes	<->	BLOB

## 第 161.2 节：使用 MySQLdb 访问 MySQL 数据库

首先需要使用 `connect` 方法创建与数据库的连接。之后，您将需要一个游标来操作该连接。

使用游标的 `execute` 方法与数据库交互，偶尔使用连接对象的 `commit` 方法提交更改。

完成所有操作后，别忘了关闭游标和连接。

这里有一个包含所有必需内容的 `Dbconnect` 类。

```
import MySQLdb

class Dbconnect(object):

    def __init__(self):

        self.dbconnection = MySQLdb.connect(host='host_example',
                                            port=int('port_example'),
                                            user='user_example',
                                            passwd='pass_example',
                                            db='schema_example')
        selfdbcursor = self.dbconnection.cursor()

    def commit_db(self):
        self.dbconnection.commit()

    def close_db(self):
        selfdbcursor.close()
        self.dbconnection.close()
```

与数据库交互很简单。创建对象后，只需使用 `execute` 方法。

## Section 161.2: Accessing MySQL database using MySQLdb

The first thing you need to do is create a connection to the database using the `connect` method. After that, you will need a cursor that will operate with that connection.

Use the `execute` method of the cursor to interact with the database, and every once in a while, commit the changes using the `commit` method of the connection object.

Once everything is done, don't forget to close the cursor and the connection.

Here is a `Dbconnect` class with everything you'll need.

```
import MySQLdb

class Dbconnect(object):

    def __init__(self):

        self.dbconnection = MySQLdb.connect(host='host_example',
                                            port=int('port_example'),
                                            user='user_example',
                                            passwd='pass_example',
                                            db='schema_example')
        selfdbcursor = self.dbconnection.cursor()

    def commit_db(self):
        self.dbconnection.commit()

    def close_db(self):
        selfdbcursor.close()
        self.dbconnection.close()
```

Interacting with the database is simple. After creating the object, just use the `execute` method.

```
db = Dbconnect()
db.dbcursor.execute('SELECT * FROM %s' % 'table_example')
```

如果想调用存储过程，请使用以下语法。注意参数列表是可选的。

```
db = Dbconnect()
db.callproc('stored_procedure_name', [parameters] )
```

查询完成后，您可以通过多种方式访问结果。cursor对象是一个生成器，可以获取所有结果或进行循环。

```
results = db.dbcursor.fetchall()
for individual_row in results:
    first_field = individual_row[0]
```

如果您想直接使用生成器进行循环：

```
for individual_row in db.dbcursor:
    first_field = individual_row[0]
```

如果您想提交对数据库的更改：

```
db.commit_db()
```

如果您想关闭游标和连接：

```
db.close_db()
```

## 第161.3节：连接

### 创建连接

根据PEP 249，连接数据库应使用connect()构造函数，该函数返回一个Connection对象。该构造函数的参数依赖于数据库。有关相关参数，请参阅数据库特定主题。

```
import MyDBAPI
con = MyDBAPI.connect(*database_dependent_args)
```

该连接对象有四个方法：

#### 1：关闭

```
con.close()
```

立即关闭连接。请注意，如果调用Connection.\_\_del\_\_方法，连接会自动关闭。任何未完成的事务将隐式回滚。

#### 2：提交

```
con.commit()
```

提交任何未完成的事务到数据库。

```
db = Dbconnect()
db.dbcursor.execute('SELECT * FROM %s' % 'table_example')
```

If you want to call a stored procedure, use the following syntax. Note that the parameters list is optional.

```
db = Dbconnect()
db.callproc('stored_procedure_name', [parameters] )
```

After the query is done, you can access the results multiple ways. The cursor object is a generator that can fetch all the results or be looped.

```
results = db.dbcursor.fetchall()
for individual_row in results:
    first_field = individual_row[0]
```

If you want a loop using directly the generator:

```
for individual_row in db.dbcursor:
    first_field = individual_row[0]
```

If you want to commit changes to the database:

```
db.commit_db()
```

If you want to close the cursor and the connection:

```
db.close_db()
```

## Section 161.3: Connection

### Creating a connection

According to PEP 249, the connection to a database should be established using a connect() constructor, which returns a Connection object. The arguments for this constructor are database dependent. Refer to the database specific topics for the relevant arguments.

```
import MyDBAPI
con = MyDBAPI.connect(*database_dependent_args)
```

This connection object has four methods:

#### 1: close

```
con.close()
```

Closes the connection instantly. Note that the connection is automatically closed if the Connection.\_\_del\_\_ method is called. Any pending transactions will implicitly be rolled back.

#### 2: commit

```
con.commit()
```

Commits any pending transaction to the database.

### 3 : 回滚

```
con.rollback()
```

回滚到任何未完成事务的开始。换句话说：这会取消对数据库的任何未提交事务。

### 4 : 游标

```
cur = con.cursor()
```

返回一个Cursor对象。它用于对数据库进行事务操作。

## 第161.4节：使用psycopg2访问PostgreSQL数据库

psycopg2是最流行的PostgreSQL数据库适配器，既轻量又高效。它是当前PostgreSQL适配器的实现。

其主要特点是完全实现了Python DB API 2.0规范和线程安全（多个线程可以共享同一个连接）

### 建立数据库连接并创建表

```
import psycopg2

# 建立数据库连接。
# 用数据库凭据替换参数值。
conn = psycopg2.connect(database="testpython",
                       user="postgres",
                       host="localhost",
                       password="abc123",
                       port="5432")

# 创建游标。游标允许你执行数据库查询。
cur = conn.cursor()

# 创建表。初始化表名、列名和数据类型。
cur.execute("""CREATE TABLE FRUITS (
              id      INT ,
              fruit_name TEXT,
              color    TEXT,
              price    REAL
            )""")
conn.commit()
conn.close()
```

### 向表中插入数据：

```
# 按照上面创建表的方式，向表中插入值。
cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
              VALUES (1, '苹果', '绿色', 1.00)""")

cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
              VALUES (1, '香蕉', '黄色', 0.80)""")
```

### 检索表数据：

### 3: rollback

```
con.rollback()
```

Rolls back to the start of any pending transaction. In other words: this cancels any non-committed transaction to the database.

### 4: cursor

```
cur = con.cursor()
```

Returns a Cursor object. This is used to do transactions on the database.

## Section 161.4: PostgreSQL Database access using psycopg2

**psycopg2** is the most popular PostgreSQL database adapter that is both lightweight and efficient. It is the current implementation of the PostgreSQL adapter.

Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety (several threads can share the same connection)

### Establishing a connection to the database and creating a table

```
import psycopg2

# Establish a connection to the database.
# Replace parameter values with database credentials.
conn = psycopg2.connect(database="testpython",
                       user="postgres",
                       host="localhost",
                       password="abc123",
                       port="5432")

# Create a cursor. The cursor allows you to execute database queries.
cur = conn.cursor()

# Create a table. Initialise the table name, the column names and data type.
cur.execute("""CREATE TABLE FRUITS (
              id      INT ,
              fruit_name TEXT,
              color    TEXT,
              price    REAL
            )""")
conn.commit()
conn.close()
```

### Inserting data into the table:

```
# After creating the table as shown above, insert values into it.
cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
              VALUES (1, 'Apples', 'green', 1.00)""")

cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
              VALUES (1, 'Bananas', 'yellow', 0.80)""")
```

### Retrieving table data:

```

# 设置查询并执行
cur.execute("""SELECT id, fruit_name, color, price
               FROM fruits""")

# 获取数据
rows = cur.fetchall()

# 处理数据
for row in rows:
    print "ID = {}".format(row[0])
    print "水果名称 = {}".format(row[1])
    print("颜色 = {}".format(row[2]))
    print("价格 = {}".format(row[3]))

```

上述代码的输出将是：

```

ID = 1
名称 = 苹果
颜色 = 绿色
价格 = 1.0

```

```

ID = 2
名称 = 香蕉
颜色 = 黄色
价格 = 0.8

```

所以，这样你就掌握了关于psycopg2的一半知识！:)

## 第161.5节：Oracle数据库

**前提条件：**

- cx\_Oracle 包 - 参见 [here](#) 获取所有版本
- Oracle 即时客户端 - 适用于 [Windows x64](#), [Linux x64](#)

**设置：**

- 安装 cx\_Oracle 包，命令如下：

```
sudo rpm -i <YOUR_PACKAGE_FILENAME>
```

- 解压 Oracle instant client 并设置环境变量，命令如下：

```

ORACLE_HOME=<PATH_TO_INSTANTCLIENT>
PATH=$ORACLE_HOME:$PATH
LD_LIBRARY_PATH=<PATH_TO_INSTANTCLIENT>:$LD_LIBRARY_PATH

```

**创建连接：**

```

import cx_Oracle

class OraExec(object):
    _db_connection = None
    _db_cur = None

    def __init__(self):

```

```

# Set up a query and execute it
cur.execute("""SELECT id, fruit_name, color, price
               FROM fruits""")

# Fetch the data
rows = cur.fetchall()

# Do stuff with the data
for row in rows:
    print "ID = {}".format(row[0])
    print "FRUIT NAME = {}".format(row[1])
    print("COLOR = {}".format(row[2]))
    print("PRICE = {}".format(row[3]))

```

The output of the above would be:

```

ID = 1
NAME = Apples
COLOR = green
PRICE = 1.0

```

```

ID = 2
NAME = Bananas
COLOR = yellow
PRICE = 0.8

```

And so, there you go, you now know half of all you need to know about **psycopg2!** :)

## Section 161.5: Oracle database

**Pre-requisites:**

- cx\_Oracle package - See [here](#) for all versions
- Oracle instant client - For [Windows x64](#), [Linux x64](#)

**Setup:**

- Install the cx\_Oracle package as:

```
sudo rpm -i <YOUR_PACKAGE_FILENAME>
```

- Extract the Oracle instant client and set environment variables as:

```

ORACLE_HOME=<PATH_TO_INSTANTCLIENT>
PATH=$ORACLE_HOME:$PATH
LD_LIBRARY_PATH=<PATH_TO_INSTANTCLIENT>:$LD_LIBRARY_PATH

```

**Creating a connection:**

```

import cx_Oracle

class OraExec(object):
    _db_connection = None
    _db_cur = None

    def __init__(self):

```

```
self._db_connection =  
cx_Oracle.connect('<USERNAME>/<PASSWORD>@<HOSTNAME>:<PORT>/<SERVICE_NAME>')  
self._db_cur = self._db_connection.cursor()
```

#### 获取数据库版本：

```
ver = con.version.split(".")  
打印 ver
```

示例输出：['12', '1', '0', '2', '0']

#### 执行查询：SELECT

```
_db_cur.execute("select * from employees order by emp_id")  
遍历 result 在 _db_cur 中：  
    打印 result
```

输出将以Python元组的形式呈现：

```
(10, 'SYSADMIN', 'IT-INFRA', 7)
```

```
(23, 'HR ASSOCIATE', 'HUMAN RESOURCES', 6)
```

#### 执行查询：INSERT

```
_db_cur.execute("insert into employees(emp_id, title, dept, grade)  
    values (31, 'MTS', 'ENGINEERING', 7)  
_db_connection.commit()
```

当您在Oracle数据库中执行插入/更新/删除操作时，变更仅在您的会话内可见，直到发出commit命令。数据一旦提交到数据库，其他用户和会话即可访问这些更新的数据。

#### 执行查询：使用绑定变量的INSERT

##### 参考

绑定变量使您能够使用新值重新执行语句，而无需重新解析语句。绑定变量提高代码的重用性，并能降低SQL注入攻击的风险。

```
rows = [ (1, "First"),  
        (2, "Second"),  
        (3, "Third") ]  
_db_cur.bindarraysize = 3  
_db_cur.setinputsizes(int, 10)  
_db_cur.executemany("insert into mytab(id, data) values (:1, :2)", rows)  
_db_connection.commit()
```

#### 关闭连接：

```
_db_connection.close()
```

close() 方法关闭连接。任何未显式关闭的连接将在脚本结束时自动释放。

```
self._db_connection =  
cx_Oracle.connect(' <USERNAME>/<PASSWORD>@<HOSTNAME>:<PORT>/<SERVICE_NAME> ')  
self._db_cur = self._db_connection.cursor()
```

#### Get database version:

```
ver = con.version.split(".")  
print ver
```

Sample Output: ['12', '1', '0', '2', '0']

#### Execute query: SELECT

```
_db_cur.execute("select * from employees order by emp_id")  
for result in _db_cur:  
    print result
```

Output will be in Python tuples:

```
(10, 'SYSADMIN', 'IT-INFRA', 7)
```

```
(23, 'HR ASSOCIATE', 'HUMAN RESOURCES', 6)
```

#### Execute query: INSERT

```
_db_cur.execute("insert into employees(emp_id, title, dept, grade)  
    values (31, 'MTS', 'ENGINEERING', 7)  
_db_connection.commit()
```

When you perform insert/update/delete operations in an Oracle Database, the changes are only available within your session until commit is issued. When the updated data is committed to the database, it is then available to other users and sessions.

#### Execute query: INSERT using Bind variables

##### Reference

Bind variables enable you to re-execute statements with new values, without the overhead of re-parsing the statement. Bind variables improve code re-usability, and can reduce the risk of SQL Injection attacks.

```
rows = [ (1, "First"),  
        (2, "Second"),  
        (3, "Third") ]  
_db_cur.bindarraysize = 3  
_db_cur.setinputsizes(int, 10)  
_db_cur.executemany("insert into mytab(id, data) values (:1, :2)", rows)  
_db_connection.commit()
```

#### Close connection:

```
_db_connection.close()
```

The close() method closes the connection. Any connections not explicitly closed will be automatically released when the script ends.

## 第161.6节：使用 sqlalchemy

使用 sqlalchemy 进行数据库操作：

```
from sqlalchemy import create_engine
from sqlalchemy.engine.url import URL

url = URL(drivername='mysql',
           username='user',
           password='passwd',
           host='host',
           database='db')

engine = create_engine(url) # sqlalchemy 引擎
```

现在这个引擎可以使用了：例如，可以用 pandas 直接从 mysql 获取数据框

```
import pandas as pd

con = engine.connect()
dataframe = pd.read_sql(sql=query, con=con)
```

## Section 161.6: Using sqlalchemy

To use sqlalchemy for database:

```
from sqlalchemy import create_engine
from sqlalchemy.engine.url import URL

url = URL(drivername='mysql',
           username='user',
           password='passwd',
           host='host',
           database='db')

engine = create_engine(url) # sqlalchemy engine
```

Now this engine can be used: e.g. with pandas to fetch dataframes directly from mysql

```
import pandas as pd

con = engine.connect()
dataframe = pd.read_sql(sql=query, con=con)
```

# 第162章：将 Python 连接到 SQL 服务器

## 第162.1节：连接服务器，创建表，查询数据

安装该包：

```
$ pip install pymssql
```

```
import pymssql
```

```
SERVER = "servername"
USER = "username"
PASSWORD = "password"
DATABASE = "dbname"
```

```
connection = pymssql.connect(server=SERVER, user=USER,
                             password=PASSWORD, database=DATABASE)
```

```
cursor = connection.cursor() # 要以字典形式访问字段, 请使用 cursor(as_dict=True)
cursor.execute("SELECT TOP 1 * FROM TableName")
row = cursor.fetchone()
```

```
##### 创建表 #####
cursor.execute("")
```

```
CREATE TABLE posts (
    post_id INT PRIMARY KEY NOT NULL,
    message TEXT,
    publish_date DATETIME
)
")
```

```
##### 在表中插入数据 #####
cursor.execute("")
```

```
INSERT INTO posts VALUES(1, "Hey There", "11.23.2016")
")
```

```
# 提交你的工作到数据库
```

```
connection.commit()
```

```
##### 遍历结果 #####
cursor.execute("SELECT TOP 10 * FROM posts ORDER BY publish_date DESC")
```

```
对于 cursor 中的 row:
```

```
print("消息: " + row[1] + " | " + "日期: " + row[2])
# 如果你传入 as_dict=True 给 cursor
# print(row["message"])

connection.close()
```

如果你的工作与 SQL 表达式相关，你可以做任何事情，只需将这些表达式传递给 execute 方法 (CRUD 操作)。

有关 with 语句、调用存储过程、错误处理或更多示例，请查看：[pymssql.org](http://pymssql.org)

# Chapter 162: Connecting Python to SQL Server

## Section 162.1: Connect to Server, Create Table, Query Data

Install the package:

```
$ pip install pymssql
```

```
import pymssql
```

```
SERVER = "servername"
USER = "username"
PASSWORD = "password"
DATABASE = "dbname"
```

```
connection = pymssql.connect(server=SERVER, user=USER,
                             password=PASSWORD, database=DATABASE)
```

```
cursor = connection.cursor() # to access field as dictionary use cursor(as_dict=True)
cursor.execute("SELECT TOP 1 * FROM TableName")
row = cursor.fetchone()
```

```
##### CREATE TABLE #####
cursor.execute("")
```

```
CREATE TABLE posts (
    post_id INT PRIMARY KEY NOT NULL,
    message TEXT,
    publish_date DATETIME
)
")
```

```
##### INSERT DATA IN TABLE #####
cursor.execute("")
```

```
INSERT INTO posts VALUES(1, "Hey There", "11.23.2016")
")
```

```
# commit your work to database
```

```
connection.commit()
```

```
##### ITERATE THROUGH RESULTS #####
cursor.execute("SELECT TOP 10 * FROM posts ORDER BY publish_date DESC")
```

```
for row in cursor:
    print("Message: " + row[1] + " | " + "Date: " + row[2])
    # if you pass as_dict=True to cursor
    # print(row["message"])

connection.close()
```

You can do anything if your work is related with SQL expressions, just pass this expressions to the execute method(CRUD operations).

For with statement, calling stored procedure, error handling or more example check: [pymssql.org](http://pymssql.org)

# 第163章：PostgreSQL

## 第163.1节：入门

PostgreSQL是一个活跃开发且成熟的开源数据库。使用psycopg2模块，我们可以在数据库上执行查询。

### 使用pip安装

```
pip install psycopg2
```

### 基本用法

假设我们在数据库my\_database中有一个表my\_table，定义如下。

#### **idfirst\_name last\_name**

1 约翰	多伊
------	----

我们可以使用psycopg2模块以下方式在数据库上运行查询。

```
import psycopg2

# 使用用户'my_user'和密码'my_password'建立到现有数据库'my_database'的连接
con = psycopg2.connect("host=localhost dbname=my_database user=my_user password=my_password")

# 创建一个游标
cur = con.cursor()

# 向'my_table'插入一条记录
cur.execute("INSERT INTO my_table(id, first_name, last_name) VALUES (2, 'Jane', 'Doe');")

# 提交当前事务
con.commit()

# 从'my_table'检索所有记录
cur.execute("SELECT * FROM my_table;")
results = cur.fetchall()

# 关闭数据库连接
con.close()

# 打印结果
print(results)

# 输出: [(1, 'John', 'Doe'), (2, 'Jane', 'Doe')]
```

# Chapter 163: PostgreSQL

## Section 163.1: Getting Started

PostgreSQL is an actively developed and mature open source database. Using the psycopg2 module, we can execute queries on the database.

### Installation using pip

```
pip install psycopg2
```

### Basic usage

Let's assume we have a table `my_table` in the database `my_database` defined as follows.

#### **id first\_name last\_name**

1 John	Doe
--------	-----

We can use the psycopg2 module to run queries on the database in the following fashion.

```
import psycopg2

# Establish a connection to the existing database 'my_database' using
# the user 'my_user' with password 'my_password'
con = psycopg2.connect("host=localhost dbname=my_database user=my_user password=my_password")

# Create a cursor
cur = con.cursor()

# Insert a record into 'my_table'
cur.execute("INSERT INTO my_table(id, first_name, last_name) VALUES (2, 'Jane', 'Doe');")

# Commit the current transaction
con.commit()

# Retrieve all records from 'my_table'
cur.execute("SELECT * FROM my_table;")
results = cur.fetchall()

# Close the database connection
con.close()

# Print the results
print(results)

# OUTPUT: [(1, 'John', 'Doe'), (2, 'Jane', 'Doe')]
```

# 第164章：Python与Excel

## 第164.1节：使用xlrd模块读取Excel数据

Python xlrd库用于从Microsoft Excel (tm) 电子表格文件中提取数据。

安装：

```
pip install xlrd
```

或者你可以使用来自 pypi 的 setup.py 文件

<https://pypi.python.org/pypi/xlrd>

读取 Excel 表格：导入 xlrd 模块并使用 open\_workbook() 方法打开 Excel 文件。

```
import xlrd  
book=xlrd.open_workbook('sample.xlsx')
```

检查 Excel 中的表格数量

```
print book.nsheets
```

打印表格名称

```
print book.sheet_names()
```

根据索引获取表格

```
sheet=book.sheet_by_index(1)
```

读取单元格内容

```
cell = sheet.cell(row,col) #其中 row=行号, col=列号  
print cell.value #打印单元格内容
```

获取 Excel 表格中的行数和列数

```
num_rows=sheet.nrows  
num_col=sheet.ncols
```

通过名称获取 Excel 表格

```
sheets = book.sheet_names()  
cur_sheet = book.sheet_by_name(sheets[0])
```

## 第 164.2 节：使用 xlsxwriter 格式化 Excel 文件

```
import xlsxwriter  
  
# 创建一个新文件  
workbook = xlsxwriter.Workbook('your_file.xlsx')  
  
# 为工作簿添加一些新的格式
```

# Chapter 164: Python and Excel

## Section 164.1: Read the excel data using xlrd module

Python xlrd library is to extract data from Microsoft Excel (tm) spreadsheet files.

Installation:

```
pip install xlrd
```

Or you can use setup.py file from pypi

<https://pypi.python.org/pypi/xlrd>

**Reading an excel sheet:** Import xlrd module and open excel file using open\_workbook() method.

```
import xlrd  
book=xlrd.open_workbook('sample.xlsx')
```

Check number of sheets in the excel

```
print book.nsheets
```

Print the sheet names

```
print book.sheet_names()
```

Get the sheet based on index

```
sheet=book.sheet_by_index(1)
```

Read the contents of a cell

```
cell = sheet.cell(row,col) #where row=row number and col=column number  
print cell.value #to print the cell contents
```

Get number of rows and number of columns in an excel sheet

```
num_rows=sheet.nrows  
num_col=sheet.ncols
```

Get excel sheet by name

```
sheets = book.sheet_names()  
cur_sheet = book.sheet_by_name(sheets[0])
```

## Section 164.2: Format Excel files with xlsxwriter

```
import xlsxwriter  
  
# create a new file  
workbook = xlsxwriter.Workbook('your_file.xlsx')  
  
# add some new formats to be used by the workbook
```

```

percent_format = workbook.add_format({'num_format': '0%'})
percent_with_decimal = workbook.add_format({'num_format': '0.0%'})
bold = workbook.add_format({'bold': True})
red_font = workbook.add_format({'font_color': 'red'})
remove_format = workbook.add_format()

# 添加一个新工作表
worksheet = workbook.add_worksheet()

# 设置A列的宽度
worksheet.set_column('A:A', 30, )

# 设置B列宽度为20，并应用之前创建的百分比格式
worksheet.set_column('B:B', 20, percent_format)

# 移除第一行的格式 (高度不变=None)
worksheet.set_row('0:0', None, remove_format)

workbook.close()

```

## 第164.3节：将列表数据写入Excel文件

```

import os, sys
from openpyxl import Workbook
from datetime import datetime

dt = datetime.now()
list_values = [[ "01/01/2016", "05:00:00", 3], \
               [ "01/02/2016", "06:00:00", 4], \
               [ "01/03/2016", "07:00:00", 5], \
               [ "01/04/2016", "08:00:00", 6], \
               [ "01/05/2016", "09:00:00", 7]]

# 在Excel中创建工作簿：
wb = Workbook()
sheet = wb.active
sheet.title = 'data'

# 将标题写入Excel工作簿：
row = 1
sheet['A'+str(row)] = '日期'
sheet['B'+str(row)] = '时间'
sheet['C'+str(row)] = '数值'

# 填充数据
for item in list_values:
    行 += 1
    sheet['A'+str(row)] = item[0]
    sheet['B'+str(row)] = item[1]
    sheet['C'+str(row)] = item[2]

# 按日期保存文件：
filename = 'data_' + dt.strftime("%Y%m%d_%I%M%S") + '.xlsx'
wb.save(filename)

# 为用户打开文件：
os.chdir(sys.path[0])
os.system('start excel.exe "%s\\%s"' % (sys.path[0], filename, ))

```

```

percent_format = workbook.add_format({'num_format': '0%'})
percent_with_decimal = workbook.add_format({'num_format': '0.0%'})
bold = workbook.add_format({'bold': True})
red_font = workbook.add_format({'font_color': 'red'})
remove_format = workbook.add_format()

# add a new sheet
worksheet = workbook.add_worksheet()

# set the width of column A
worksheet.set_column('A:A', 30, )

# set column B to 20 and include the percent format we created earlier
worksheet.set_column('B:B', 20, percent_format)

# remove formatting from the first row (change in height=None)
worksheet.set_row('0:0', None, remove_format)

workbook.close()

```

## Section 164.3: Put list data into a Excel's file

```

import os, sys
from openpyxl import Workbook
from datetime import datetime

dt = datetime.now()
list_values = [[ "01/01/2016", "05:00:00", 3], \
               [ "01/02/2016", "06:00:00", 4], \
               [ "01/03/2016", "07:00:00", 5], \
               [ "01/04/2016", "08:00:00", 6], \
               [ "01/05/2016", "09:00:00", 7]]

# Create a Workbook on Excel:
wb = Workbook()
sheet = wb.active
sheet.title = 'data'

# Print the titles into Excel Workbook:
row = 1
sheet['A'+str(row)] = 'Date'
sheet['B'+str(row)] = 'Hour'
sheet['C'+str(row)] = 'Value'

# Populate with data
for item in list_values:
    row += 1
    sheet['A'+str(row)] = item[0]
    sheet['B'+str(row)] = item[1]
    sheet['C'+str(row)] = item[2]

# Save a file by date:
filename = 'data_' + dt.strftime("%Y%m%d_%I%M%S") + '.xlsx'
wb.save(filename)

# Open the file for the user:
os.chdir(sys.path[0])
os.system('start excel.exe "%s\\%s"' % (sys.path[0], filename, ))

```

## 第164.4节：OpenPyXL

OpenPyXL 是一个用于操作和创建内存中 xlsx/xlsm/xltx/xltm 工作簿的模块。

操作和读取现有工作簿：

```
import openpyxl as opx
#要修改现有的工作簿，我们通过引用其路径来定位它
workbook = opx.load_workbook(workbook_path)
```

load\_workbook() 包含参数 read\_only，将其设置为 True 会以只读模式加载工作簿，这在读取较大的 xlsx 文件时很有用：

```
workbook = opx.load_workbook(workbook_path, read_only=True)
```

一旦将工作簿加载到内存中，就可以使用 workbook.sheets 访问各个工作表

```
first_sheet = workbook.worksheets[0]
```

如果想指定可用工作表的名称，可以使用 workbook.get\_sheet\_names()。

```
sheet = workbook.get_sheet_by_name('Sheet Name')
```

最后，可以使用 sheet.rows 访问工作表的行。要遍历工作表中的行，使用：

```
for row in sheet.rows:
    print row[0].value
```

由于 rows 中的每一 row 都是 Cell 的列表，使用 Cell.value 来获取单元格的内容。

在内存中创建一个新的工作簿：

```
#调用 Workbook() 函数会在内存中创建一个新的工作簿
wb = opx.Workbook()

#然后我们可以在 wb 中创建一个新的工作表
ws = wb.create_sheet('工作表名称', 0) #0 指的是工作表在 wb 中的索引顺序
```

可以通过 openpyxl 更改多个标签属性，例如 tabColor：

```
ws.sheet_properties.tabColor = 'FFC0CB'
```

要保存我们创建的工作簿，最后执行：

```
wb.save('filename.xlsx')
```

## 第164.5节：使用xlsxwriter创建Excel图表

```
import xlsxwriter

#示例数据
chart_data = [
    {'name': 'Lorem', 'value': 23},
    {'name': 'Ipsum', 'value': 48},
    {'name': 'Dolor', 'value': 15},
```

## Section 164.4: OpenPyXL

OpenPyXL is a module for manipulating and creating xlsx/xlsm/xltx/xltm workbooks in memory.

Manipulating and reading an existing workbook:

```
import openpyxl as opx
#To change an existing workbook we located it by referencing its path
workbook = opx.load_workbook(workbook_path)
```

load\_workbook() contains the parameter read\_only, setting this to True will load the workbook as read\_only, this is helpful when reading larger xlsx files:

```
workbook = opx.load_workbook(workbook_path, read_only=True)
```

Once you have loaded the workbook into memory, you can access the individual sheets using workbook.sheets

```
first_sheet = workbook.worksheets[0]
```

If you want to specify the name of an available sheets, you can use workbook.get\_sheet\_names().

```
sheet = workbook.get_sheet_by_name('Sheet Name')
```

Finally, the rows of the sheet can be accessed using sheet.rows. To iterate over the rows in a sheet, use:

```
for row in sheet.rows:
    print row[0].value
```

Since each row in rows is a list of Cells, use Cell.value to get the contents of the Cell.

Creating a new Workbook in memory:

```
#Calling the Workbook() function creates a new book in memory
wb = opx.Workbook()
```

```
#We can then create a new sheet in the wb
ws = wb.create_sheet('Sheet Name', 0) #0 refers to the index of the sheet order in the wb
```

Several tab properties may be changed through openpyxl, for example the tabColor:

```
ws.sheet_properties.tabColor = 'FFC0CB'
```

To save our created workbook we finish with:

```
wb.save('filename.xlsx')
```

## Section 164.5: Create excel charts with xlsxwriter

```
import xlsxwriter

# sample data
chart_data = [
    {'name': 'Lorem', 'value': 23},
    {'name': 'Ipsum', 'value': 48},
    {'name': 'Dolor', 'value': 15},
```

```

{'name': 'Sit', 'value': 8},
{'name': 'Amet', 'value': 32}
]

# Excel文件路径
xls_file = 'chart.xlsx'

# 工作簿
workbook = xlsxwriter.Workbook(xls_file)

# 将工作表添加到工作簿
worksheet = workbook.add_worksheet()

row_ = 0
col_ = 0

# 写入表头
worksheet.write(row_, col_, 'NAME')
col_ += 1
worksheet.write(row_, col_, 'VALUE')
row_ += 1

# 写入示例数据
for item in chart_data:
    col_ = 0
    worksheet.write(row_, col_, item['name'])
    col_ += 1
    worksheet.write(row_, col_, item['value'])
    row_ += 1

# 创建饼图
pie_chart = workbook.add_chart({'type': 'pie'})

# 向饼图添加系列
pie_chart.add_series({
    'name': '系列名称',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%' % row_,
    'marker': {'type': 'circle'}
})
# 插入饼图
worksheet.insert_chart('D2', pie_chart)

# 创建柱状图
column_chart = workbook.add_chart({'type': 'column'})

# 将序列添加到柱状图
column_chart.add_series({
    'name': '系列名称',
    'categories': '=Sheet1!$A$3:$A$%' % row_,
    'values': '=Sheet1!$B$3:$B$%' % row_,
    'marker': {'type': 'circle'}
})
# 插入柱状图
worksheet.insert_chart('D20', column_chart)

workbook.close()

```

结果：

```

{'name': 'Sit', 'value': 8},
{'name': 'Amet', 'value': 32}
]

# excel file path
xls_file = 'chart.xlsx'

# the workbook
workbook = xlsxwriter.Workbook(xls_file)

# add worksheet to workbook
worksheet = workbook.add_worksheet()

row_ = 0
col_ = 0

# write headers
worksheet.write(row_, col_, 'NAME')
col_ += 1
worksheet.write(row_, col_, 'VALUE')
row_ += 1

# write sample data
for item in chart_data:
    col_ = 0
    worksheet.write(row_, col_, item['name'])
    col_ += 1
    worksheet.write(row_, col_, item['value'])
    row_ += 1

# create pie chart
pie_chart = workbook.add_chart({'type': 'pie'})

# add series to pie chart
pie_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%' % row_,
    'values': '=Sheet1!$B$3:$B$%' % row_,
    'marker': {'type': 'circle'}
})
# insert pie chart
worksheet.insert_chart('D2', pie_chart)

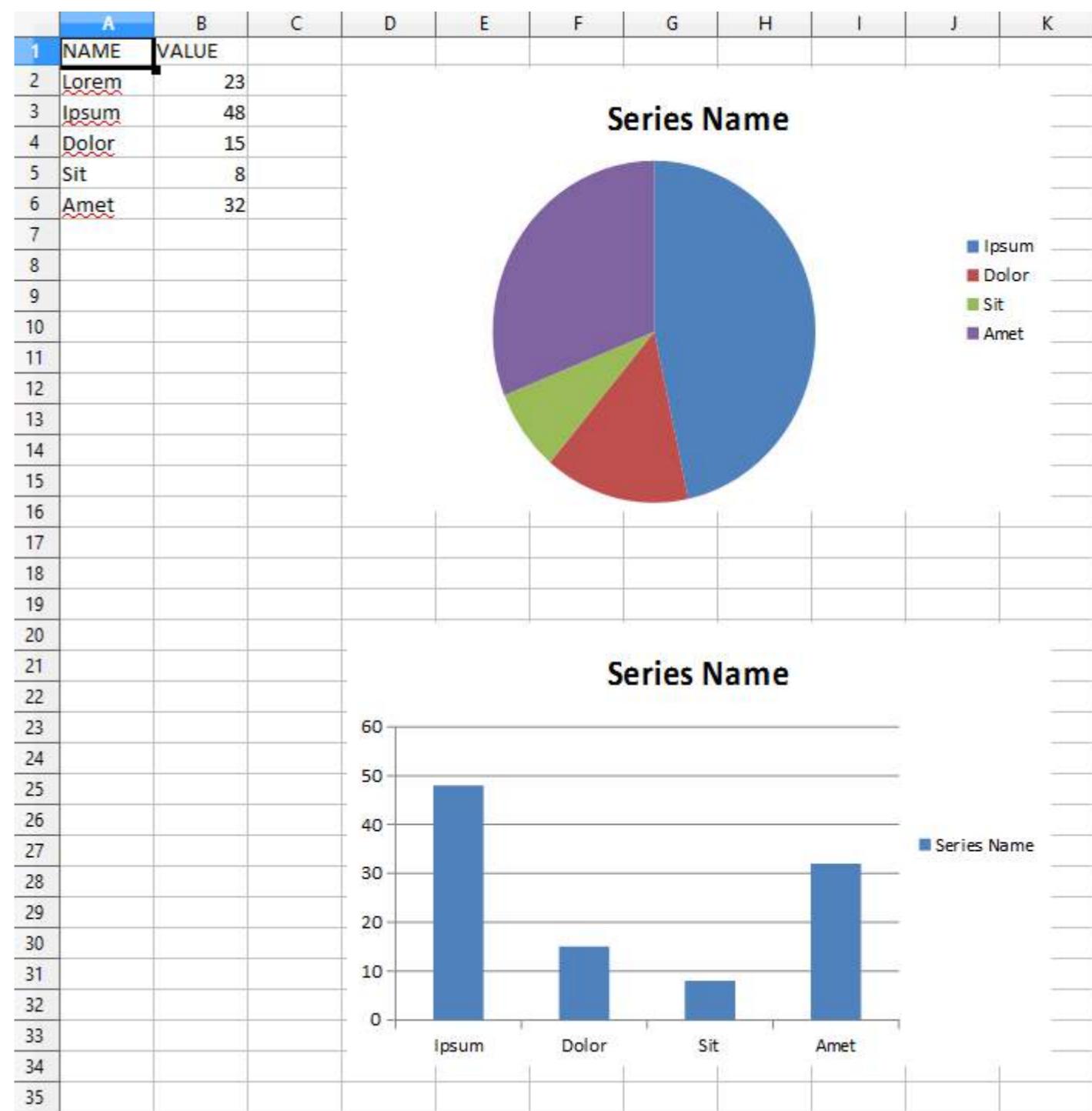
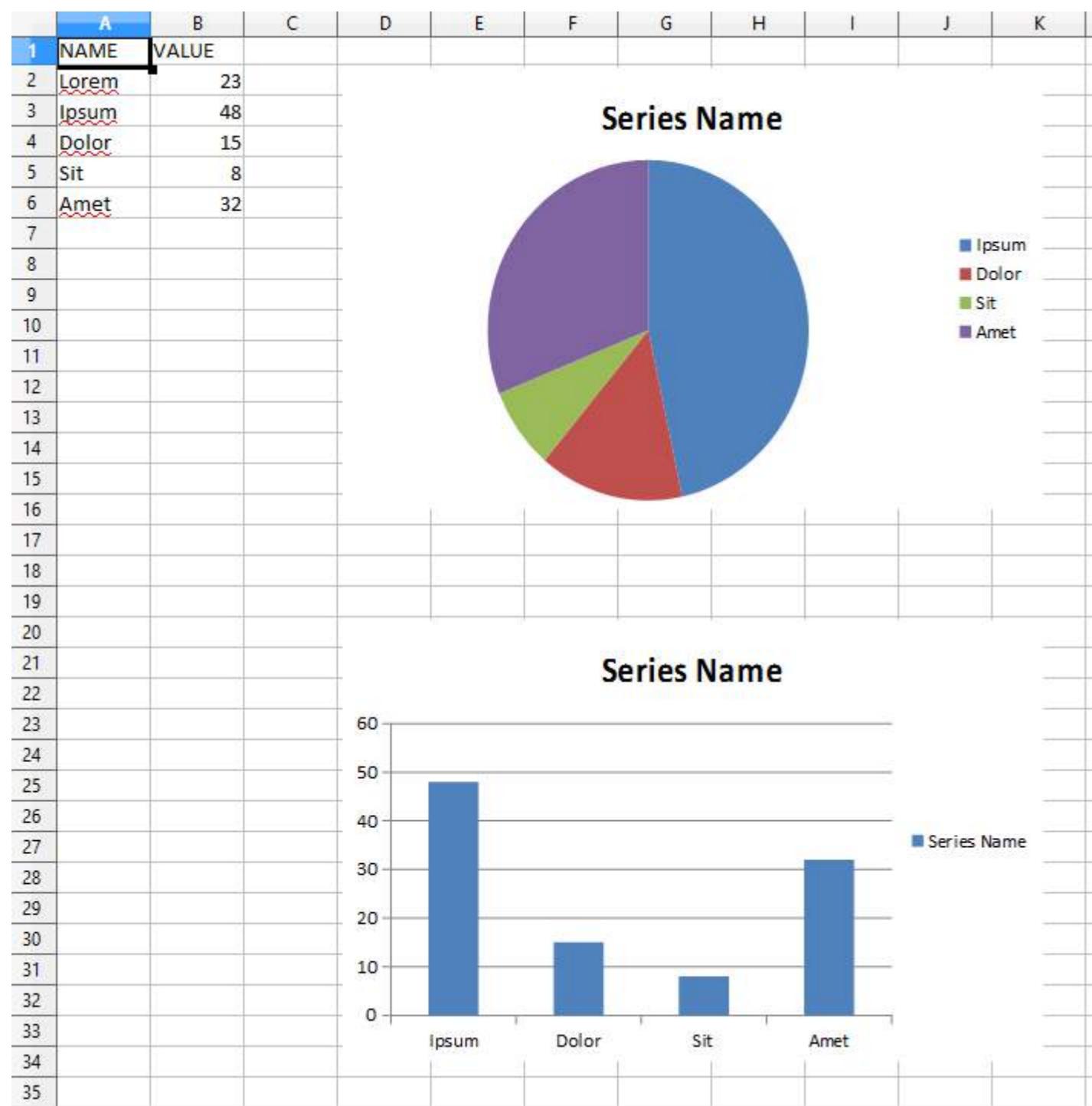
# create column chart
column_chart = workbook.add_chart({'type': 'column'})

# add serie to column chart
column_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%' % row_,
    'values': '=Sheet1!$B$3:$B$%' % row_,
    'marker': {'type': 'circle'}
})
# insert column chart
worksheet.insert_chart('D20', column_chart)

workbook.close()

```

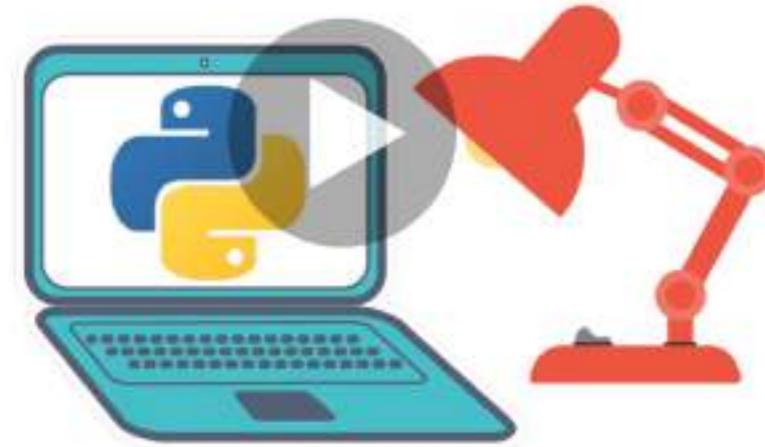
Result:



# 视频：完整的Python训练营：从零开始成为Python 3高手

像专业人士一样学习Python！从基础开始，直到创建你自己的应用程序和游戏！

## COMPLETE PYTHON 3 BOOTCAMP



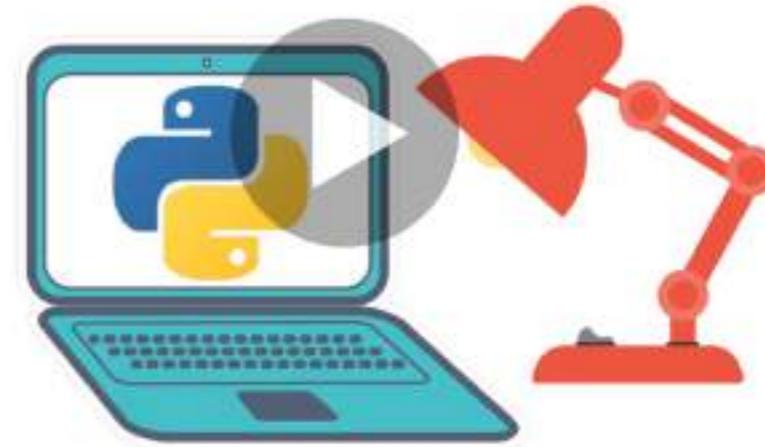
- ✓ 学习专业使用Python，掌握Python 2和Python 3！
- ✓ 用Python创建游戏，比如井字棋和二十一点！
- ✓ 学习高级Python功能，如collections模块和时间戳的使用！
- ✓ 学习使用面向对象编程和类！
- ✓ 理解复杂主题，如装饰器。
- ✓ 理解如何使用Jupyter Notebook并创建.py文件
- ✓ 了解如何在Jupyter Notebook系统中创建图形用户界面（GUI）！
- ✓ 从零开始全面掌握Python！

今天观看 →

# VIDEO: Complete Python Bootcamp: Go from zero to hero in Python 3

Learn Python like a Professional! Start from the basics and go all the way to creating your own applications and games!

## COMPLETE PYTHON 3 BOOTCAMP

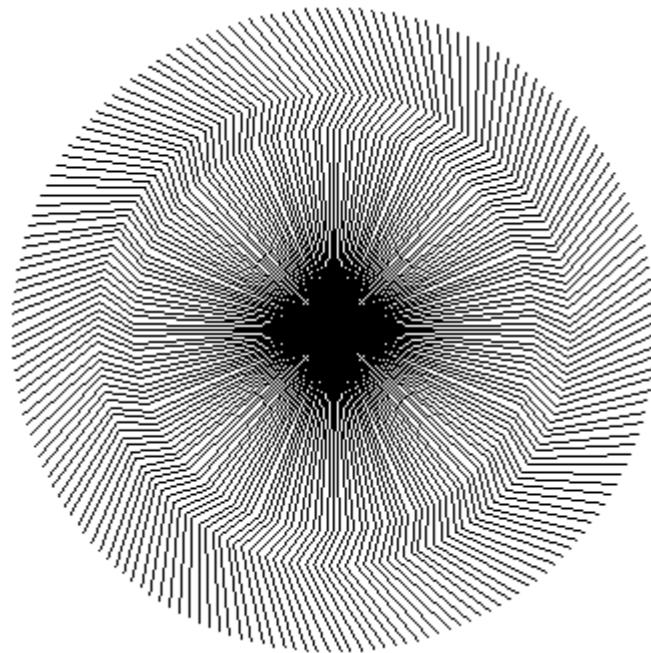


- ✓ Learn to use Python professionally, learning both Python 2 and Python 3!
- ✓ Create games with Python, like Tic Tac Toe and Blackjack!
- ✓ Learn advanced Python features, like the collections module and how to work with timestamps!
- ✓ Learn to use Object Oriented Programming with classes!
- ✓ Understand complex topics, like decorators.
- ✓ Understand how to use both the Jupyter Notebook and create .py files
- ✓ Get an understanding of how to create GUIs in the Jupyter Notebook system!
- ✓ Build a complete understanding of Python from the ground up!

Watch Today →

# 第165章：海龟图形

## 第165.1节：忍者扭转（海龟图形）

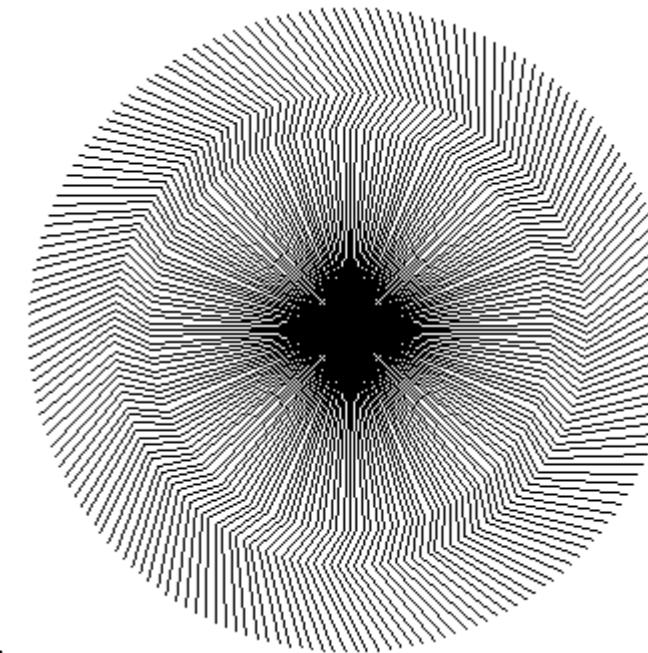


这里是一个海龟图形忍者扭转：

```
import turtle  
  
ninja = turtle.Turtle()  
  
ninja.speed(10)  
  
for i in range(180):  
    ninja.forward(100)  
    ninja.right(30)  
    ninja.forward(20)  
    ninja.left(60)  
    ninja.forward(50)  
    ninja.right(30)  
  
    ninja.penup()  
    ninja.setposition(0, 0)  
    ninja.pendown()  
  
    ninja.right(2)  
  
turtle.done()
```

# Chapter 165: Turtle Graphics

## Section 165.1: Ninja Twist (Turtle Graphics)



Here a Turtle Graphics Ninja Twist:

```
import turtle  
  
ninja = turtle.Turtle()  
  
ninja.speed(10)  
  
for i in range(180):  
    ninja.forward(100)  
    ninja.right(30)  
    ninja.forward(20)  
    ninja.left(60)  
    ninja.forward(50)  
    ninja.right(30)  
  
    ninja.penup()  
    ninja.setposition(0, 0)  
    ninja.pendown()  
  
    ninja.right(2)  
  
turtle.done()
```

# 第166章：Python持久化

参数	详细信息
obj	obj的pickle表示，写入打开的文件对象file中
协议	一个整数，告诉pickle使用指定的协议，0- ASCII, 1- 旧的二进制格式
文件	file参数必须有write()方法wb用于dump方法，加载时需有read()方法rb

## 第166.1节：Python持久化

像数字、列表、字典、嵌套结构和类实例对象这样的对象存在于计算机的内存中，一旦脚本结束就会丢失。

pickle 将数据持久化存储在单独的文件中。

对象的 pickled 表示在所有情况下始终是字节对象，因此必须以wb模式打开文件以存储数据，以 rb模式打开文件以从 pickle 加载数据。

数据可以是任何类型，例如，

```
data={'a':'some_value',
      'b':[9,4,7],
      'c':['some_str','another_str','spam','ham'],
      'd':{'key':'nested_dictionary'},
      }
```

### 存储数据

```
import pickle
file=open('filename','wb') #以二进制写入模式打开文件对象
pickle.dump(data,file) #将数据写入文件对象
file.close() #关闭文件以完成写入
```

### 加载数据

```
import pickle
file=open('filename','rb') #以二进制读取模式打开文件对象
data=pickle.load(file) #加载数据
file.close()

>>>data
{'b': [9, 4, 7], 'a': 'some_value', 'd': {'key': 'nested_dictionary'},
 'c': ['some_str', 'another_str', 'spam', 'ham']}
```

### 以下类型可以被序列化 (pickle)

1. None、True 和 False
2. 整数、浮点数、复数
3. 字符串、字节、字节数组
4. 只包含可序列化对象的元组、列表、集合和字典
5. 在模块顶层定义的函数（使用 def，非 lambda）
6. 在模块顶层定义的内置函数
7. 在模块顶层定义的类
8. 此类的实例，其 dict 或调用 getstate() 的结果

# Chapter 166: Python Persistence

Parameter	Details
obj	pickled representation of obj to the open file object file
protocol	an integer, tells the pickler to use the given protocol,0-ASCII, 1- old binary format
file	The file argument must have a write() method wb for dump method and for loading read() method rb

## Section 166.1: Python Persistence

Objects like numbers, lists, dictionaries,nested structures and class instance objects live in your computer's memory and are lost as soon as the script ends.

pickle stores data persistently in separate file.

pickled representation of an object is always a bytes object in all cases so one must open files in wb to store data and rb to load data from pickle.

the data may be of any kind, for example,

```
data={'a':'some_value',
      'b':[9,4,7],
      'c':['some_str','another_str','spam','ham'],
      'd':{'key':'nested_dictionary'},
      }
```

### Store data

```
import pickle
file=open('filename','wb') #file object in binary write mode
pickle.dump(data,file) #dump the data in the file object
file.close() #close the file to write into the file
```

### Load data

```
import pickle
file=open('filename','rb') #file object in binary read mode
data=pickle.load(file) #load the data back
file.close()

>>>data
{'b': [9, 4, 7], 'a': 'some_value', 'd': {'key': 'nested_dictionary'},
 'c': ['some_str', 'another_str', 'spam', 'ham']}
```

### The following types can be pickled

1. None, True, and False
2. integers, floating point numbers, complex numbers
3. strings, bytes, bytearray
4. tuples, lists, sets, and dictionaries containing only picklable objects
5. functions defined at the top level of a module (using def, not lambda)
6. built-in functions defined at the top level of a module
7. classes that are defined at the top level of a module
8. instances of such classes whose dict or the result of calling getstate()

## 第166.2节：保存和加载的函数工具

将数据保存到文件及从文件加载

```
import pickle
def save(filename,object):
    file=open(filename,'wb')
    pickle.dump(object,file)
    file.close()

def load(filename):
    file=open(filename,'rb')
    object=pickle.load(file)
    file.close()
    return object

>>>list_object=[1,1,2,3,5,8,'a','e','i','o','u']
>>>save(list_file,list_object)
>>>new_list=load(list_file)
>>>new_list
[1, 1, 2, 3, 5, 8, 'a', 'e', 'i', 'o', 'u']
```

## Section 166.2: Function utility for save and load

Save data to and from file

```
import pickle
def save(filename,object):
    file=open(filename,'wb')
    pickle.dump(object,file)
    file.close()

def load(filename):
    file=open(filename,'rb')
    object=pickle.load(file)
    file.close()
    return object

>>>list_object=[1,1,2,3,5,8,'a','e','i','o','u']
>>>save(list_file,list_object)
>>>new_list=load(list_file)
>>>new_list
[1, 1, 2, 3, 5, 8, 'a', 'e', 'i', 'o', 'u']
```

# 第167章：设计模式

设计模式是软件开发中常见问题的一般解决方案。本章节专门旨在提供Python中常见设计模式的示例。

## 第167.1节：设计模式及单例模式介绍

设计模式为软件设计中常见的问题提供了解决方案。设计模式最初由GoF（四人帮）提出，他们将常见模式描述为反复出现的问题及这些问题的解决方案。

**设计模式有四个基本要素：**

1. 模式名称是我们用来描述设计问题、其解决方案及后果的一个句柄，一两个字。
2. 该问题描述了何时应用该模式。
3. 该解决方案描述了构成设计的元素、它们之间的关系、职责，以及合作。
4. 后果是应用该模式的结果和权衡。

**设计模式的优点：**

1. 它们可以在多个项目中重复使用。
2. 问题的架构层面可以被解决
3. 它们经过时间考验且被充分验证，这是开发者和架构师的经验
4. 它们具有可靠性和依赖性

**设计模式可以分为三类：**

1. 创建型模式
2. 结构型模式
3. 行为型模式

创建型模式 - 它们关注对象如何被创建，并且隔离对象创建的细节。

结构型模式 - 它们设计类和对象的结构，使它们能够组合以实现更大的结果。

行为型模式 - 它们关注对象之间的交互和对象的职责。

**单例模式：**

它是一种创建型模式，提供了一种机制，使得某一类型只有一个对象，并提供一个全局访问点。

例如，单例模式可以用于数据库操作中，我们希望数据库对象保持数据一致性。

**实现**

我们可以通过只创建 Singleton 类的一个实例并再次返回同一个对象来实现 Python 中的单例模式。

# Chapter 167: Design Patterns

A design pattern is a general solution to a commonly occurring problem in software development. This documentation topic is specifically aimed at providing examples of common design patterns in Python.

## Section 167.1: Introduction to design patterns and Singleton Pattern

Design Patterns provide solutions to the commonly occurring problems in software design. The design patterns were first introduced by GoF(Gang of Four) where they described the common patterns as problems which occur over and over again and solutions to those problems.

**Design patterns have four essential elements:**

1. The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
2. The problem describes when to apply the pattern.
3. The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations.
4. The consequences are the results and trade-offs of applying the pattern.

**Advantages of design patterns:**

1. They are reusable across multiple projects.
2. The architectural level of problems can be solved
3. They are time-tested and well-proven, which is the experience of developers and architects
4. They have reliability and dependence

**Design patterns can be classified into three categories:**

1. Creational Pattern
2. Structural Pattern
3. Behavioral Pattern

Creational Pattern - They are concerned with how the object can be created and they isolate the details of object creation.

Structural Pattern - They design the structure of classes and objects so that they can compose to achieve larger results.

Behavioral Pattern - They are concerned with interaction among objects and responsibility of objects.

**Singleton Pattern:**

It is a type of creational pattern which provides a mechanism to have only one and one object of a given type and provides a global point of access.

e.g. Singleton can be used in database operations, where we want database object to maintain data consistency.

**Implementation**

We can implement Singleton Pattern in Python by creating only one instance of Singleton class and serving the same object again.

```

class Singleton(object):
    def __new__(cls):
        # hasattr 方法检查类对象是否具有实例属性。
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
            return cls.instance

    s = Singleton()
    print ("对象已创建", s)

    s1 = Singleton()
    print ("Object2 created", s1)

```

输出：

```

('对象已创建', <__main__.Singleton 对象 位于 0x10a7cc310>)
('对象2已创建', <__main__.Singleton 对象 位于 0x10a7cc310>)

```

请注意，在像C++或Java这样的语言中，这种模式是通过将构造函数设为私有并创建一个静态方法来实现对象初始化的。这样，第一次调用时会创建一个对象，之后类会返回同一个对象。但在Python中，我们没有办法创建私有构造函数。

## 工厂模式

工厂模式也是一种创建型模式。“工厂”一词意味着一个类负责创建其他类型的对象。存在一个作为工厂的类，它拥有相关的对象和方法。客户端通过调用带有特定参数的方法来创建对象，工厂则创建所需类型的对象并返回给客户端。

```

from abc import ABCMeta, abstractmethod

class 音乐():
    __metaclass__ = ABCMeta
    @abstractmethod
    def 播放(self):
        通过

class Mp3(音乐):
    def 播放(self):
        print ("正在播放 .mp3 音乐 !")

class Ogg(音乐):
    def 播放(self):
        print ("正在播放 .ogg 音乐 !")

class 音乐工厂(object):
    def 播放声音(self, 对象类型):
        return eval(对象类型)().do_play()

if __name__ == "__main__":
    mf = 音乐工厂()
    music = input("你想播放哪种音乐 Mp3 还是 Ogg")
    mf.play_sound(music)

```

输出：

```

你想播放哪种音乐 Mp3 还是 Ogg"Ogg"
正在播放 .ogg 音乐 !

```

```

class Singleton(object):
    def __new__(cls):
        # hasattr method checks if the class object an instance property or not.
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
            return cls.instance

    s = Singleton()
    print ("Object created", s)

    s1 = Singleton()
    print ("Object2 created", s1)

```

Output:

```

('Object created', <__main__.Singleton object at 0x10a7cc310>)
('Object2 created', <__main__.Singleton object at 0x10a7cc310>)

```

Note that in languages like C++ or Java, this pattern is implemented by making the constructor private and creating a static method that does the object initialization. This way, one object gets created on the first call and class returns the same object thereafter. But in Python, we do not have any way to create private constructors.

## Factory Pattern

Factory pattern is also a Creational pattern. The term factory means that a class is responsible for creating objects of other types. There is a class that acts as a factory which has objects and methods associated with it. The client creates an object by calling the methods with certain parameters and factory creates the object of the desired type and return it to the client.

```

from abc import ABCMeta, abstractmethod

class Music():
    __metaclass__ = ABCMeta
    @abstractmethod
    def do_play(self):
        pass

class Mp3(Music):
    def do_play(self):
        print ("Playing .mp3 music!")

class Ogg(Music):
    def do_play(self):
        print ("Playing .ogg music!")

class MusicFactory(object):
    def play_sound(self, object_type):
        return eval(object_type)().do_play()

if __name__ == "__main__":
    mf = MusicFactory()
    music = input("Which music you want to play Mp3 or Ogg")
    mf.play_sound(music)

```

Output:

```

Which music you want to play Mp3 or Ogg"Ogg"
Playing .ogg music!

```

音乐工厂 是这里的工厂类，根据用户提供的选择创建 Mp3 或 Ogg 类型的对象。

## 第167.2节：策略模式

这种设计模式称为策略模式。它用于定义一系列算法，将每个算法封装起来，并使它们可以互换。策略设计模式使算法可以独立于使用它的客户端而变化。

例如，动物可以用许多不同的方式“行走”。行走可以被视为一种策略，由不同类型的动物来实现：

```
from types import MethodType

class 动物(object):

    def __init__(self, *args, **kwargs):
        self.name = kwargs.pop('name', None) or '动物'
        if kwargs.get('walk', None):
            self.walk = MethodType(kwargs.pop('walk'), self)

    def walk(self):
        """
        使动物实例行走      行走功能是一种策略，旨
        在由不同类型的动物分别实现。
        """

    message = '{} 应该实现一个 walk 方法'.format(
        self.__class__.__name__)
    raise NotImplementedError(message)
```

```
# 这里有一些可以与 Animal 一起使用的不同行走算法
def snake_walk(self):
    print('我正在左右爬行，因为我是{}。'.format(self.name))

def four_legged_animal_walk(self):
    print('我用四条腿走路，因为我是{}。'.format(
        self.name))

def two_legged_animal_walk(self):
    print('我站在两条腿上走路，因为我是{}。'.format(
        self.name))
```

运行此示例将产生以下输出：

```
generic_animal = Animal()
king_cobra = Animal(name='眼镜王蛇', walk=蛇行走方式)
elephant = Animal(name='大象', walk=四足动物行走方式)
kangaroo = Animal(name='袋鼠', walk=两足动物行走方式)

kangaroo.walk()
elephant.walk()
king_cobra.walk()
# 这将引发 NotImplementedError，提醒程序员
# walk 方法是作为一种策略使用的。
generic_animal.walk()
```

MusicFactory is the factory class here that creates either an object of type Mp3 or Ogg depending on the choice user provides.

## Section 167.2: Strategy Pattern

This design pattern is called Strategy Pattern. It is used to define a family of algorithms, encapsulates each one, and make them interchangeable. Strategy design pattern lets an algorithm vary independently from clients that use it.

For example, animals can "walk" in many different ways. Walking could be considered a strategy that is implemented by different types of animals:

```
from types import MethodType

class Animal(object):

    def __init__(self, *args, **kwargs):
        self.name = kwargs.pop('name', None) or 'Animal'
        if kwargs.get('walk', None):
            self.walk = MethodType(kwargs.pop('walk'), self)

    def walk(self):
        """
        Cause animal instance to walk

        Walking functionality is a strategy, and is intended to
        be implemented separately by different types of animals.
        """

    message = '{} should implement a walk method'.format(
        self.__class__.__name__)
    raise NotImplementedError(message)

    # Here are some different walking algorithms that can be used with Animal
    def snake_walk(self):
        print('I am slithering side to side because I am a {}.'.format(self.name))

    def four_legged_animal_walk(self):
        print('I am using all four of my legs to walk because I am a(n) {}.'.format(
            self.name))

    def two_legged_animal_walk(self):
        print('I am standing up on my two legs to walk because I am a {}.'.format(
            self.name))
```

Running this example would produce the following output:

```
generic_animal = Animal()
king_cobra = Animal(name='King Cobra', walk=snake_walk)
elephant = Animal(name='Elephant', walk=four_legged_animal_walk)
kangaroo = Animal(name='Kangaroo', walk=two_legged_animal_walk)

kangaroo.walk()
elephant.walk()
king_cobra.walk()
# This one will Raise a NotImplementedError to let the programmer
# know that the walk method is intended to be used as a strategy.
generic_animal.walk()
```

```

# 输出：
#
# 我用两条腿站立行走，因为我是一只袋鼠。
# 我用四条腿行走，因为我是一头大象。
# 我左右摆动爬行，因为我是一条眼镜王蛇。
# Traceback (most recent call last):
#   文件 "./strategy.py", 第 56 行, 位于 <module>
#     generic_animal.walk()
#   文件 "./strategy.py", 第30行, 位于 walk
#     引发 NotImplementedError(message)
# NotImplementedError: Animal 应该实现一个 walk 方法

```

请注意，在像 C++ 或 Java 这样的语言中，这种模式是通过抽象类或接口来定义策略实现的。在 Python 中，更合理的做法是定义一些外部函数，然后使用 `types.MethodType` 动态地添加到类中。

### 第167.3节：代理（Proxy）

代理对象通常用于确保对另一个对象的受保护访问，我们不希望将安全需求污染其内部业务逻辑。

假设我们想保证只有具有特定权限的用户才能访问资源。

代理定义：（它确保只有实际可以查看预订的用户才能使用 `reservation_service`）

```

来自 datetime 导入 date
来自 operator 导入 attrgetter

类 Proxy:
    定义 __init__(self, current_user, reservation_service):
        self.current_user = current_user
        self.reservation_service = reservation_service

    定义 highest_total_price_reservations(self, date_from, date_to, reservations_count):
        如果self.current_user.can_see_reservations:
            返回 self.reservation_service.highest_total_price_reservations(
                date_from,
                date_to,
                reservations_count
            )
        else:
            返回 []

```

#模型和预订服务：

```

class 预订(Reservation):
    定义 __init__(self, 日期(date), 总价(total_price)):
        self.日期(date) = 日期(date)
        self.总价(total_price) = 总价(total_price)

class 预订服务(ReservationService):
    定义 highest_total_price_reservations(self, date_from, date_to, reservations_count):
        # 通常会从数据库/外部服务读取
reservations = [
    Reservation(date(2014, 5, 15), 100),
    Reservation(date(2017, 5, 15), 10),
    Reservation(date(2017, 1, 15), 50)
]

```

```

# OUTPUT:
#
# I am standing up on my two legs to walk because I am a Kangaroo.
# I am using all four of my legs to walk because I am a(n) Elephant.
# I am slithering side to side because I am a King Cobra.
# Traceback (most recent call last):
#   File "./strategy.py", line 56, in <module>
#     generic_animal.walk()
#   File "./strategy.py", line 30, in walk
#     raise NotImplementedError(message)
# NotImplementedError: Animal should implement a walk method

```

Note that in languages like C++ or Java, this pattern is implemented using an abstract class or an interface to define a strategy. In Python it makes more sense to just define some functions externally that can be added dynamically to a class using `types.MethodType`.

### Section 167.3: Proxy

Proxy object is often used to ensure guarded access to another object, which internal business logic we don't want to pollute with safety requirements.

Suppose we'd like to guarantee that only user of specific permissions can access resource.

Proxy definition: (it ensure that only users which actually can see reservations will be able to consumer `reservation_service`)

```

from datetime import date
from operator import attrgetter

class Proxy:
    定义 __init__(self, current_user, reservation_service):
        self.current_user = current_user
        self.reservation_service = reservation_service

    定义 highest_total_price_reservations(self, date_from, date_to, reservations_count):
        如果 self.current_user.can_see_reservations:
            返回 self.reservation_service.highest_total_price_reservations(
                date_from,
                date_to,
                reservations_count
            )
        else:
            返回 []

```

#Models and ReservationService:

```

class Reservation:
    定义 __init__(self, date, total_price):
        self.date = date
        self.total_price = total_price

class ReservationService:
    定义 highest_total_price_reservations(self, date_from, date_to, reservations_count):
        # normally it would be read from database/external service
reservations = [
    Reservation(date(2014, 5, 15), 100),
    Reservation(date(2017, 5, 15), 10),
    Reservation(date(2017, 1, 15), 50)
]

```

```

filtered_reservations = [r for r in reservations if (date_from <= r.date <= date_to)]

sorted_reservations = sorted(filtered_reservations, key=attrgetter('total_price'),
reverse=True)

return sorted_reservations[0:reservations_count]

class User:
    def __init__(self, can_see_reservations, name):
        self.can_see_reservations = can_see_reservations
        self.name = name

#Consumer service:

class StatsService:
    def __init__(self, reservation_service):
        self.reservation_service = reservation_service

    def year_top_100_reservations_average_total_price(self, year):
        reservations = self.reservation_service.highest_total_price_reservations(
            date(year, 1, 1),
            date(year, 12, 31),
            1
        )

        if len(reservations) > 0:
            total = sum(r.total_price for r in reservations)

            return total / len(reservations)
        else:
            return 0

#Test:
def test(user, year):
    reservations_service = Proxy(user, ReservationService())
    stats_service = StatsService(reservations_service)
    average_price = stats_service.year_top_100_reservations_average_total_price(year)
    print("{0} 将看到: {1}.format(user.name, average_price))

test(User(True, "管理员约翰"), 2017)
test(User(False, "访客"), 2017)

```

## 优势

- 当访问权限更改时，我们避免对ReservationService进行任何更改。
- 我们没有在服务中混合业务相关数据 (date\_from, date\_to, reservations\_count) 和领域无关的概念（用户权限）。
- 消费者 (StatsService) 也不涉及权限相关的逻辑

## 注意事项

- 代理接口始终与其隐藏的对象完全相同，因此使用被代理包装的服务的用户甚至不会察觉代理的存在。

```

filtered_reservations = [r for r in reservations if (date_from <= r.date <= date_to)]

sorted_reservations = sorted(filtered_reservations, key=attrgetter('total_price'),
reverse=True)

return sorted_reservations[0:reservations_count]

class User:
    def __init__(self, can_see_reservations, name):
        self.can_see_reservations = can_see_reservations
        self.name = name

#Consumer service:

class StatsService:
    def __init__(self, reservation_service):
        self.reservation_service = reservation_service

    def year_top_100_reservations_average_total_price(self, year):
        reservations = self.reservation_service.highest_total_price_reservations(
            date(year, 1, 1),
            date(year, 12, 31),
            1
        )

        if len(reservations) > 0:
            total = sum(r.total_price for r in reservations)

            return total / len(reservations)
        else:
            return 0

#Test:
def test(user, year):
    reservations_service = Proxy(user, ReservationService())
    stats_service = StatsService(reservations_service)
    average_price = stats_service.year_top_100_reservations_average_total_price(year)
    print("{0} will see: {1}.format(user.name, average_price))

test(User(True, "John the Admin"), 2017)
test(User(False, "Guest"), 2017)

```

## BENEFITS

- we're avoiding any changes in ReservationService when access restrictions are changed.
- we're not mixing business related data (date\_from, date\_to, reservations\_count) with domain unrelated concepts (user permissions) in service.
- Consumer (StatsService) is free from permissions related logic as well

## CAVEATS

- Proxy interface is always exactly the same as the object it hides, so that user that consumes service wrapped by proxy wasn't even aware of proxy presence.

# 第168章：hashlib

hashlib 实现了许多不同安全哈希和消息摘要算法的通用接口。包括 FIPS 安全哈希算法 SHA1、SHA224、SHA256、SHA384 和 SHA512。

## 第 168.1 节：字符串的 MD5 哈希

该模块实现了许多不同安全哈希和消息摘要算法的通用接口。

包括 FIPS 安全哈希算法 SHA1、SHA224、SHA256、SHA384 和 SHA512（定义于 FIPS 180-2）以及 RSA 的 MD5 算法（定义于互联网 RFC 1321）。

每种哈希类型都有一个同名的构造方法。所有方法都返回具有相同简单接口的哈希对象。例如：使用sha1()创建一个 SHA1 哈希对象。

`hash.sha1()`

该模块中始终存在的哈希算法构造函数有`md5()`、`sha1()`、`sha224()`、`sha256()`、`sha384()`和`sha512()`。

现在你可以使用`update()`方法向该对象输入任意字符串。在任何时候，你都可以使用`digest()`或`hexdigest()`方法获取到目前为止输入字符串连接后的摘要。

`hash.update(arg)`

使用字符串 `arg` 更新哈希对象。重复调用等同于一次调用，参数为所有参数的连接：`m.update(a); m.update(b)` 等同于 `m.update(a+b)`。

`hash.digest()`

返回到目前为止传递给 `update()` 方法的字符串的摘要。这是一个长度为 `digest_size` 字节的字符串，可能包含非 ASCII 字符，包括空字节。

`hash.hexdigest()`

类似于 `digest()`，但摘要以双倍长度的字符串返回，仅包含十六进制数字。  
这可以用于在电子邮件或其他非二进制环境中安全地交换该值。

下面是一个示例：

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbdb\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.hexdigest()
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
>>> m.digest_size
16
```

# Chapter 168: hashlib

hashlib implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512.

## Section 168.1: MD5 hash of a string

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet RFC 1321).

There is one constructor method named for each type of hash. All return a hash object with the same simple interface. For example: use `sha1()` to create a SHA1 hash object.

`hash.sha1()`

Constructors for hash algorithms that are always present in this module are `md5()`, `sha1()`, `sha224()`, `sha256()`, `sha384()`, and `sha512()`.

You can now feed this object with arbitrary strings using the `update()` method. At any point you can ask it for the digest of the concatenation of the strings fed to it so far using the `digest()` or `hexdigest()` methods.

`hash.update(arg)`

Update the hash object with the string `arg`. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

`hash.digest()`

Return the digest of the strings passed to the `update()` method so far. This is a string of `digest_size` bytes which may contain non-ASCII characters, including null bytes.

`hash.hexdigest()`

Like `digest()` except the digest is returned as a string of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

Here is an example:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbdb\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.hexdigest()
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
>>> m.digest_size
16
```

```
>>> m.block_size  
64
```

或者：

```
hashlib.md5("Nobody inspects the spammish repetition").hexdigest()  
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
```

## 第168.2节：OpenSSL提供的算法

还存在一个通用的new()构造函数，它以所需算法的字符串名称作为第一个参数，允许访问上述列出的哈希算法以及您的OpenSSL库可能提供的任何其他算法。命名构造函数比new()快得多，应该优先使用。

使用OpenSSL提供的算法调用new()：

```
>>> h = hashlib.new('ripemd160')  
>>> h.update("Nobody inspects the spammish repetition")  
>>> h.hexdigest()  
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

```
>>> m.block_size  
64
```

or:

```
hashlib.md5("Nobody inspects the spammish repetition").hexdigest()  
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
```

## Section 168.2: algorithm provided by OpenSSL

A generic new() constructor that takes the string name of the desired algorithm as its first parameter also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than new() and should be preferred.

Using new() with an algorithm provided by OpenSSL:

```
>>> h = hashlib.new('ripemd160')  
>>> h.update("Nobody inspects the spammish repetition")  
>>> h.hexdigest()  
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

# 第169章：使用Python创建Windows服务

Windows中无界面（无UI）的进程称为服务。它们可以通过标准的Windows控制方式进行管理（启动、停止等），例如命令控制台、PowerShell或任务管理器中的服务标签页。一个典型的例子可能是提供网络服务的应用程序，如Web应用，或者执行各种后台归档任务的备份应用程序。在Windows中，有多种方法可以将Python应用程序创建并安装为服务。

## 169.1节：可以作为服务运行的Python脚本

本示例中使用的模块属于pywin32（Python for Windows扩展）。根据你安装Python的方式，可能需要单独安装该模块。

```
import win32serviceutil
import win32service
import win32event
import servicemanager
import socket

class AppServerSvc (win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "测试服务"

    def __init__(self,args):
        win32serviceutil.ServiceFramework.__init__(self,args)
        self.hWaitStop = win32event.CreateEvent(None,0,0,None)
        socket.setdefaulttimeout(60)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        win32event.SetEvent(self.hWaitStop)

    def SvcDoRun(self):
        servicemanager.LogMsg(servicemanager.EVENTLOG_INFORMATION_TYPE,
                              servicemanager.PYS_SERVICE_STARTED,
                              (_(self._svc_name_), ''))
        self.main()

    def main(self):
        通过

if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(AppServerSvc)
```

这只是模板代码。您的应用程序代码，可能调用一个独立的脚本，应放在 `main()` 函数中。

您还需要将其安装为服务。目前最好的解决方案似乎是使用Non-吸血鬼服务管理器（Non-sucking Service Manager）。它允许您安装服务，并提供一个用于配置服务执行命令行的图形界面。对于 Python，您可以这样做，一次性创建服务：

```
nssm install MyServiceName c:\python27\python.exe c:\emp\myscript.py
```

其中 `my_script.py` 是上面的模板脚本，修改为在 `main()` 函数中调用您的应用程序脚本或代码。注意，服务并不是直接运行 Python 脚本，而是运行 Python 解释器并传递给它

# Chapter 169: Creating a Windows service using Python

Headless processes (with no UI) in Windows are called Services. They can be controlled (started, stopped, etc) using standard Windows controls such as the command console, PowerShell or the Services tab in Task Manager. A good example might be an application that provides network services, such as a web application, or maybe a backup application that performs various background archival tasks. There are several ways to create and install a Python application as a Service in Windows.

## Section 169.1: A Python script that can be run as a service

The modules used in this example are part of [pywin32](#) (Python for Windows extensions). Depending on how you installed Python, you might need to install this separately.

```
import win32serviceutil
import win32service
import win32event
import servicemanager
import socket

class AppServerSvc (win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"

    def __init__(self,args):
        win32serviceutil.ServiceFramework.__init__(self,args)
        self.hWaitStop = win32event.CreateEvent(None,0,0,None)
        socket.setdefaulttimeout(60)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        win32event.SetEvent(self.hWaitStop)

    def SvcDoRun(self):
        servicemanager.LogMsg(servicemanager.EVENTLOG_INFORMATION_TYPE,
                              servicemanager.PYS_SERVICE_STARTED,
                              (_(self._svc_name_), ''))
        self.main()

    def main(self):
        pass

if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(AppServerSvc)
```

This is just boilerplate. Your application code, probably invoking a separate script, would go in the `main()` function.

You will also need to install this as a service. The best solution for this at the moment appears to be to use [Non-sucking Service Manager](#). This allows you to install a service and provides a GUI for configuring the command line the service executes. For Python you can do this, which creates the service in one go:

```
nssm install MyServiceName c:\python27\python.exe c:\temp\myscript.py
```

Where `my_script.py` is the boilerplate script above, modified to invoke your application script or code in the `main()` function. Note that the service doesn't run the Python script directly, it runs the Python interpreter and passes it the

命令行上的主脚本。

或者，您可以使用适用于您的操作系统版本的 Windows Server 资源工具包中提供的工具来创建服务。

## 第169.2节：将 Flask Web 应用作为服务运行

这是通用示例的一个变体。您只需导入您的应用脚本并在服务的main()函数中调用它的 run()方法。在这种情况下，由于访问 WSGIRequestHandler 的问题。

```
import win32serviceutil
import win32service
import win32event
import servicemanager
from multiprocessing import Process

from app import app

class Service(win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"
    _svc_description_ = "通过接收并回显通过命名管道传输的消息来测试Python服务框架"

    def __init__(self, *args):
        super().__init__(*args)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        self.process.terminate()
        self.ReportServiceStatus(win32service.SERVICE_STOPPED)

    def SvcDoRun(self):
        self.process = Process(target=self.main)
        self.process.start()
        self.process.run()

    def main(self):
        app.run()

if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(Service)
```

Adapted from <http://stackoverflow.com/a/25130524/318488>

main script on the command line.

Alternatively you can use tools provided in the Windows Server Resource Kit for your operating system version so create the service.

## Section 169.2: Running a Flask web application as a service

This is a variation on the generic example. You just need to import your app script and invoke its run() method in the service's main() function. In this case we're also using the multiprocessing module due to an issue accessing WSGIRequestHandler.

```
import win32serviceutil
import win32service
import win32event
import servicemanager
from multiprocessing import Process

from app import app

class Service(win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"
    _svc_description_ = "Tests Python service framework by receiving and echoing messages over a named pipe"

    def __init__(self, *args):
        super().__init__(*args)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        self.process.terminate()
        self.ReportServiceStatus(win32service.SERVICE_STOPPED)

    def SvcDoRun(self):
        self.process = Process(target=self.main)
        self.process.start()
        self.process.run()

    def main(self):
        app.run()

if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(Service)
```

Adapted from <http://stackoverflow.com/a/25130524/318488>

# 视频：Python数据科学与机器学习训练营

学习如何使用NumPy、Pandas、Seaborn、Matplotlib、Plotly、Scikit-Learn、机器学习、Tensorflow等！



- ✓ 使用Python进行数据科学和机器学习
- ✓ 使用Spark进行大数据分析
- ✓ 实现机器学习算法
- ✓ 学习使用NumPy进行数值数据处理
- ✓ 学习使用Pandas进行数据分析
- ✓ 学习使用Matplotlib进行Python绘图
- ✓ 学习使用Seaborn进行统计图表绘制
- ✓ 使用Plotly进行交互式动态可视化
- ✓ 使用SciKit-Learn完成机器学习任务
- ✓ K均值聚类
- ✓ 逻辑回归
- ✓ 线性回归
- ✓ 随机森林和决策树
- ✓ 神经网络
- ✓ 支持向量机今天观看→

# VIDEO: Python for Data Science and Machine Learning Bootcamp

Learn how to use NumPy, Pandas, Seaborn, Matplotlib , Plotly, Scikit-Learn , Machine Learning, Tensorflow, and more!



- ✓ Use Python for Data Science and Machine Learning
- ✓ Use Spark for Big Data Analysis
- ✓ Implement Machine Learning Algorithms
- ✓ Learn to use NumPy for Numerical Data
- ✓ Learn to use Pandas for Data Analysis
- ✓ Learn to use Matplotlib for Python Plotting
- ✓ Learn to use Seaborn for statistical plots
- ✓ Use Plotly for interactive dynamic visualizations
- ✓ Use SciKit-Learn for Machine Learning Tasks
- ✓ K-Means Clustering
- ✓ Logistic Regression
- ✓ Linear Regression
- ✓ Random Forest and Decision Trees
- ✓ Neural Networks
- ✓ Support Vector Machines

Watch Today →

# 第170章：Python中的可变与不可变（以及可哈希）

## 第170.1节：可变与不可变

Python中有两种类型。不可变类型和可变类型。

### 不可变类型

不可变类型的对象无法被更改。任何修改对象的尝试都会导致创建一个副本。

这一类别包括：整数、浮点数、复数、字符串、字节、元组、范围和冻结集合（frozenset）。

为了突出这一特性，我们来玩一下内置函数id。该函数返回作为参数传入对象的唯一标识符。如果id相同，则表示是同一个对象。如果发生变化，则表示是另一个对象。（有人说这实际上是对象的内存地址，但要小心他们，那是黑暗势力的说法……）

```
>>> a = 1
>>> id(a)
140128142243264
>>> a += 2
>>> a
3
>>> id(a)
140128142243328
```

好吧，1不是3……爆炸性新闻……也许不是。然而，当涉及更复杂的类型，尤其是字符串时，这种行为常常被遗忘。

```
>>> stack = "Overflow"
>>> stack
'Overflow'
>>> id(stack)
14012812395504
>>> stack += " rocks!"
>>> stack
'Overflow rocks!'
```

啊哈！看到了吗？我们可以修改它！

```
>>> id(stack)
140128123911472
```

不。虽然看起来我们可以改变变量名为stack的字符串，但实际上我们所做的是创建一个新的对象来保存连接的结果。我们被误导了，因为在这个过程中，旧对象没有被引用，因此它被销毁了。在另一种情况下，这一点会更明显：

```
>>> stack = "Stack"
>>> stackoverflow = stack + "Overflow"
>>> id(stack)
140128069348184
>>> id(stackoverflow)
140128123911480
```

# Chapter 170: Mutable vs Immutable (and Hashable) in Python

## Section 170.1: Mutable vs Immutable

There are two kind of types in Python. Immutable types and mutable types.

### Immutables

An object of an immutable type cannot be changed. Any attempt to modify the object will result in a copy being created.

This category includes: integers, floats, complex, strings, bytes, tuples, ranges and frozensets.

To highlight this property, let's play with the id builtin. This function returns the unique identifier of the object passed as parameter. If the id is the same, this is the same object. If it changes, then this is another object. (*Some say that this is actually the memory address of the object, but beware of them, they are from the dark side of the force...*)

```
>>> a = 1
>>> id(a)
140128142243264
>>> a += 2
>>> a
3
>>> id(a)
140128142243328
```

Okay, 1 is not 3... Breaking news... Maybe not. However, this behaviour is often forgotten when it comes to more complex types, especially strings.

```
>>> stack = "Overflow"
>>> stack
'Overflow'
>>> id(stack)
14012812395504
>>> stack += " rocks!"
>>> stack
'Overflow rocks!'
```

Aha! See? We can modify it!

```
>>> id(stack)
140128123911472
```

No. While it seems we can change the string named by the variable stack, what we actually do, is creating a new object to contain the result of the concatenation. We are fooled because in the process, the old object goes nowhere, so it is destroyed. In another situation, that would have been more obvious:

```
>>> stack = "Stack"
>>> stackoverflow = stack + "Overflow"
>>> id(stack)
140128069348184
>>> id(stackoverflow)
140128123911480
```

在这种情况下，很明显如果我们想保留第一个字符串，就需要一个副本。但对于其他类型来说，这一点是否同样明显呢？

## 练习

现在，了解了不可变类型的工作原理后，你如何看待下面这段代码？这样做明智吗？

```
s = ""  
for i in range(1, 1000):  
    s += str(i)  
    s += ", "
```

## 可变对象

可变类型的对象可以被修改，且是在原地（in-situ）修改的。不会进行隐式复制。

此类别包括：列表、字典、字节数组和集合。

让我们继续玩弄我们的小id函数。

```
>>> b = bytearray(b'Stack')  
>>> b  
bytearray(b'Stack')  
>>> b = bytearray(b'Stack')  
>>> id(b)  
140128030688288  
>>> b += b'Overflow'  
>>> b  
bytearray(b'StackOverflow')  
>>> id(b)  
140128030688288
```

（顺便说一句，我使用包含ascii数据的字节来说明我的观点，但请记住，字节并不是设计用来存储文本数据的。愿原力宽恕我。）

我们得到了什么？我们创建了一个bytearray，修改它，并使用id，

我们可以确保这是同一个被修改的对象，而不是它的副本。

当然，如果一个对象将被频繁修改，可变类型比不可变类型做得更好。

不幸的是，当这种属性最需要被记住时，现实往往被遗忘。

```
>>> c = b  
>>> c += b' rocks!'  
>>> c  
bytearray(b'StackOverflow rocks!')
```

好的...

```
>>> b  
bytearray(b'StackOverflow rocks!')
```

等一下.....

```
>>> id(c) == id(b)  
True
```

确实。c 不是 b 的副本。c 就是 b。

## 练习

In this case it is clear that if we want to retain the first string, we need a copy. But is that so obvious for other types?

## Exercise

Now, knowing how immutable types work, what would you say with the below piece of code? Is it wise?

```
s = ""  
for i in range(1, 1000):  
    s += str(i)  
    s += ", "
```

## Mutables

An object of a mutable type can be changed, and it is changed *in-situ*. No implicit copies are done.

This category includes: lists, dictionaries, bytearrays and sets.

Let's continue to play with our little id function.

```
>>> b = bytearray(b'Stack')  
>>> b  
bytearray(b'Stack')  
>>> b = bytearray(b'Stack')  
>>> id(b)  
140128030688288  
>>> b += b'Overflow'  
>>> b  
bytearray(b'StackOverflow')  
>>> id(b)  
140128030688288
```

（As a side note, I use bytes containing ascii data to make my point clear, but remember that bytes are not designed to hold textual data. May the force pardon me.）

What do we have? We create a bytearray, modify it and using the id, we can ensure that this is the same object, modified. Not a copy of it.

Of course, if an object is going to be modified often, a mutable type does a much better job than an immutable type. Unfortunately, the reality of this property is often forgotten when it hurts the most.

```
>>> c = b  
>>> c += b' rocks!'  
>>> c  
bytearray(b'StackOverflow rocks!')
```

Okay...

```
>>> b  
bytearray(b'StackOverflow rocks!')
```

Waiit a second...

```
>>> id(c) == id(b)  
True
```

Indeed. c is not a copy of b. c is b.

## Exercise

现在你更好地理解了可变类型所隐含的副作用，你能解释一下这个例子中出了什么问题吗？

```
>>> ll = [ ]*4 # 创建一个包含4个列表的列表来存放我们的结果
>>> ll
[[], [], [], []]
>>> ll[0].append(23) # 向第一个列表添加结果23
>>> ll
[[23], [23], [23], [23]]
>>> # 啊呀.....
```

## 第170.2节：作为参数的可变与不可变类型

当开发者需要考虑可变性时，一个主要的使用场景是在向函数传递参数时。这一点非常重要，因为它决定了函数是否能够修改不属于其作用域的对象，换句话说，函数是否具有副作用。这对于理解函数的结果需要在哪里可用也很重要。

```
>>> def list_add3(lin):
lin += [3]
    return lin

>>> a = [1, 2, 3]
>>> b = list_add3(a)
>>> b
[1, 2, 3, 3]
>>> a
[1, 2, 3, 3]
```

这里的错误是认为作为函数参数的lin可以在局部被修改。实际上，lin和a引用的是同一个对象。由于该对象是可变的，修改是在原地进行的，这意味着lin和a引用的对象都被修改了。实际上lin不需要返回，因为我们已经通过a持有该对象的引用。a和b最终引用的是同一个对象。

元组则不一样。

```
>>> def tuple_add3(tin):
tin += (3,)
    return tin

>>> a = (1, 2, 3)
>>> b = tuple_add3(a)
>>> b
(1, 2, 3, 3)
>>> a
(1, 2, 3)
```

在函数开始时，tin 和 a 引用的是同一个对象。但这是一个不可变对象。因此当函数尝试修改它时，tin 会接收一个带有修改的新对象，而 a 仍然保持对原始对象的引用。在这种情况下，必须返回 tin，否则新对象将会丢失。

### 练习

```
>>> def yoda(prologue, sentence):
sentence.reverse()
    prologue += " ".join(sentence)
return prologue
```

Now you better understand what side effect is implied by a mutable type, can you explain what is going wrong in this example?

```
>>> ll = [ ]*4 # Create a list of 4 lists to contain our results
>>> ll
[[], [], [], []]
>>> ll[0].append(23) # Add result 23 to first list
>>> ll
[[23], [23], [23], [23]]
>>> # Oops...
```

## Section 170.2: Mutable and Immutable as Arguments

One of the major use case when a developer needs to take mutability into account is when passing arguments to a function. This is very important, because this will determine the ability for the function to modify objects that doesn't belong to its scope, or in other words if the function has side effects. This is also important to understand where the result of a function has to be made available.

```
>>> def list_add3(lin):
lin += [3]
    return lin

>>> a = [1, 2, 3]
>>> b = list_add3(a)
>>> b
[1, 2, 3, 3]
>>> a
[1, 2, 3, 3]
```

Here, the mistake is to think that lin, as a parameter to the function, can be modified locally. Instead, lin and a reference the same object. As this object is mutable, the modification is done in-place, which means that the object referenced by both lin and a is modified. lin doesn't really need to be returned, because we already have a reference to this object in the form of a. a and b end referencing the same object.

This doesn't go the same for tuples.

```
>>> def tuple_add3(tin):
tin += (3,)
    return tin

>>> a = (1, 2, 3)
>>> b = tuple_add3(a)
>>> b
(1, 2, 3, 3)
>>> a
(1, 2, 3)
```

At the beginning of the function, tin and a reference the same object. But this is an immutable object. So when the function tries to modify it, tin receive a new object with the modification, while a keeps a reference to the original object. In this case, returning tin is mandatory, or the new object would be lost.

### Exercise

```
>>> def yoda(prologue, sentence):
sentence.reverse()
    prologue += " ".join(sentence)
return prologue
```

```
>>> focused = ["你必须", "保持专注"]
>>> saying = "尤达说："
>>> yoda_sentence = yoda(saying, focused)
```

注意：`reverse` 是原地操作。

你怎么看这个函数？它有副作用吗？返回值有必要吗？调用后，  
`saying` 的值是多少？`focused` 的值是多少？如果用相同的参数再次调用该函数，会发生什么？

```
>>> focused = ["You must", "stay focused"]
>>> saying = "Yoda said: "
>>> yoda_sentence = yoda(saying, focused)
```

*Note: reverse operates in-place.*

What do you think of this function? Does it have side effects? Is the return necessary? After the call, what is the value of `saying`? Of `focused`? What happens if the function is called again with the same parameters?

# 第171章： configparser

该模块提供了 ConfigParser 类，实现了 INI 文件中的基本配置语言。你可以使用它编写易于终端用户自定义的 Python 程序。

## 第171.1节：以编程方式创建配置文件

配置文件包含多个节，每个节包含键和值。configparser 模块可用于读取和写入配置文件。创建配置文件：

```
import configparser
config = configparser.ConfigParser()
config['settings']={'resolution':'320x240',
                    'color':'blue'}
with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

输出文件包含以下结构

```
[settings]
resolution = 320x240
color = blue
```

如果你想更改特定字段，获取该字段并赋值

```
settings=config['settings']
settings['color']='red'
```

## 第171.2节：基本用法

在 config.ini 中：

```
[DEFAULT]
debug = True
name = Test
password = password

[FILES]
path = /path/to/file
```

在 Python 中：

```
from ConfigParser import ConfigParser
config = ConfigParser()

# 加载配置文件
config.read("config.ini")

# 访问 "DEFAULT" 部分中的键 "debug"
config.get("DEFAULT", "debug")
# 返回 'True'

# 访问 "FILES" 部分中的键 "path"
config.get("FILES", "path")
# 返回 '/path/to/file'
```

# Chapter 171: configparser

This module provides the ConfigParser class which implements a basic configuration language in INI files. You can use this to write Python programs which can be customized by end users easily.

## Section 171.1: Creating configuration file programmatically

Configuration file contains sections, each section contains keys and values. configparser module can be used to read and write config files. Creating the configuration file:

```
import configparser
config = configparser.ConfigParser()
config['settings']={'resolution':'320x240',
                    'color':'blue'}
with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

The output file contains below structure

```
[settings]
resolution = 320x240
color = blue
```

If you want to change particular field ,get the field and assign the value

```
settings=config['settings']
settings['color']='red'
```

## Section 171.2: Basic usage

In config.ini:

```
[DEFAULT]
debug = True
name = Test
password = password

[FILES]
path = /path/to/file
```

In Python:

```
from ConfigParser import ConfigParser
config = ConfigParser()

# Load configuration file
config.read("config.ini")

# Access the key "debug" in "DEFAULT" section
config.get("DEFAULT", "debug")
# Return 'True'

# Access the key "path" in "FILES" section
config.get("FILES", "path")
# Return '/path/to/file'
```

# 第172章：光学字符识别

光学字符识别是将文本图像转换为实际文本。在这些示例中，寻找在python中使用OCR的方法。

## 第172.1节：PyTesseract

PyTesseract是一个正在开发中的python OCR包。

使用PyTesseract非常简单：

```
尝试:  
    import Image  
except ImportError:  
    from PIL import Image  
  
import pytesseract  
  
#基础OCR  
print(pytesseract.image_to_string(Image.open('test.png')))  
  
#法语  
print(pytesseract.image_to_string(Image.open('test-european.jpg'), lang='fra'))
```

PyTesseract 是开源的，可以在 [here](#) 找到。

## 第172.2节：PyOCR

另一个有用的模块是 PyOCR，其源代码在 [here](#)。

使用也很简单，且功能比 PyTesseract 更多。

初始化方法：

```
from PIL import Image  
import sys  
  
import pyocr  
import pyocr.builders  
  
tools = pyocr.get_available_tools()  
# 这些工具按推荐的使用顺序返回  
tool = tools[0]  
  
langs = tool.get_available_languages()  
lang = langs[0]  
# 注意，语言列表没有经过任何排序。请参考# 系统区域设置以确定默认使用的  
# 语言。
```

以下是一些使用示例：

```
txt = tool.image_to_string(  
    Image.open('test.png'),  
    lang=lang,
```

# Chapter 172: Optical Character Recognition

Optical Character Recognition is converting images of text into actual text. In these examples find ways of using OCR in python.

## Section 172.1: PyTesseract

PyTesseract is an in-development python package for OCR.

Using PyTesseract is pretty easy:

```
try:  
    import Image  
except ImportError:  
    from PIL import Image  
  
import pytesseract  
  
#Basic OCR  
print(pytesseract.image_to_string(Image.open('test.png')))  
  
#In French  
print(pytesseract.image_to_string(Image.open('test-european.jpg'), lang='fra'))
```

PyTesseract is open source and can be found [here](#).

## Section 172.2: PyOCR

Another module of some use is PyOCR, source code of which is [here](#).

Also simple to use and has more features than PyTesseract.

To initialize:

```
from PIL import Image  
import sys  
  
import pyocr  
import pyocr.builders  
  
tools = pyocr.get_available_tools()  
# The tools are returned in the recommended order of usage  
tool = tools[0]  
  
langs = tool.get_available_languages()  
lang = langs[0]  
# Note that languages are NOT sorted in any way. Please refer  
# to the system locale settings for the default language  
# to use.
```

And some examples of usage:

```
txt = tool.image_to_string(  
    Image.open('test.png'),  
    lang=lang,
```

```

builder=pyocr.builders.TextBuilder()
)
# txt 是一个 Python 字符串

word_boxes = tool.image_to_string(
    Image.open('test.png'),
    lang="eng",
    builder=pyocr.builders.WordBoxBuilder()
)
# 盒子对象列表。对于每个盒子对象：
#   box.content 是盒子中的单词
#   box.position 是其在页面上的位置（以像素为单位）
#
# 注意某些 OCR 工具（例如 Tesseract）
# 可能会返回空盒子

line_and_word_boxes = tool.image_to_string(
    Image.open('test.png'), lang="fra",
    builder=pyocr.builders.LineBoxBuilder()
)
# 行对象列表。对于每个行对象：
#   line.word_boxes 是单词盒子列表（该行中的各个单词）
#   line.content 是整行文本内容
#   line.position 是整行在页面上的位置（以像素为单位）
#
# 注意某些 OCR 工具（例如 Tesseract）
# 可能会返回空盒子

# 数字 - 仅限 Tesseract (尚不支持 'libtesseract' !)
digits = tool.image_to_string(
    Image.open('test-digits.png'),
    lang=lang,
    builder=pyocr.tesseract.DigitBuilder()
)
# digits 是一个 Python 字符串

```

```

builder=pyocr.builders.TextBuilder()
)
# txt is a Python string

word_boxes = tool.image_to_string(
    Image.open('test.png'),
    lang="eng",
    builder=pyocr.builders.WordBoxBuilder()
)
# list of box objects. For each box object:
#   box.content is the word in the box
#   box.position is its position on the page (in pixels)
#
# Beware that some OCR tools (Tesseract for instance)
# may return empty boxes

line_and_word_boxes = tool.image_to_string(
    Image.open('test.png'), lang="fra",
    builder=pyocr.builders.LineBoxBuilder()
)
# list of line objects. For each line object:
#   line.word_boxes is a list of word boxes (the individual words in the line)
#   line.content is the whole text of the line
#   line.position is the position of the whole line on the page (in pixels)
#
# Beware that some OCR tools (Tesseract for instance)
# may return empty boxes

# Digits - Only Tesseract (not 'libtesseract' yet !)
digits = tool.image_to_string(
    Image.open('test-digits.png'),
    lang=lang,
    builder=pyocr.tesseract.DigitBuilder()
)
# digits is a python string

```

# 第173章：虚拟环境

虚拟环境是一种工具，通过为不同项目创建虚拟的Python环境，将它们所需的依赖项分别存放在不同的位置。它解决了“项目X依赖于1.x版本，但项目Y需要4.x版本”的难题，并保持你的全局site-packages目录干净且易于管理。

这有助于将不同项目的环境彼此隔离开来，同时也与系统库隔离。

## 第173.1节：创建和使用虚拟环境

virtualenv是一个构建隔离Python环境的工具。该程序会创建一个文件夹，里面包含使用Python项目所需包的所有必要可执行文件。

### 安装virtualenv工具

只需执行一次。virtualenv程序可能通过你的发行版提供。在类似Debian的发行版中，该软件包名为python-virtualenv或python3-virtualenv。

你也可以使用pip安装virtualenv：

```
$ pip install virtualenv
```

### 创建新的虚拟环境

每个项目只需执行一次。当开始一个需要隔离依赖项的项目时，您可以为该项目设置一个新的虚拟环境：

```
$ virtualenv foo
```

这将创建一个foo文件夹，包含工具脚本和python二进制文件的副本。文件夹名称无关紧要。虚拟环境一旦创建，它是自包含的，不需要进一步使用virtualenv工具进行操作。您现在可以开始使用该虚拟环境。

### 激活已有的虚拟环境

要激活一个虚拟环境，需要一些shell技巧，使您的Python版本是foo中的，而不是系统自带的。这个activate文件的作用就是如此，您必须将其source到当前shell中：

```
$ source foo/bin/activate
```

Windows用户应输入：

```
$ foo\Scripts\activate.bat
```

一旦虚拟环境被激活，python和pip二进制文件以及所有由第三方模块安装的脚本，都是foo中的版本。特别是，所有通过pip安装的模块都会部署到虚拟环境中，从而实现一个独立的开发环境。激活虚拟环境后，您的命令提示符前缀也会相应添加，如以下命令所示。

```
# 仅在 foo 中安装 'requests'，而非全局安装  
(foo)$ pip install requests
```

### 保存和恢复依赖项

# Chapter 173: Virtual environments

A Virtual Environment is a tool to keep the dependencies required by different projects in separate places, by creating virtual Python environments for them. It solves the “Project X depends on version 1.x but, Project Y needs 4.x” dilemma, and keeps your global site-packages directory clean and manageable.

This helps isolate your environments for different projects from each other and from your system libraries.

## Section 173.1: Creating and using a virtual environment

virtualenv is a tool to build isolated Python environments. This program creates a folder which contains all the necessary executables to use the packages that a Python project would need.

### Installing the virtualenv tool

This is only required once. The virtualenv program may be available through your distribution. On Debian-like distributions, the package is called python-virtualenv or python3-virtualenv.

You can alternatively install virtualenv using pip:

```
$ pip install virtualenv
```

### Creating a new virtual environment

This only required once per project. When starting a project for which you want to isolate dependencies, you can setup a new virtual environment for this project:

```
$ virtualenv foo
```

This will create a foo folder containing tooling scripts and a copy of the python binary itself. The name of the folder is not relevant. Once the virtual environment is created, it is self-contained and does not require further manipulation with the virtualenv tool. You can now start using the virtual environment.

### Activating an existing virtual environment

To activate a virtual environment, some shell magic is required so your Python is the one inside foo instead of the system one. This is the purpose of the activate file, that you must source into your current shell:

```
$ source foo/bin/activate
```

Windows users should type:

```
$ foo\Scripts\activate.bat
```

Once a virtual environment has been activated, the python and pip binaries and all scripts installed by third party modules are the ones inside foo. Particularly, all modules installed with pip will be deployed to the virtual environment, allowing for a contained development environment. Activating the virtual environment should also add a prefix to your prompt as seen in the following commands.

```
# Installs 'requests' to foo only, not globally  
(foo)$ pip install requests
```

### Saving and restoring dependencies

要保存通过pip安装的模块，可以使用freeze命令将所有这些模块（及对应版本）列入一个文本文件中。这样，其他人就可以通过安装命令快速安装应用所需的Python模块。此类文件的惯用名称是requirements.txt：

```
(foo)$ pip freeze > requirements.txt  
(foo)$ pip install -r requirements.txt
```

请注意，freeze会列出所有模块，包括手动安装的顶级模块所需的传递依赖项。因此，你可能更愿意手动编写requirements.txt文件，只放入你需要的顶级模块。

## 退出虚拟环境

如果你完成了虚拟环境中的工作，可以通过停用它返回到正常的shell：

```
(foo)$ deactivate
```

## 在共享主机中使用虚拟环境

有时无法通过\$ source bin/activate激活虚拟环境，例如当你使用mod\_wsgi在共享主机上，或者你无法访问文件系统，比如在Amazon API Gateway或Google AppEngine中。对于这些情况，你可以部署你在本地虚拟环境中安装的库，并修改你的sys.path。

幸运的是，virtualenv自带一个脚本，可以同时更新你的sys.path和你的sys.prefix

```
import os
```

```
mydir = os.path.dirname(os.path.realpath(__file__))  
activate_this = mydir + '/bin/activate_this.py'  
execfile(activate_this, dict(__file__=activate_this))
```

你应该将这些代码行添加到服务器执行文件的最开头。

这将找到bin/activate\_this.py，该文件是virtualenv在你执行的同一目录下创建的，并将你的lib/python2.7/site-packages添加到sys.path

如果你打算使用activate\_this.py脚本，记得至少部署bin和lib/python2.7/site-packages目录及其内容。

Python 3.x 版本 ≥ 3.3

## 内置虚拟环境

从Python 3.3开始，venv模块将创建虚拟环境。无需单独安装pyvenv命令：

```
$ pyvenv foo  
$ source foo/bin/activate
```

或者

```
$ python3 -m venv foo  
$ source foo/bin/activate
```

To save the modules that you have installed via pip, you can list all of those modules (and the corresponding versions) into a text file by using the freeze command. This allows others to quickly install the Python modules needed for the application by using the install command. The conventional name for such a file is requirements.txt:

```
(foo)$ pip freeze > requirements.txt  
(foo)$ pip install -r requirements.txt
```

Please note that freeze lists all the modules, including the transitive dependencies required by the top-level modules you installed manually. As such, you may prefer to [craft the requirements.txt file by hand](#), by putting only the top-level modules you need.

## Exiting a virtual environment

If you are done working in the virtual environment, you can deactivate it to get back to your normal shell:

```
(foo)$ deactivate
```

## Using a virtual environment in a shared host

Sometimes it's not possible to \$ source bin/activate a virtualenv, for example if you are using mod\_wsgi in shared host or if you don't have access to a file system, like in Amazon API Gateway, or Google AppEngine. For those cases you can deploy the libraries you installed in your local virtualenv and patch your sys.path.

Luckily virtualenv ships with a script that updates both your sys.path and your sys.prefix

```
import os
```

```
mydir = os.path.dirname(os.path.realpath(__file__))  
activate_this = mydir + '/bin/activate_this.py'  
execfile(activate_this, dict(__file__=activate_this))
```

You should append these lines at the very beginning of the file your server will execute.

This will find the bin/activate\_this.py that virtualenv created file in the same dir you are executing and add your lib/python2.7/site-packages to sys.path

If you are looking to use the activate\_this.py script, remember to deploy with, at least, the bin and lib/python2.7/site-packages directories and their content.

Python 3.x Version ≥ 3.3

## Built-in virtual environments

From Python 3.3 onwards, the [venv module](#) will create virtual environments. The pyvenv command does not need installing separately:

```
$ pyvenv foo  
$ source foo/bin/activate
```

or

```
$ python3 -m venv foo  
$ source foo/bin/activate
```

## 第173.2节：在Unix/Linux脚本中指定使用的特定Python版本

为了指定Linux shell应使用哪个版本的Python，Python脚本的第一行可以是一个以#!开头的shebang行：

```
#!/usr/bin/python
```

如果你处于虚拟环境中，那么python myscript.py将使用虚拟环境中的Python，但./myscript.py将使用#!行中指定的Python解释器。为了确保使用虚拟环境的Python，请将第一行改为：

```
#!/usr/bin/env python
```

指定shebang行后，记得通过以下命令赋予脚本执行权限：

```
chmod +x myscript.py
```

这样你就可以通过运行./myscript.py（或提供脚本的绝对路径）来执行脚本，而不必使用python myscript.py或python3 myscript.py。

## 第173.3节：为不同版本的Python创建虚拟环境

假设python和python3都已安装，即使python3不是默认的Python，也可以为Python 3创建虚拟环境：

```
virtualenv -p python3 foo
```

或者

```
virtualenv --python=python3 foo
```

或者

```
python3 -m venv foo
```

或者

```
pyvenv foo
```

实际上，你可以基于系统中任何可用的 Python 版本创建虚拟环境。你可以在 Linux 系统的 /usr/bin/ 或 /usr/local/bin/ 目录下检查不同的可用 Python 版本，或者在 /Library/Frameworks/Python.framework/Versions/X.X/bin/ (macOS) 中，然后确定名称，并在创建虚拟环境时使用 --python 或 -p 参数指定该版本。

## 第173.4节：使用 Anaconda 创建虚拟环境

一个强大的 virtualenv 替代方案是 [Anaconda](#) —— 一个跨平台的、类似 pip 的包管理器，内置了快速创建和删除虚拟环境的功能。安装 Anaconda 后，可以使用以下命令开始：

## Section 173.2: Specifying specific python version to use in script on Unix/Linux

In order to specify which version of python the Linux shell should use the first line of Python scripts can be a shebang line, which starts with #!:

```
#!/usr/bin/python
```

If you are in a virtual environment, then python myscript.py will use the Python from your virtual environment, but ./myscript.py will use the Python interpreter in the #! line. To make sure the virtual environment's Python is used, change the first line to:

```
#!/usr/bin/env python
```

After specifying the shebang line, remember to give execute permissions to the script by doing:

```
chmod +x myscript.py
```

Doing this will allow you to execute the script by running ./myscript.py (or provide the absolute path to the script) instead of python myscript.py or python3 myscript.py.

## Section 173.3: Creating a virtual environment for a different version of python

Assuming python and python3 are both installed, it is possible to create a virtual environment for Python 3 even if python3 is not the default Python:

```
virtualenv -p python3 foo
```

或者

```
virtualenv --python=python3 foo
```

或者

```
python3 -m venv foo
```

或者

```
pyvenv foo
```

Actually you can create virtual environment based on any version of working python of your system. You can check different working python under your /usr/bin/ or /usr/local/bin/ (In Linux) OR in /Library/Frameworks/Python.framework/Versions/X.X/bin/ (OSX), then figure out the name and use that in the --python or -p flag while creating virtual environment.

## Section 173.4: Making virtual environments using Anaconda

A powerful alternative to virtualenv is [Anaconda](#) - a cross-platform, pip-like package manager bundled with features for quickly making and removing virtual environments. After installing Anaconda, here are some commands to get started:

## 创建环境

```
conda create --name <envname> python=<version>
```

其中 `<envname>` 是你虚拟环境的任意名称，`<version>` 是你希望设置的特定 Python 版本。

## 激活和停用你的环境

```
# Linux, Mac  
source activate <envname>  
source deactivate
```

或者

```
# Windows  
activate <envname>  
deactivate
```

## 查看已创建环境列表

```
conda env list
```

## 删除环境

```
conda env remove -n <envname>
```

在官方conda文档中查找更多命令和功能。

## 第173.5节：使用virtualenvwrapper管理多个虚拟环境

[virtualenvwrapper](#) 工具简化了虚拟环境的使用，尤其适合处理多个虚拟环境/项目的情况。

与其自己管理虚拟环境目录，不如让virtualenvwrapper为你管理，通过将所有虚拟环境存储在一个中央目录（默认是`~/virtualenvs`）。

### 安装

使用系统的包管理器安装virtualenvwrapper。

基于Debian/Ubuntu的系统：

```
apt-get install virtualenvwrapper
```

Fedora/CentOS/RHEL：

```
yum install python-virtualenvwrapper
```

Arch Linux：

```
pacman -S python-virtualenvwrapper
```

或者使用pip从PyPI安装：

```
pip install virtualenvwrapper
```

在Windows下，你可以使用[virtualenvwrapper-win](#)或[virtualenvwrapper-powershell](#)代替。

## Create an environment

```
conda create --name <envname> python=<version>
```

where `<envname>` is an arbitrary name for your virtual environment, and `<version>` is a specific Python version you wish to setup.

## Activate and deactivate your environment

```
# Linux, Mac  
source activate <envname>  
source deactivate
```

or

```
# Windows  
activate <envname>  
deactivate
```

## View a list of created environments

```
conda env list
```

## Remove an environment

```
conda env remove -n <envname>
```

Find more commands and features in the official [conda documentation](#).

## Section 173.5: Managing multiple virtual environments with virtualenvwrapper

The [virtualenvwrapper](#) utility simplifies working with virtual environments and is especially useful if you are dealing with many virtual environments/projects.

Instead of having to deal with the virtual environment directories yourself, virtualenvwrapper manages them for you, by storing all virtual environments under a central directory (`~/virtualenvs` by default).

### Installation

Install virtualenvwrapper with your system's package manager.

Debian/Ubuntu-based:

```
apt-get install virtualenvwrapper
```

Fedora/CentOS/RHEL:

```
yum install python-virtualenvwrapper
```

Arch Linux:

```
pacman -S python-virtualenvwrapper
```

Or install it from PyPI using pip:

```
pip install virtualenvwrapper
```

Under Windows you can use either [virtualenvwrapper-win](#) or [virtualenvwrapper-powershell](#) instead.

## 用法

虚拟环境通过mkvirtualenv创建。原始virtualenv命令的所有参数也被接受。

```
mkvirtualenv my-project
```

或者例如

```
mkvirtualenv --system-site-packages my-project
```

新的虚拟环境已自动激活。在新的 shell 中，你可以使用以下命令启用虚拟环境  
workon

```
workon my-project
```

与传统的.path/to/my-env/bin/activate命令相比，workon命令的优势在于，workon命令可以在任何目录下使用；你不必记住项目的特定虚拟环境存放在哪个目录。

## 项目目录

你甚至可以在创建虚拟环境时使用-a选项指定项目目录，或者稍后使用setvirtualenvproject命令指定。

```
mkvirtualenv -a /path/to/my-project my-project
```

或者

```
workon my-project  
cd /path/to/my-project  
setvirtualenvproject
```

设置项目后，workon命令将自动切换到该项目，并启用cdproject命令，允许你切换到项目目录。

要查看 virtualenvwrapper 管理的所有虚拟环境列表，请使用lsvirtualenv命令。

要删除一个虚拟环境，使用 rmvirtualenv：

```
rmvirtualenv my-project
```

每个由 virtualenvwrapper 管理的 virtualenv 都包含 4 个空的 bash 脚本：preactivate、postactivate、predeactivate 和 postdeactivate。这些脚本作为钩子，用于在 virtualenv 生命周期的某些阶段执行 bash 命令；例如，postactivate 脚本中的任何命令都会在 virtualenv 激活后立即执行。这是设置特殊环境变量、别名或其他相关内容的好地方。所有这 4 个脚本都位于 .virtualenvs/<virtualenv\_name>/bin/ 目录下。

更多详情请阅读 [virtualenvwrapper 文档](#)。

## 第 173.6 节：在虚拟环境中安装包

一旦激活了虚拟环境，您安装的任何包都会被安装在 virtualenv 中，而不是全局环境中。因此，安装新包时无需 root 权限。

## Usage

Virtual environments are created with mkvirtualenv. All arguments of the original virtualenv command are accepted as well.

```
mkvirtualenv my-project
```

or e.g.

```
mkvirtualenv --system-site-packages my-project
```

The new virtual environment is automatically activated. In new shells you can enable the virtual environment with workon

```
workon my-project
```

The advantage of the workon command compared to the traditional .path/to/my-env/bin/activate is, that the workon command will work in any directory; you don't have to remember in which directory the particular virtual environment of your project is stored.

## Project Directories

You can even specify a project directory during the creation of the virtual environment with the -a option or later with the setvirtualenvproject command.

```
mkvirtualenv -a /path/to/my-project my-project
```

or

```
workon my-project  
cd /path/to/my-project  
setvirtualenvproject
```

Setting a project will cause the workon command to switch to the project automatically and enable the cdproject command that allows you to change to project directory.

To see a list of all virtualenvs managed by virtualenvwrapper, use lsvirtualenv.

To remove a virtualenv, use rmvirtualenv:

```
rmvirtualenv my-project
```

Each virtualenv managed by virtualenvwrapper includes 4 empty bash scripts: preactivate, postactivate, predeactivate, and postdeactivate. These serve as hooks for executing bash commands at certain points in the life cycle of the virtualenv; for example, any commands in the postactivate script will execute just after the virtualenv is activated. This would be a good place to set special environment variables, aliases, or anything else relevant. All 4 scripts are located under .virtualenvs/<virtualenv\_name>/bin/.

For more details read the [virtualenvwrapper documentation](#).

## Section 173.6: Installing packages in a virtual environment

Once your virtual environment has been activated, any package that you install will now be installed in the virtualenv & not globally. Hence, new packages can be without needing root privileges.

要验证包是否安装在 virtualenv 中，请运行以下命令检查所使用的可执行文件路径：

```
(<Virtualenv 名称>) $ which python  
/<Virtualenv 目录>/bin/python
```

```
(Virtualenv 名称) $ which pip  
/<Virtualenv 目录>/bin/pip
```

然后，使用 pip 安装的任何包都会安装在 virtualenv 本身的以下目录中：

```
/<虚拟环境目录>/lib/python2.7/site-packages/
```

或者，您可以创建一个列出所需包的文件。

#### requirements.txt:

```
requests==2.10.0
```

执行：

```
# 从 requirements.txt 安装包  
pip install -r requirements.txt
```

将安装版本为 2.10.0 的包 requests。

您也可以获取当前激活虚拟环境中已安装包及其版本的列表：

```
# 获取已安装包列表  
pip freeze  
  
# 将软件包及其版本列表输出到 requirements.txt 文件，以便您可以重新创建虚拟环境  
pip freeze > requirements.txt
```

或者，您不必每次安装软件包时都激活虚拟环境。您可以直接使用虚拟环境目录中的 pip 可执行文件来安装软件包。

```
$ /<虚拟环境目录>/bin/pip install requests
```

有关使用 pip 的更多信息，请参见 PIP 主题。

由于您是在虚拟环境中以非 root 用户身份安装，这不是全局安装，不会影响整个系统——安装的软件包仅在当前虚拟环境中可用。

## 第 173.7 节：发现您正在使用的虚拟环境

如果您使用的是 Linux 上的默认 bash 提示符，您应该会在提示符开头看到虚拟环境的名称。

```
(my-project-env) user@hostname:~$ which python  
/home/user/my-project-env/bin/python
```

To verify that the packages are being installed into the virtualenv run the following command to check the path of the executable that is being used:

```
(<Virtualenv Name>) $ which python  
/<Virtualenv Directory>/bin/python
```

```
(Virtualenv Name) $ which pip  
/<Virtualenv Directory>/bin/pip
```

Any package then installed using pip will be installed in the virtualenv itself in the following directory:

```
/<Virtualenv Directory>/lib/python2.7/site-packages/
```

Alternatively, you may create a file listing the needed packages.

#### requirements.txt:

```
requests==2.10.0
```

Executing:

```
# Install packages from requirements.txt  
pip install -r requirements.txt
```

will install version 2.10.0 of the package requests.

You can also get a list of the packages and their versions currently installed in the active virtual environment:

```
# Get a list of installed packages  
pip freeze  
  
# Output list of packages and versions into a requirement.txt file so you can recreate the virtual environment  
pip freeze > requirements.txt
```

Alternatively, you do not have to activate your virtual environment each time you have to install a package. You can directly use the pip executable in the virtual environment directory to install packages.

```
$ /<Virtualenv Directory>/bin/pip install requests
```

More information about using pip can be found on the PIP topic.

Since you're installing without root in a virtual environment, this is *not* a global install, across the entire system - the installed package will only be available in the current virtual environment.

## Section 173.7: Discovering which virtual environment you are using

If you are using the default bash prompt on Linux, you should see the name of the virtual environment at the start of your prompt.

```
(my-project-env) user@hostname:~$ which python  
/home/user/my-project-env/bin/python
```

## 第173.8节：检查是否在虚拟环境中运行

有时shell提示符不会显示虚拟环境的名称，而你想确认自己是否处于虚拟环境中。

运行Python解释器并尝试：

```
import sys  
sys.prefix  
sys.real_prefix
```

- 在虚拟环境外，`sys.prefix` 会指向系统的Python安装路径，且 `sys.real_prefix` 未定义。
- 在虚拟环境内，`sys.prefix` 会指向虚拟环境的Python安装路径，`sys.real_prefix` 会指向系统的Python安装路径。

对于使用标准库中的 `venv` 模块 创建的虚拟环境，没有 `sys.real_prefix`。取而代之的是，检查 `sys.base_prefix` 是否与 `sys.prefix` 相同。

## 第173.9节：在fish shell中使用virtualenv

Fish shell更友好，但在使用 `virtualenv` 或 `virtualenvwrapper` 时可能会遇到问题。作为替代，`virtualfish` 可以帮你解决。只需按照以下步骤开始在Fish shell中使用virtualenv。

- 将`virtualfish`安装到全局环境

```
sudo pip install virtualfish
```

- 在 fish shell 启动时加载 python 模块 `virtualfish`

```
$ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
```

- 通过 `$ funcdef fish_prompt --editor vim` 编辑此函数 `fish_prompt`，并添加以下内容后关闭 vim 编辑器

```
if set -q VIRTUAL_ENV  
echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color  
normal) " "  
end
```

注意：如果你不熟悉 vim，可以直接使用你喜欢的编辑器，例如 `$ funcdef fish_prompt --editor nano` 或 `$ funcdef fish_prompt --editor gedit`

- 使用 `funcsave` 保存更改

```
funcsave fish_prompt
```

- 要创建新的虚拟环境，请使用 `vf new`

```
vf new my_new_env # 确保 $HOME/.virtualenv 存在
```

## Section 173.8: Checking if running inside a virtual environment

Sometimes the shell prompt doesn't display the name of the virtual environment and you want to be sure if you are in a virtual environment or not.

Run the python interpreter and try:

```
import sys  
sys.prefix  
sys.real_prefix
```

- Outside a virtual environment `sys.prefix` will point to the system python installation and `sys.real_prefix` is not defined.
- Inside a virtual environment, `sys.prefix` will point to the virtual environment python installation and `sys.real_prefix` will point to the system python installation.

For virtual environments created using the standard library `venv module` there is no `sys.real_prefix`. Instead, check whether `sys.base_prefix` is the same as `sys.prefix`.

## Section 173.9: Using virtualenv with fish shell

Fish shell is friendlier yet you might face trouble while using with `virtualenv` or `virtualenvwrapper`. Alternatively `virtualfish` exists for the rescue. Just follow the below sequence to start using Fish shell with virtualenv.

- Install `virtualfish` to the global space

```
sudo pip install virtualfish
```

- Load the python module `virtualfish` during the fish shell startup

```
$ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
```

- Edit this function `fish_prompt` by `$ funcdef fish_prompt --editor vim` and add the below lines and close the vim editor

```
if set -q VIRTUAL_ENV  
echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color  
normal) " "  
end
```

Note: If you are unfamiliar with vim, simply supply your favorite editor like this `$ funcdef fish_prompt --editor nano` or `$ funcdef fish_prompt --editor gedit`

- Save changes using `funcsave`

```
funcsave fish_prompt
```

- To create a new virtual environment use `vf new`

```
vf new my_new_env # Make sure $HOME/.virtualenv exists
```

- 如果你想创建一个新的 python3 环境，可以通过 -p 参数指定

```
vf new -p python3 my_new_env
```

- 切换虚拟环境请使用 vf deactivate 和 vf activate another\_env

官方链接：

- <https://github.com/adambrenecki/virtualfish>
- <http://virtualfish.readthedocs.io/en/latest/>

- If you want create a new python3 environment specify it via -p flag

```
vf new -p python3 my_new_env
```

- To switch between virtualenvironments use vf deactivate & vf activate another\_env

Official Links:

- <https://github.com/adambrenecki/virtualfish>
- <http://virtualfish.readthedocs.io/en/latest/>

# 第174章：Python 虚拟环境 - virtualenv

虚拟环境（“virtualenv”）是一种创建隔离 Python 环境的工具。它通过为不同项目创建虚拟 Python 环境，将它们所需的依赖分开存放。它解决了“项目 A 依赖版本 2.xxx，但项目 B 需要 2.xxx”的矛盾，同时保持你的全局 site-packages 目录干净且易于管理。

“virtualenv” 创建一个文件夹，里面包含了一个 Python 项目所需使用的所有必要的库和可执行文件。

## 第174.1节：安装

通过pip / (apt-get)安装virtualenv：

```
pip install virtualenv
```

或者

```
apt-get install python-virtualenv
```

注意：如果遇到权限问题，请使用sudo。

## 第174.2节：使用

```
$ cd test_proj
```

创建虚拟环境：

```
$ virtualenv test_proj
```

要开始使用虚拟环境，需要先激活它：

```
$ source test_project/bin/activate
```

要退出你的虚拟环境，只需输入“deactivate”：

```
$ deactivate
```

## 第174.3节：在你的虚拟环境中安装包

如果你查看虚拟环境中的bin目录，你会看到easy\_install，它已被修改为将egg和包放入虚拟环境的site-packages目录中。要在你的虚拟环境中安装一个应用：

```
$ source test_project/bin/activate  
$ pip install flask
```

此时，你不必使用sudo，因为所有文件都会安装在本地虚拟环境的site-packages目录中。该目录是以你自己的用户账户创建的。

# Chapter 174: Python Virtual Environment - virtualenv

A Virtual Environment ("virtualenv") is a tool to create isolated Python environments. It keeps the dependencies required by different projects in separate places, by creating virtual Python env for them. It solves the “project A depends on version 2.xxx but, project B needs 2.xxx” dilemma, and keeps your global site-packages directory clean and manageable.

“virtualenv” creates a folder which contains all the necessary libs and bins to use the packages that a Python project would need.

## Section 174.1: Installation

Install virtualenv via pip / (apt-get):

```
pip install virtualenv
```

OR

```
apt-get install python-virtualenv
```

Note: In case you are getting permission issues, use sudo.

## Section 174.2: Usage

```
$ cd test_proj
```

Create virtual environment:

```
$ virtualenv test_proj
```

To begin using the virtual environment, it needs to be activated:

```
$ source test_project/bin/activate
```

To exit your virtualenv just type “deactivate”:

```
$ deactivate
```

## Section 174.3: Install a package in your Virtualenv

If you look at the bin directory in your virtualenv, you'll see easy\_install which has been modified to put eggs and packages in the virtualenv's site-packages directory. To install an app in your virtual environment:

```
$ source test_project/bin/activate  
$ pip install flask
```

At this time, you don't have to use sudo since the files will all be installed in the local virtualenv site-packages directory. This was created as your own user account.

## 第174.4节：其他有用的virtualenv命令

**lsvirtualenv** : 列出所有环境。

**cdvirtualenv** : 进入当前激活的虚拟环境目录，这样你就可以浏览其 site-packages，例如。

**cdsitepackages** : 与上述类似，但直接进入site-packages目录。

**lssitepackages** : 显示site-packages目录的内容。

## Section 174.4: Other useful virtualenv commands

**lsvirtualenv** : List all of the environments.

**cdvirtualenv** : Navigate into the directory of the currently activated virtual environment, so you can browse its site-packages, for example.

**cdsitepackages** : Like the above, but directly into site-packages directory.

**lssitepackages** : Shows contents of site-packages directory.

# 视频：机器学习 A-Z：动手 Python数据科学

学习如何从两位数据科学专家那里用Python创建机器学习算法。包含代码模板。



- ✓ 精通Python上的机器学习
- ✓ 对多种机器学习模型有良好的直觉
- ✓ 做出准确的预测
- ✓ 进行强有力的数据分析
- ✓ 制作稳健的机器学习模型
- ✓ 为您的业务创造强大的附加价值
- ✓ 将机器学习用于个人目的
- ✓ 处理强化学习、自然语言处理和深度学习等特定主题
- ✓ 掌握降维等高级技术
- ✓ 了解针对每种问题应选择哪种机器学习模型
- ✓ 构建一支强大的机器学习模型军团，并知道如何组合它们以解决任何问题

今天观看 →

# VIDEO: Machine Learning A-Z: Hands-On Python In Data Science

Learn to create Machine Learning Algorithms in Python from two Data Science experts. Code templates included.



- ✓ Master Machine Learning on Python
- ✓ Have a great intuition of many Machine Learning models
- ✓ Make accurate predictions
- ✓ Make powerful analysis
- ✓ Make robust Machine Learning models
- ✓ Create strong added value to your business
- ✓ Use Machine Learning for personal purpose
- ✓ Handle specific topics like Reinforcement Learning, NLP and Deep Learning
- ✓ Handle advanced techniques like Dimensionality Reduction
- ✓ Know which Machine Learning model to choose for each type of problem
- ✓ Build an army of powerful Machine Learning models and know how to combine them to solve any problem

Watch Today →

# 第175章：使用 virtualenvwrapper创建虚拟环境

假设你需要同时处理三个不同的项目：项目A、项目B和项目C。项目A和项目B需要Python 3及一些必需的库。但项目C需要Python 2.7及其依赖库。

因此，最佳做法是将这些项目环境分开。要创建虚拟环境，你可以使用以下技术：

Virtualenv、Virtualenvwrapper和Conda

虽然我们有多种虚拟环境选项，但最推荐的是virtualenvwrapper。

## 第175.1节：使用 virtualenvwrapper创建虚拟环境

假设你需要同时处理三个不同的项目：项目A、项目B和项目C。项目A和项目B需要Python 3及一些必需的库。但项目C需要Python 2.7及其依赖库。

因此，最佳做法是将这些项目环境分开。要创建虚拟环境，你可以使用以下技术：

Virtualenv、Virtualenvwrapper和Conda

虽然我们有多种虚拟环境选项，但最推荐的是virtualenvwrapper。

**虽然我们有多种虚拟环境选项，但我总是更喜欢virtualenvwrapper，因为它比其他工具提供更多功能。**

```
$ pip install virtualenvwrapper

$ export WORKON_HOME=~/Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh$ printf '
%ss%s' '# virtualenv' 'export WORKON_HOME=~/virtualenvs' 'source/home/salayhin/bin/virtualenv
wrapper.sh' >> ~/.bashrc
$ source ~/.bashrc

$ mkvirtualenv python_3.5
Installing
setuptools.....
.....
.....done.
virtualenvwrapper.user_scripts 创建 /Users/salayhin/Envs/python_3.5/bin/predeactivate
virtualenvwrapper.user_scripts 创建 /Users/salayhin/Envs/python_3.5/bin/postdeactivate
virtualenvwrapper.user_scripts 创建 /Users/salayhin/Envs/python_3.5/bin/preactivate
virtualenvwrapper.user_scripts 创建 /Users/salayhin/Envs/python_3.5/bin/postactivate 新的 python
可执行文件 in python_3.5/bin/python

(python_3.5)$ ls $WORKON_HOME
python_3.5 hook.log
```

现在我们可以在环境中安装一些软件。

# Chapter 175: Virtual environment with virtualenvwrapper

Suppose you need to work on three different projects project A, project B and project C. project A and project B need python 3 and some required libraries. But for project C you need python 2.7 and dependent libraries.

So best practice for this is to separate those project environments. To create virtual environment you can use below technique:

Virtualenv, Virtualenvwrapper and Conda

Although we have several options for virtual environment but virtualenvwrapper is most recommended.

## Section 175.1: Create virtual environment with virtualenvwrapper

Suppose you need to work on three different projects project A, project B and project C. project A and project B need python 3 and some required libraries. But for project C you need python 2.7 and dependent libraries.

So best practice for this is to separate those project environments. To create virtual environment you can use below technique:

Virtualenv, Virtualenvwrapper and Conda

Although we have several options for virtual environment but virtualenvwrapper is most recommended.

**Although we have several options for virtual environment but I always prefer virtualenvwrapper because it has more facility than others.**

```
$ pip install virtualenvwrapper

$ export WORKON_HOME=~/Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
$ printf '\n%s\n%s\n%s' '# virtualenv' 'export WORKON_HOME=~/virtualenvs' 'source
/home/salayhin/bin/virtualenvwrapper.sh' >> ~/.bashrc
$ source ~/.bashrc

$ mkvirtualenv python_3.5
Installing
setuptools.....
.....
.....done.
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postactivate New python
executable in python_3.5/bin/python

(python_3.5)$ ls $WORKON_HOME
python_3.5 hook.log
```

Now we can install some software into the environment.

```
(python_3.5)$ pip install django
正在下载/解包 django
正在下载 Django-1.1.1.tar.gz (5.6Mb) : 已下载 5.6Mb
正在为包 django 运行 setup.py egg_info
正在安装收集的包 : django
正在为 django 运行 setup.py install
将 build/scripts-2.6/django-admin.py 的权限从 644 改为 755
将 /Users/salayhin/Envs/env1/bin/django-admin.py 的权限改为 755
django 安装成功
```

我们可以用 `lssitepackages` 查看新安装的包：

```
(python_3.5)$ lssitepackages
Django-1.1.1-py2.6.egg-info easy-install.pth
setuptools-0.6.10-py2.6.egg pip-0.6.3-py2.6.egg
django setuptools.pth
```

如果需要，我们可以创建多个虚拟环境。

使用 `workon` 在环境间切换：

```
(python_3.6)$ workon python_3.5
(python_3.5)$ echo $VIRTUAL_ENV
/Users/salayhin/Envs/env1
(python_3.5)$
```

退出虚拟环境

```
$ deactivate
```

```
(python_3.5)$ pip install django
Downloading/unpacking django
  Downloading Django-1.1.1.tar.gz (5.6Mb): 5.6Mb downloaded
    Running setup.py egg_info for package django
      Installing collected packages: django
        Running setup.py install for django
          changing mode of build/scripts-2.6/django-admin.py from 644 to 755
          changing mode of /Users/salayhin/Envs/env1/bin/django-admin.py to 755
          Successfully installed django
```

We can see the new package with `lssitepackages`:

```
(python_3.5)$ lssitepackages
Django-1.1.1-py2.6.egg-info easy-install.pth
setuptools-0.6.10-py2.6.egg pip-0.6.3-py2.6.egg
django setuptools.pth
```

We can create multiple virtual environment if we want.

Switch between environments with `workon`:

```
(python_3.6)$ workon python_3.5
(python_3.5)$ echo $VIRTUAL_ENV
/Users/salayhin/Envs/env1
(python_3.5)$
```

To exit the virtualenv

```
$ deactivate
```

# 第176章：在Windows中使用virtualenvwrapper创建虚拟环境

## 第176.1节：Windows下使用virtualenvwrapper的虚拟环境

假设你需要同时处理三个不同的项目：项目A、项目B和项目C。项目A和项目B需要Python 3及一些必需的库。但项目C需要Python 2.7及其依赖库。

最佳实践是将项目环境分开。要创建独立的Python虚拟环境，需要遵循以下步骤：

步骤1： 使用以下命令安装pip：python -m pip install -U pip

然后使用命令安装“virtualenvwrapper-win”包（命令可在Windows PowerShell中执行）：

```
pip install virtualenvwrapper-win
```

步骤3： 使用命令创建新的虚拟环境：mkvirtualenv python\_3.5

步骤4： 使用命令激活环境：

```
workon < 环境名称>
```

virtualenvwrapper的主要命令：

```
mkvirtualenv <name>
```

创建一个名为<name>的新虚拟环境。该环境将被创建在WORKON\_HOME目录中。

```
lsvirtualenv
```

列出所有存储在WORKON\_HOME中的环境。

```
rmvirtualenv <name>
```

移除环境 <name>。使用 folder\_delete.bat。

```
workon [<name>]
```

如果指定了<name>，则激活名为<name>的环境（将工作虚拟环境切换为<name>）。如果已定义项目目录，我们将切换到该目录。如果未指定参数，则列出可用的环境。可以在 virtualenv 名称后传递额外选项 -c，以便在未设置 projectdir 时切换到 virtualenv 目录。

停用

停用当前虚拟环境并切换回默认的系统 Python。

```
add2virtualenv <完整路径或相对路径>
```

如果虚拟环境（virtualenv）处于激活状态，则将 <路径> 追加到该环境的 site-packages 目录下的 virtualenv\_path\_extensions.pth 文件中，这实际上将 <路径> 添加到了该环境的 PYTHONPATH 中。如果虚拟环境未激活，则将 <路径> 追加到默认 Python 的 site-packages 目录下的 virtualenv\_path\_extensions.pth 文件中。如果 <路径> 不存在，则会创建该文件。

# Chapter 176: Create virtual environment with virtualenvwrapper in windows

## Section 176.1: Virtual environment with virtualenvwrapper for windows

Suppose you need to work on three different projects project A, project B and project C. project A and project B need python 3 and some required libraries. But for project C you need python 2.7 and dependent libraries.

So best practice for this is to separate those project environments. For creating separate python virtual environment need to follow below steps:

**Step 1:** Install pip with this command: python -m pip install -U pip

**Step 2:** Then install "virtualenvwrapper-win" package by using command (command can be executed windows power shell):

```
pip install virtualenvwrapper-win
```

**Step 3:** Create a new virtualenv environment by using command: mkvirtualenv python\_3.5

**Step 4:** Activate the environment by using command:

```
workon < environment name>
```

Main commands for virtualenvwrapper:

```
mkvirtualenv <name>
```

Create a new virtualenv environment named <name>. The environment will be created in WORKON\_HOME.

```
lsvirtualenv
```

List all of the environments stored in WORKON\_HOME.

```
rmvirtualenv <name>
```

Remove the environment <name>. Uses folder\_delete.bat.

```
workon [<name>]
```

If <name> is specified, activate the environment named <name> (change the working virtualenv to <name>). If a project directory has been defined, we will change into it. If no argument is specified, list the available environments. One can pass additional option -c after virtualenv name to cd to virtualenv directory if no projectdir is set.

```
deactivate
```

Deactivate the working virtualenv and switch back to the default system Python.

```
add2virtualenv <full or relative path>
```

If a virtualenv environment is active, appends <path> to virtualenv\_path\_extensions.pth inside the environment's site-packages, which effectively adds <path> to the environment's PYTHONPATH. If a virtualenv environment is not active, appends <path> to virtualenv\_path\_extensions.pth inside the default Python's site-packages. If <path> doesn't exist, it will be created.

# 第177章：sys

sys 模块提供了访问与程序运行环境相关的函数和变量，例如 sys.argv 中的命令行参数，或用于从程序任意位置结束当前进程的函数 sys.exit()。

虽然被清晰地划分为一个模块，但它实际上是内置的，因此在正常情况下总是可用的。

## 第177.1节：命令行参数

```
if len(sys.argv) != 4:      # 需要考虑脚本名称本身。
    raise RuntimeError("expected 3 command line arguments")

f = open(sys.argv[1], 'rb')  # 使用第一个命令行参数。
start_line = int(sys.argv[2]) # 所有参数都是字符串，因此如果需要其他类型，必须显式转换
end_line = int(sys.argv[3]) # 进行转换。
```

请注意，在更大且更完善的程序中，您会使用诸如[click](#)这样的模块来处理命令行参数，而不是自己手动处理。

## 第177.2节：脚本名称

```
# 被执行脚本的名称位于argv列表的开头。
print('用法:', sys.argv[0], '<文件名> <起始> <结束>')

# 您可以使用它来生成被执行程序的路径前缀
# (与当前模块相对)，以便访问相对于该路径的文件，
# 例如这对于游戏资源来说非常有用。
program_file = sys.argv[0]

import pathlib
program_path = pathlib.Path(program_file).resolve().parent
```

## 第177.3节：标准错误流

```
# 错误信息应尽可能不输出到标准输出。
print('错误：我们根本没有奶酪。', file=sys.stderr)

尝试：
f = open('nonexistent-file.xyz', 'rb')
except OSError as e:
    print(e, file=sys.stderr)
```

## 第177.4节：提前结束进程并返回退出代码

```
def main():
    if len(sys.argv) != 4 or '--help' in sys.argv[1:]:
        print('用法：my_program <arg1> <arg2> <arg3>', file=sys.stderr)

        sys.exit(1)      # 使用退出代码表示程序未成功

    process_data()
```

# Chapter 177: sys

The **sys** module provides access to functions and values concerning the program's runtime environment, such as the command line parameters in `sys.argv` or the function `sys.exit()` to end the current process from any point in the program flow.

While cleanly separated into a module, it's actually built-in and as such will always be available under normal circumstances.

## Section 177.1: Command line arguments

```
if len(sys.argv) != 4:      # The script name needs to be accounted for as well.
    raise RuntimeError("expected 3 command line arguments")

f = open(sys.argv[1], 'rb')  # Use first command line argument.
start_line = int(sys.argv[2]) # All arguments come as strings, so need to be
end_line = int(sys.argv[3]) # converted explicitly if other types are required.
```

Note that in larger and more polished programs you would use modules such as [click](#) to handle command line arguments instead of doing it yourself.

## Section 177.2: Script name

```
# The name of the executed script is at the beginning of the argv list.
print('usage:', sys.argv[0], '<filename> <start> <end>')

# You can use it to generate the path prefix of the executed program
# (as opposed to the current module) to access files relative to that,
# which would be good for assets of a game, for instance.
program_file = sys.argv[0]

import pathlib
program_path = pathlib.Path(program_file).resolve().parent
```

## Section 177.3: Standard error stream

```
# Error messages should not go to standard output, if possible.
print('ERROR: We have no cheese at all.', file=sys.stderr)

try:
    f = open('nonexistent-file.xyz', 'rb')
except OSError as e:
    print(e, file=sys.stderr)
```

## Section 177.4: Ending the process prematurely and returning an exit code

```
def main():
    if len(sys.argv) != 4 or '--help' in sys.argv[1:]:
        print('usage: my_program <arg1> <arg2> <arg3>', file=sys.stderr)

        sys.exit(1)      # use an exit code to signal the program was unsuccessful

    process_data()
```

# 第178章：ChemPy - Python包

ChemPy 是一个主要用于解决物理、分析和无机化学问题的 Python 包。它是一个免费、开源的 Python 工具包，适用于化学、化学工程和材料科学应用。

## 第178.1节：解析化学式

```
from chempy import Substance
ferricyanide = Substance.from_formula('Fe(CN)6-3')
ferricyanide.composition == {0: -3, 26: 1, 6: 6, 7: 6}
True
print(ferricyanide.unicode_name)
Fe(CN)₆³⁻
print(ferricyanide.latex_name + ", " + ferricyanide.html_name)
Fe(CN)_{6}^{3-}, Fe(CN)<sub>6</sub><sup>3-</sup>
print('%.3f' % ferricyanide.mass)
211.955
```

在组成中，原子序数（以及0表示电荷）用作键，每种元素的数量作为对应的值。

## 第178.2节：化学反应的化学计量平衡

```
from chempy import balance_stoichiometry # NASA助推火箭的主要反应：
reac, prod = balance_stoichiometry({'NH4ClO4', 'Al'}, {'Al2O3', 'HCl', 'H2O', 'N2'})
from pprint import pprint
pprint(reac)
{'铝': 10, '高氯酸铵': 6}
pprint(prod)
{'氧化铝': 5, '水': 9, '盐酸': 6, '氮气': 3}
from chempy import mass_fractions
for fractions in map(mass_fractions, [reac, prod]):
...     pprint({k: '{0:.3g} wt%'.format(v*100) for k, v in fractions.items()})
...
{'铝': '27.7 wt%', '高氯酸铵': '72.3 wt%'}
{'氧化铝': '52.3 wt%', '水': '16.6 wt%', '盐酸': '22.4 wt%', '氮气': '8.62 wt%'}
```

## 第178.3节：反应配平

```
from chempy import Equilibrium
from sympy import symbols
K1, K2, Kw = symbols('K1 K2 Kw')
e1 = Equilibrium({'高锰酸根': 1, '氢离子': 8, '电子': 5}, {'锰离子': 1, '水': 4}, K1)
e2 = Equilibrium({'氧气': 1, '水': 2, '电子': 4}, {'氢氧根': 4}, K2)
coeff = Equilibrium.eliminate([e1, e2], 'e-')
coeff
[4, -5]
redox = e1*coeff[0] + e2*coeff[1]
print(redox)
20OH⁻ + 32 H⁺ + 4 MnO₄⁻ = 26 H₂O + 4 Mn²⁺ + 5 O₂; K₁**⁴/K₂**⁵
autoprot = Equilibrium({'H₂O': 1}, {'H⁺': 1, 'OH⁻': 1}, Kw)
n = redox.cancel(autoprot)
n
20
redox2 = redox + n*autoprot
print(redox2)
```

# Chapter 178: ChemPy - python package

ChemPy is a python package designed mainly to solve and address problems in physical, analytical and inorganic Chemistry. It is a free, open-source Python toolkit for chemistry, chemical engineering, and materials science applications.

## Section 178.1: Parsing formulae

```
from chempy import Substance
ferricyanide = Substance.from_formula('Fe(CN)6-3')
ferricyanide.composition == {0: -3, 26: 1, 6: 6, 7: 6}
True
print(ferricyanide.unicode_name)
Fe(CN)₆³⁻
print(ferricyanide.latex_name + ", " + ferricyanide.html_name)
Fe(CN)_{6}^{3-}, Fe(CN)<sub>6</sub><sup>3-</sup>
print('%.3f' % ferricyanide.mass)
211.955
```

In composition, the atomic numbers (and 0 for charge) is used as keys and the count of each kind became respective value.

## Section 178.2: Balancing stoichiometry of a chemical reaction

```
from chempy import balance_stoichiometry # Main reaction in NASA's booster rockets:
reac, prod = balance_stoichiometry({'NH4ClO4', 'Al'}, {'Al2O3', 'HCl', 'H2O', 'N2'})
from pprint import pprint
pprint(reac)
{'Al': 10, 'NH4ClO4': 6}
pprint(prod)
{'Al2O3': 5, 'H2O': 9, 'HCl': 6, 'N2': 3}
from chempy import mass_fractions
for fractions in map(mass_fractions, [reac, prod]):
...     pprint({k: '{0:.3g} wt%'.format(v*100) for k, v in fractions.items()})
...
{'Al': '27.7 wt%', 'NH4ClO4': '72.3 wt%'}
{'Al2O3': '52.3 wt%', 'H2O': '16.6 wt%', 'HCl': '22.4 wt%', 'N2': '8.62 wt%'}
```

## Section 178.3: Balancing reactions

```
from chempy import Equilibrium
from sympy import symbols
K1, K2, Kw = symbols('K1 K2 Kw')
e1 = Equilibrium({'MnO₄⁻': 1, 'H⁺': 8, 'e⁻': 5}, {'Mn²⁺': 1, 'H₂O': 4}, K1)
e2 = Equilibrium({'O₂': 1, 'H₂O': 2, 'e⁻': 4}, {'OH⁻': 4}, K2)
coeff = Equilibrium.eliminate([e1, e2], 'e-')
coeff
[4, -5]
redox = e1*coeff[0] + e2*coeff[1]
print(redox)
20 OH⁻ + 32 H⁺ + 4 MnO₄⁻ = 26 H₂O + 4 Mn²⁺ + 5 O₂; K₁**⁴/K₂**⁵
autoprot = Equilibrium({'H₂O': 1}, {'H⁺': 1, 'OH⁻': 1}, Kw)
n = redox.cancel(autoprot)
n
20
redox2 = redox + n*autoprot
print(redox2)
```

## 第178.4节：化学平衡

```
from chempy import Equilibrium
from chempy.chemistry import Species
water_autop = Equilibrium({'H2O'}, {'H+', 'OH-'}, 10**-14) # 假定单位为“摩尔浓度”
ammonia_prot = Equilibrium({'NH4+'}, {'NH3', 'H+'}, 10**-9.24) # 同上
from chempy.equilibria import EqSystem
substances = map(Species.from_formula, 'H2O OH- H+ NH3 NH4+'.split())
eqsys = EqSystem([water_autop, ammonia_prot], substances)
print("\n".join(map(str, eqsys.rxns))) # "rxns" 是 "reactions" 的缩写
H2O = H+ + OH-; 1e-14
NH4+ = H+ + NH3; 5.75e-10
from collections import defaultdict
init_conc = defaultdict(float, {'H2O': 1, 'NH3': 0.1})
x, sol, sane = eqsys.root(init_conc)
assert sol['success'] and sane
print(sorted(sol.keys())) # 更多信息请参见“pyneqsys”包
['fun', 'intermediate_info', 'internal_x_vecs', 'nfev', 'njev', 'success', 'x', 'x_vecs']
print(', '.join(['%.2g' % v for v in x]))
1, 0.0013, 7.6e-12, 0.099, 0.0013
```

## 第178.5节：离子强度

```
from chempy.electrolytes import ionic_strength
ionic_strength({'Fe3+': 0.05, 'ClO4-': 0.15}) == .3
True
```

## 第178.6节：化学动力学（常微分方程组）

```
from chempy import ReactionSystem # 以下速率常数为任意值
rsys = ReactionSystem.from_string("""2 Fe+2 + H2O2 -> 2 Fe+3 + 2 OH-; 42
2 Fe+3 + H2O2 -> 2 Fe+2 + O2 + 2 H+; 17
H+ + OH- -> H2O; 1e10
H2O -> H+ + OH-; 1e-4
Fe+3 + 2 H2O -> FeOOH(s) + 3 H+; 1
FeOOH(s) + 3 H+ -> Fe+3 + 2 H2O; 2.5""") # "[H2O]" = 1.0 (实际为室温下55.4)
from chempy.kinetics.ode import get_odesys
odesys, extra = get_odesys(rsys)
from collections import defaultdict
import numpy as np
tout = sorted(np.concatenate((np.linspace(0, 23), np.logspace(-8, 1))))
c0 = defaultdict(float, {'Fe2+': 0.05, 'H2O2': 0.1, 'H2O': 1.0, 'H+': 1e-7, 'OH-': 1e-7})
result = odesys.integrate(tout, c0, atol=1e-12, rtol=1e-14)
import matplotlib.pyplot as plt
_ = plt.subplot(1, 2, 1)
_= result.plot(names=[k for k in rsys.substances if k != 'H2O'])
_= plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('时间'); _ =
plt.ylabel('浓度')
_= plt.subplot(1, 2, 2)
_= result.plot(names=[k for k in rsys.substances if k != 'H2O'], xscale='log', yscale='log')
_= plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('时间'); _ =
plt.ylabel('浓度')
_= plt.tight_layout()
plt.show()
```

## Section 178.4: Chemical equilibria

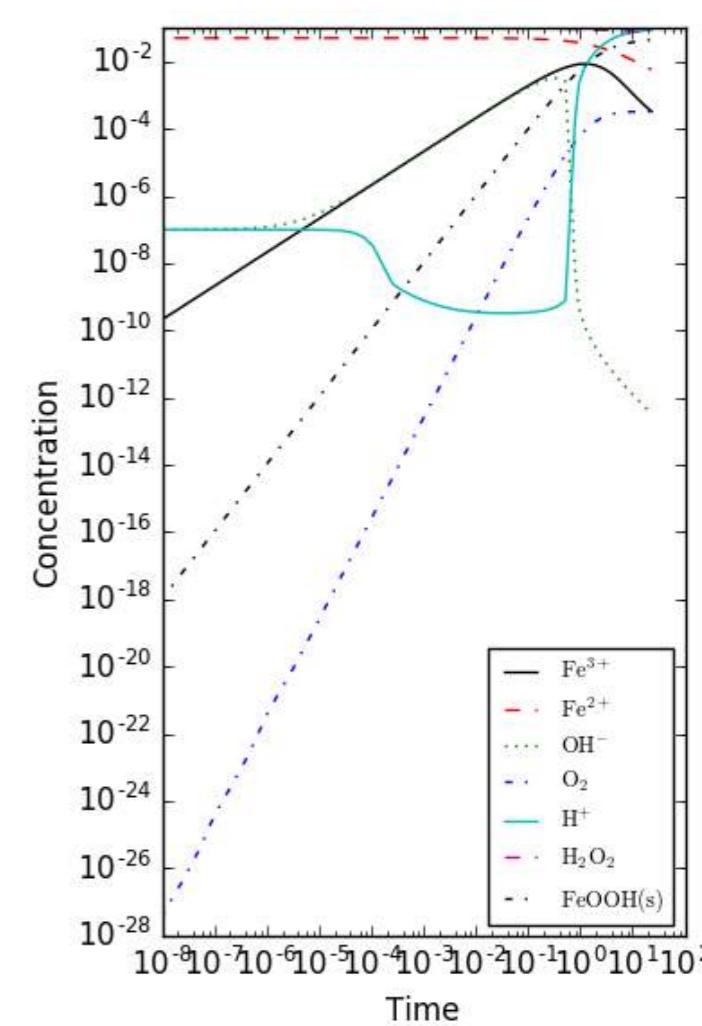
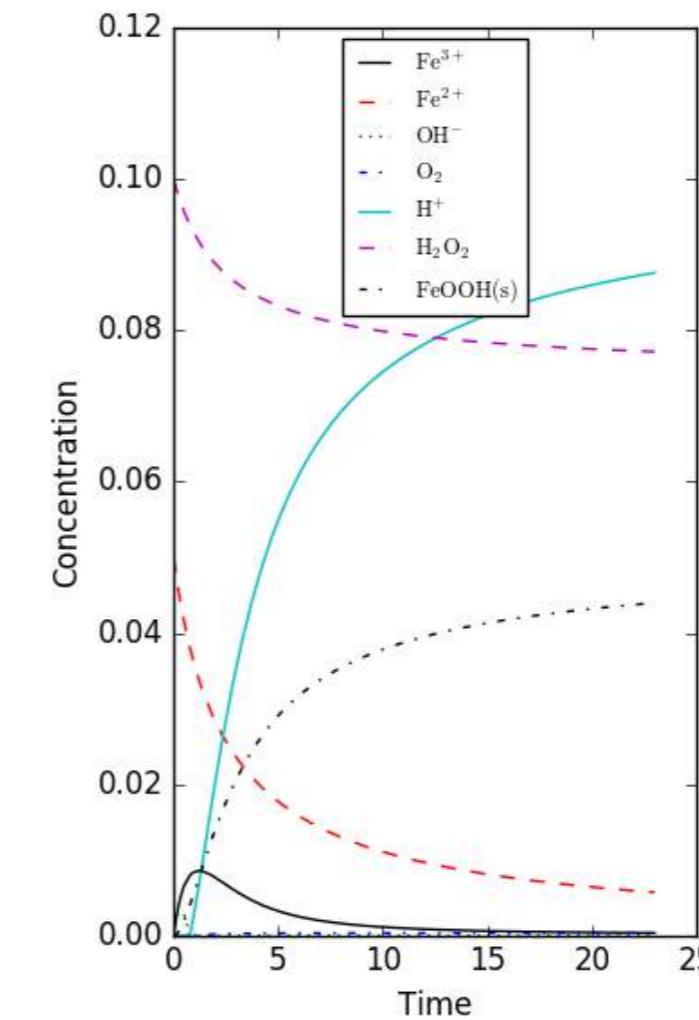
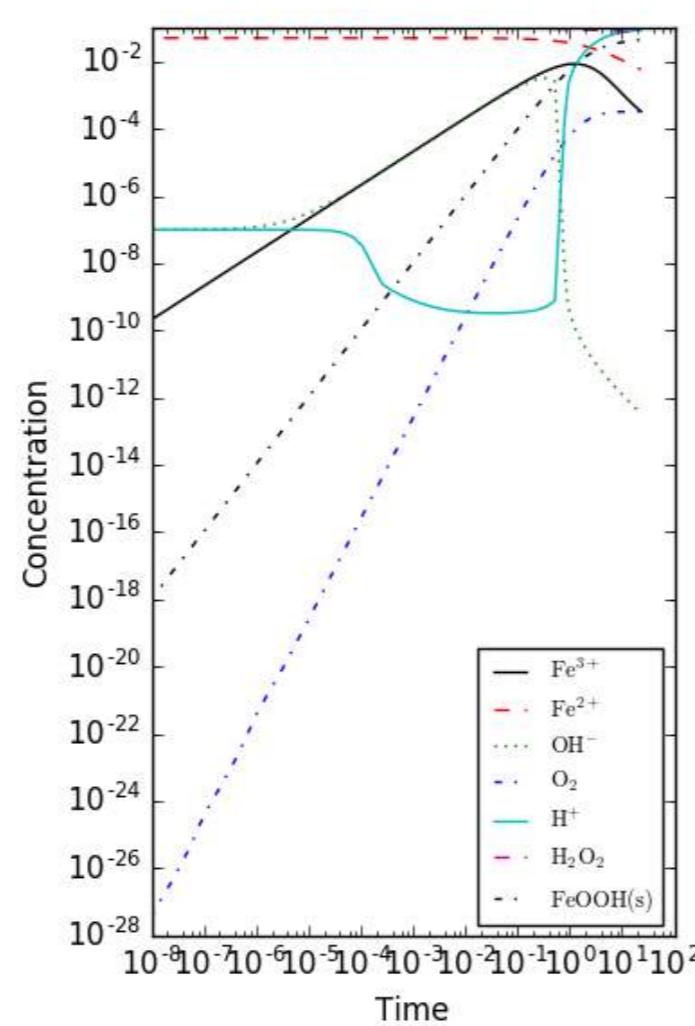
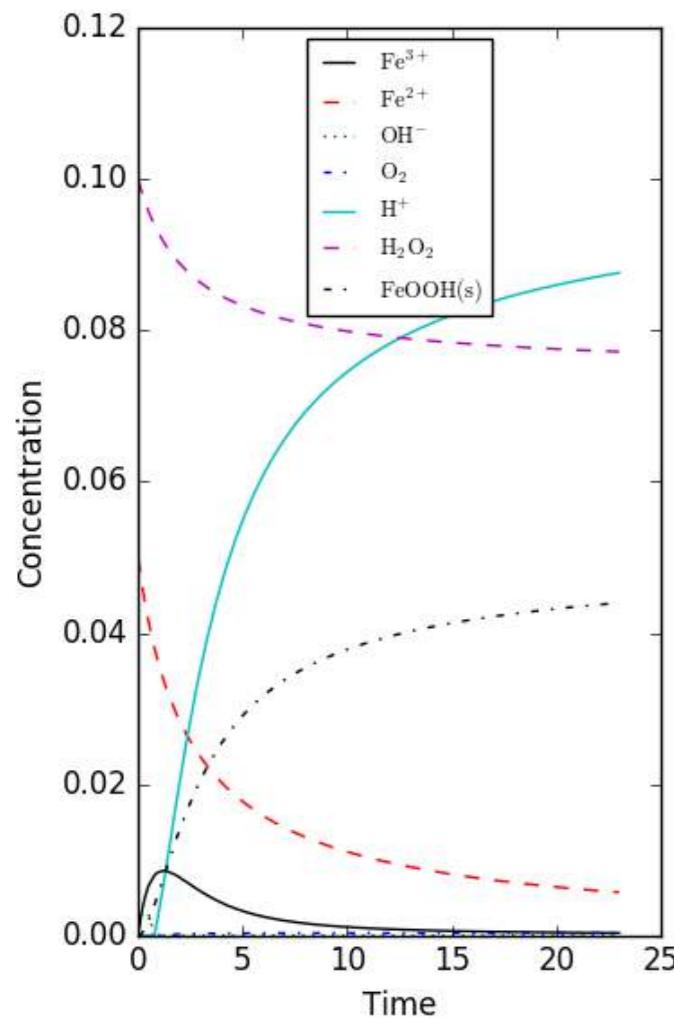
```
from chempy import Equilibrium
from chempy.chemistry import Species
water_autop = Equilibrium({'H2O'}, {'H+', 'OH-'}, 10**-14) # unit "molar" assumed
ammonia_prot = Equilibrium({'NH4+'}, {'NH3', 'H+'}, 10**-9.24) # same here
from chempy.equilibria import EqSystem
substances = map(Species.from_formula, 'H2O OH- H+ NH3 NH4+'.split())
eqsys = EqSystem([water_autop, ammonia_prot], substances)
print("\n".join(map(str, eqsys.rxns))) # "rxns" short for "reactions"
H2O = H+ + OH-; 1e-14
NH4+ = H+ + NH3; 5.75e-10
from collections import defaultdict
init_conc = defaultdict(float, {'H2O': 1, 'NH3': 0.1})
x, sol, sane = eqsys.root(init_conc)
assert sol['success'] and sane
print(sorted(sol.keys())) # see package "pyneqsys" for more info
['fun', 'intermediate_info', 'internal_x_vecs', 'nfev', 'njev', 'success', 'x', 'x_vecs']
print(', '.join(['%.2g' % v for v in x]))
1, 0.0013, 7.6e-12, 0.099, 0.0013
```

## Section 178.5: Ionic strength

```
from chempy.electrolytes import ionic_strength
ionic_strength({'Fe3+': 0.05, 'ClO4-': 0.15}) == .3
True
```

## Section 178.6: Chemical kinetics (system of ordinary differential equations)

```
from chempy import ReactionSystem # The rate constants below are arbitrary
rsys = ReactionSystem.from_string("""2 Fe+2 + H2O2 -> 2 Fe+3 + 2 OH-; 42
2 Fe+3 + H2O2 -> 2 Fe+2 + O2 + 2 H+; 17
H+ + OH- -> H2O; 1e10
H2O -> H+ + OH-; 1e-4
Fe+3 + 2 H2O -> FeOOH(s) + 3 H+; 1
FeOOH(s) + 3 H+ -> Fe+3 + 2 H2O; 2.5""") # "[H2O]" = 1.0 (actually 55.4 at RT)
from chempy.kinetics.ode import get_odesys
odesys, extra = get_odesys(rsys)
from collections import defaultdict
import numpy as np
tout = sorted(np.concatenate((np.linspace(0, 23), np.logspace(-8, 1))))
c0 = defaultdict(float, {'Fe2+': 0.05, 'H2O2': 0.1, 'H2O': 1.0, 'H+': 1e-7, 'OH-': 1e-7})
result = odesys.integrate(tout, c0, atol=1e-12, rtol=1e-14)
import matplotlib.pyplot as plt
_ = plt.subplot(1, 2, 1)
_= result.plot(names=[k for k in rsys.substances if k != 'H2O'])
_= plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ =
plt.ylabel('Concentration')
_= plt.subplot(1, 2, 2)
_= result.plot(names=[k for k in rsys.substances if k != 'H2O'], xscale='log', yscale='log')
_= plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ =
plt.ylabel('Concentration')
_= plt.tight_layout()
plt.show()
```



# 第179章：pygame

参数	详细信息
count	表示需要保留的通道数量之类的正整数。
force	一个布尔值 ( <code>False</code> 或 <code>True</code> )，用于确定 <code>find_channel()</code> 是否必须返回一个通道 (无论是否活跃) 时返回 <code>True</code> ，或者 (如果没有非活跃通道) 返回 <code>False</code>

Pygame 是用于制作多媒体应用程序，尤其是游戏的首选 Python 库。官方网站是 <http://www.pygame.org/>。

## 第179.1节：Pygame 的 mixer 模块

`pygame.mixer` 模块帮助控制 pygame 程序中使用的音乐。目前，mixer 模块有15个不同的函数。

### 初始化

类似于必须使用 `pygame.init()` 初始化 pygame，`pygame.mixer` 也必须初始化。

使用第一个选项时，我们使用默认值初始化模块。不过，你也可以覆盖这些默认选项。使用第二个选项时，我们可以使用自己手动输入的值来初始化模块。  
标准值：

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

要检查我们是否已经初始化，可以使用`pygame.mixer.get_init()`，如果已初始化则返回`True`，未初始化则返回`False`。要退出/撤销初始化，只需使用`pygame.mixer.quit()`。如果想继续使用该模块播放声音，可能需要重新初始化该模块。

### 可能的操作

当声音正在播放时，可以使用`pygame.mixer.pause()`暂时暂停。要恢复播放声音，只需使用`pygame.mixer.unpause()`。你也可以通过使用`pygame.mixer.fadeout()`来淡出声音的结尾。它接受一个参数，即完成淡出音乐所需的毫秒数。

### 通道

只要有足够的空闲通道支持，你可以播放任意数量的歌曲。默认情况下，有8个通道。要更改通道数量，使用`pygame.mixer.set_num_channels()`。参数是一个非负整数。如果通道数量减少，任何在被移除通道上播放的声音将立即停止。

要查询当前有多少通道正在使用，调用`pygame.mixer.get_channels(count)`。输出是当前未打开的通道数量。你也可以通过使用`pygame.mixer.set_reserved(count)`为必须播放的声音保留通道。参数同样是非负整数。任何在新保留通道上播放的声音不会被停止。

你还可以使用`pygame.mixer.find_channel(force)`找出哪个通道未被使用。它的参数是一个布尔值：`True`或`False`。如果没有空闲通道且`force`为`False`，则返回`None`。如果`force`为`True`，则返回播放时间最长的通道。

# Chapter 179: pygame

Parameter	Details
count	A positive integer that represents something like the number of channels needed to be reserved.
force	A boolean value ( <code>False</code> or <code>True</code> ) that determines whether <code>find_channel()</code> has to return a channel (inactive or not) with <code>True</code> or not (if there are no inactive channels) with <code>False</code>

Pygame is the go-to library for making multimedia applications, especially games, in Python. The official website is <http://www.pygame.org/>.

## Section 179.1: Pygame's mixer module

The `pygame.mixer` module helps control the music used in pygame programs. As of now, there are 15 different functions for the `mixer` module.

### Initializing

Similar to how you have to initialize pygame with `pygame.init()`, you must initialize `pygame.mixer` as well.

By using the first option, we initialize the module using the default values. You can though, override these default options. By using the second option, we can initialize the module using the values we manually put in ourselves.  
Standard values:

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

To check whether we have initialized it or not, we can use `pygame.mixer.get_init()`, which returns `True` if it is and `False` if it is not. To quit/undo the initializing, simply use `pygame.mixer.quit()`. If you want to continue playing sounds with the module, you might have to reinitialize the module.

### Possible Actions

As your sound is playing, you can pause it temporarily with `pygame.mixer.pause()`. To resume playing your sounds, simply use `pygame.mixer.unpause()`. You can also fadeout the end of the sound by using `pygame.mixer.fadeout()`. It takes an argument, which is the number of milliseconds it takes to finish fading out the music.

### Channels

You can play as many songs as needed as long there are enough open channels to support them. By default, there are 8 channels. To change the number of channels there are, use `pygame.mixer.set_num_channels()`. The argument is a non-negative integer. If the number of channels are decreased, any sounds playing on the removed channels will immediately stop.

To find how many channels are currently being used, call `pygame.mixer.get_channels(count)`. The output is the number of channels that are not currently open. You can also reserve channels for sounds that must be played by using `pygame.mixer.set_reserved(count)`. The argument is also a non-negative integer. Any sounds playing on the newly reserved channels will not be stopped.

You can also find out which channel isn't being used by using `pygame.mixer.find_channel(force)`. Its argument is a bool: either `True` or `False`. If there are no channels that are idle and `force` is `False`, it will return `None`. If `force` is `true`, it will return the channel that has been playing for the longest time.

## 第179.2节：安装pygame

使用pip时：

```
pip install pygame
```

使用conda：

```
conda install -c tlatorre pygame=1.9.2
```

直接从网站下载：<http://www.pygame.org/download.shtml>

您可以找到适用于Windows和其他操作系统的合适安装程序。

项目也可以在<http://www.pygame.org/> 找到

## Section 179.2: Installing pygame

With pip:

```
pip install pygame
```

With conda:

```
conda install -c tlatorre pygame=1.9.2
```

Direct download from website : <http://www.pygame.org/download.shtml>

You can find the suitable installers for Windows and other operating systems.

Projects can also be found at <http://www.pygame.org/>

# 视频：机器学习、数据科学和使用 Python 的深度学习

完整的动手机器学习教程，涵盖  
数据科学、Tensorflow、人工智能  
和神经网络



- ✓ 使用Tensorflow和Keras构建人工神经网络
- ✓ 使用深度学习对图像、数据和情感进行分类
- ✓ 使用线性回归、多项式回归和多元回归进行预测
- ✓ 使用Matplotlib和Seaborn进行数据可视化
- ✓ 使用 Apache Spark 的 MLLib 实现大规模机器学习
- ✓ 理解强化学习——以及如何构建一个吃豆人机器人
- ✓ 使用 K-Means 聚类、支持向量机 (SVM) 、KNN、决策树、朴素贝叶斯和主成分分析 (PCA) 对数据进行分类
- ✓ 使用训练/测试和 K 折交叉验证来选择和调整模型
- ✓ 使用基于物品和基于用户的协同过滤构建电影推荐系统

今天观看 →

# VIDEO: Machine Learning, Data Science and Deep Learning with Python

Complete hands-on machine learning tutorial with data science, Tensorflow, artificial intelligence, and neural networks



- ✓ Build artificial neural networks with Tensorflow and Keras
- ✓ Classify images, data, and sentiments using deep learning
- ✓ Make predictions using linear regression, polynomial regression, and multivariate regression
- ✓ Data Visualization with Matplotlib and Seaborn
- ✓ Implement machine learning at massive scale with Apache Spark's MLLib
- ✓ Understand reinforcement learning - and how to build a Pac-Man bot
- ✓ Classify data using K-Means clustering, Support Vector Machines (SVM), KNN, Decision Trees, Naive Bayes, and PCA
- ✓ Use train/test and K-Fold cross validation to choose and tune your models
- ✓ Build a movie recommender system using item-based and user-based collaborative filtering

Watch Today →

# 第180章：Pyglet

Pyglet是一个用于视觉和声音的Python模块。它不依赖于其他模块。官方信息请参见[pyglet.org][1]  
[1]: <http://pyglet.org>

## 第180.1节：Pyglet的安装

安装Python，进入命令行并输入：

Python 2:

```
pip install pyglet
```

Python 3:

```
pip3 install pyglet
```

## 第180.2节：Pyglet中的Hello World

```
import pyglet
window = pyglet.window.Window()
label = pyglet.text.Label('Hello, world',
                         font_name='Times New Roman',
                         font_size=36,
                         x=window.width//2, y=window.height//2,
                         anchor_x='center', anchor_y='center')
@window.event
def on_draw():
    window.clear()
    label.draw()
pyglet.app.run()
```

## 第180.3节：在Pyglet中播放声音

```
sound = pyglet.media.load(sound.wav)
sound.play()
```

## 第180.4节：使用Pyglet进行OpenGL

```
import pyglet
from pyglet.gl import *

win = pyglet.window.Window()

@win.event()
def on_draw():
    #OpenGL代码写在这里。正常使用OpenGL。

pyglet.app.run()
```

## 第180.5节：使用Pyglet和OpenGL绘制点

```
import pyglet
from pyglet.gl import *
```

# Chapter 180: Pyglet

Pyglet is a Python module used for visuals and sound. It has no dependencies on other modules. See [pyglet.org][1]  
for the official information. [1]: <http://pyglet.org>

## Section 180.1: Installation of Pyglet

Install Python, go into the command line and type:

Python 2:

```
pip install pyglet
```

Python 3:

```
pip3 install pyglet
```

## Section 180.2: Hello World in Pyglet

```
import pyglet
window = pyglet.window.Window()
label = pyglet.text.Label('Hello, world',
                         font_name='Times New Roman',
                         font_size=36,
                         x=window.width//2, y=window.height//2,
                         anchor_x='center', anchor_y='center')
@window.event
def on_draw():
    window.clear()
    label.draw()
pyglet.app.run()
```

## Section 180.3: Playing Sound in Pyglet

```
sound = pyglet.media.load(sound.wav)
sound.play()
```

## Section 180.4: Using Pyglet for OpenGL

```
import pyglet
from pyglet.gl import *

win = pyglet.window.Window()

@win.event()
def on_draw():
    #OpenGL goes here. Use OpenGL as normal.

pyglet.app.run()
```

## Section 180.5: Drawing Points Using Pyglet and OpenGL

```
import pyglet
from pyglet.gl import *
```

```
win = pyglet.window.Window()
glClear(GL_COLOR_BUFFER_BIT)

@win.event
def on_draw():
    glBegin(GL_POINTS)
    glVertex2f(x, y) #x 是距离窗口左侧的期望距离, y 是距离窗口底部的期望距离
    #可以创建任意数量的顶点
    glEnd
```

要连接这些点，将GL\_POINTS替换为GL\_LINE\_LOOP。

```
win = pyglet.window.Window()
glClear(GL_COLOR_BUFFER_BIT)

@win.event
def on_draw():
    glBegin(GL_POINTS)
    glVertex2f(x, y) #x is desired distance from left side of window, y is desired distance from
    bottom of window
    #make as many vertexes as you want
    glEnd
```

To connect the points, replace GL\_POINTS with GL\_LINE\_LOOP.

# 第181章：音频

## 第181.1节：处理WAV文件

### winsound

- Windows环境

```
import winsound
winsound.PlaySound("path_to_wav_file.wav", winsound.SND_FILENAME)
```

### wave

- 支持单声道/立体声
- 不支持压缩/解压缩

```
import wave
with wave.open("path_to_wav_file.wav", "rb") as wav_file:    # 以只读模式打开WAV文件。
    # 获取基本信息。
    n_channels = wav_file.getnchannels()      # 通道数。 (1=单声道, 2=立体声)。
    sample_width = wav_file.getsampwidth()       # 采样宽度 (字节)。
    framerate = wav_file.getframerate()        # 帧率。
    n_frames = wav_file.getnframes()           # 帧数。
    comp_type = wav_file.getcomptype()         # 压缩类型 (仅支持 "NONE")。
    comp_name = wav_file.getcompname()         # 压缩名称。

    # 读取音频数据。
    frames = wav_file.readframes(n_frames)     # 读取 n_frames 个新帧。
    assert len(frames) == sample_width * n_frames

# 复制到新的 WAV 文件。
with wave.open("path_to_new_wav_file.wav", "wb") as wav_file:    # 以写入模式打开 WAV 文件
    # 写入音频数据。
    params = (n_channels, sample_width, framerate, n_frames, comp_type, comp_name)
    wav_file.setparams(params)
    wav_file.writeframes(frames)
```

## 第181.2节：使用Python和mpeg转换任何音频文件

```
from subprocess import check_call

ok = check_call(['ffmpeg', '-i', 'input.mp3', 'output.wav'])
if ok:
    with open('output.wav', 'rb') as f:
        wav_file = f.read()
```

注意：

- <http://superuser.com/questions/507386/why-would-i-choose-libav-over-ffmpeg-or-is-there-even-a-difference>
- [ffmpeg、libav 和 avconv 之间的异同是什么？](#)

## 第181.3节：播放 Windows 的蜂鸣声

Windows 提供了一个明确的接口，通过winsound模块允许你以给定的频率和持续时间播放原始蜂鸣声。

```
import winsound
```

# Chapter 181: Audio

## Section 181.1: Working with WAV files

### winsound

- Windows environment

```
import winsound
winsound.PlaySound("path_to_wav_file.wav", winsound.SND_FILENAME)
```

### wave

- Support mono/stereo
- Doesn't support compression/decompression

```
import wave
with wave.open("path_to_wav_file.wav", "rb") as wav_file:    # Open WAV file in read-only mode.
    # Get basic information.
    n_channels = wav_file.getnchannels()      # Number of channels. (1=Mono, 2=Stereo).
    sample_width = wav_file.getsampwidth()       # Sample width in bytes.
    framerate = wav_file.getframerate()        # Frame rate.
    n_frames = wav_file.getnframes()           # Number of frames.
    comp_type = wav_file.getcomptype()         # Compression type (only supports "NONE").
    comp_name = wav_file.getcompname()         # Compression name.

    # Read audio data.
    frames = wav_file.readframes(n_frames)     # Read n_frames new frames.
    assert len(frames) == sample_width * n_frames

# Duplicate to a new WAV file.
with wave.open("path_to_new_wav_file.wav", "wb") as wav_file:    # Open WAV file in write-only mode.
    # Write audio data.
    params = (n_channels, sample_width, framerate, n_frames, comp_type, comp_name)
    wav_file.setparams(params)
    wav_file.writeframes(frames)
```

## Section 181.2: Convert any soundfile with python and ffmpeg

```
from subprocess import check_call

ok = check_call(['ffmpeg', '-i', 'input.mp3', 'output.wav'])
if ok:
    with open('output.wav', 'rb') as f:
        wav_file = f.read()
```

note:

- <http://superuser.com/questions/507386/why-would-i-choose-libav-over-ffmpeg-or-is-there-even-a-difference>
- [What are the differences and similarities between ffmpeg, libav, and avconv?](#)

## Section 181.3: Playing Windows' beeps

Windows provides an explicit interface through which the `winsound` module allows you to play raw beeps at a given frequency and duration.

```
import winsound
```

```
freq = 2500 # 设置频率为 2500 赫兹  
dur = 1000 # 设置持续时间为 1000 毫秒 == 1 秒  
winsound.Beep(freq, dur)
```

## 第181.4节：使用Pyglet播放音频

```
import pyglet  
audio = pyglet.media.load("audio.wav")  
audio.play()
```

更多信息，请参见[pyglet](#)

```
freq = 2500 # Set frequency To 2500 Hertz  
dur = 1000 # Set duration To 1000 ms == 1 second  
winsound.Beep(freq, dur)
```

## Section 181.4: Audio With Pyglet

```
import pyglet  
audio = pyglet.media.load("audio.wav")  
audio.play()
```

For further information, see [pyglet](#)

# 第182章：pyaudio

PyAudio为PortAudio（跨平台音频输入输出库）提供了Python绑定。使用PyAudio，您可以轻松地在各种平台上使用Python播放和录制音频。PyAudio的灵感来源于：

- 1.pyPortAudio/fastaudio : PortAudio v18 API的Python绑定。
- 2.tkSnack : 适用于Tcl/Tk和Python的跨平台声音工具包。

## 第182.1节：回调模式音频输入输出

"""PyAudio示例：播放波形文件（回调版本）。"""

```
import pyaudio
import wave
import time
import sys

if len(sys.argv) < 2:
    print("播放一个波形文件。用法: %s filename.wav" % sys.argv[0])sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# 实例化 PyAudio (1)
p = pyaudio.PyAudio()

# 定义回调函数 (2)
def callback(in_data, frame_count, time_info, status):
    data = wf.readframes(frame_count)
    return (data, pyaudio.paContinue)

# 使用回调函数打开流 (3)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                 channels=wf.getnchannels(),
                 rate=wf.getframerate(),
                 output=True,
                 stream_callback=callback)

# 启动流 (4)
stream.start_stream()

# 等待流结束 (5)
while stream.is_active():
    time.sleep(0.1)

# 停止流 (6)
stream.stop_stream()
stream.close()
wf.close()

# 关闭 PyAudio (7)
p.terminate()
```

在回调模式下，PyAudio 会在需要新的音频数据（用于播放）和/或有新的（录制的）音频数据可用时调用指定的回调函数（2）。注意，PyAudio 在单独的线程中调用回调函数。该函数的签名如下callback(<input\_data>, <frame\_count>, <time\_info>, <status\_flag>)，且必须返回一个包含frame\_count帧音频数据和一个标志的元组

# Chapter 182: pyaudio

PyAudio provides Python bindings for PortAudio, the cross-platform audio I/O library. With PyAudio, you can easily use Python to play and record audio on a variety of platforms. PyAudio is inspired by:

- 1.pyPortAudio/fastaudio: Python bindings for PortAudio v18 API.
- 2.tkSnack: cross-platform sound toolkit for Tcl/Tk and Python.

## Section 182.1: Callback Mode Audio I/O

```
"""PyAudio Example: Play a wave file (callback version)."""

import pyaudio
import wave
import time
import sys

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# instantiate PyAudio (1)
p = pyaudio.PyAudio()

# define callback (2)
def callback(in_data, frame_count, time_info, status):
    data = wf.readframes(frame_count)
    return (data, pyaudio.paContinue)

# open stream using callback (3)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                 channels=wf.getnchannels(),
                 rate=wf.getframerate(),
                 output=True,
                 stream_callback=callback)

# start the stream (4)
stream.start_stream()

# wait for stream to finish (5)
while stream.is_active():
    time.sleep(0.1)

# stop stream (6)
stream.stop_stream()
stream.close()
wf.close()

# close PyAudio (7)
p.terminate()
```

In callback mode, PyAudio will call a specified callback function (2) whenever it needs new audio data (to play) and/or when there is new (recorded) audio data available. Note that PyAudio calls the callback function in a separate thread. The function has the following signature callback(<input\_data>, <frame\_count>, <time\_info>, <status\_flag>) and must return a tuple containing frame\_count frames of audio data and a flag

表示是否还有更多帧需要播放/录制。

使用pyaudio.Stream.start\_stream() (4) 开始处理音频流，该方法会重复调用回调函数，直到该函数返回pyaudio.paComplete。

为了保持流的活动状态，主线程不能终止，例如，可以通过睡眠 (5) 来实现。

## 第182.2节：阻塞模式音频输入输出

"""PyAudio 示例：播放一个波形文件。"""

```
import pyaudio
import wave
import sys

CHUNK = 1024

if len(sys.argv) < 2:
    print("播放一个波形文件。用法: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# 实例化 PyAudio (1)
p = pyaudio.PyAudio()

# 打开流 (2)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                 channels=wf.getnchannels(),
                 sample_rate=wf.getframerate(),
                 output=True)

# 读取数据
data = wf.readframes(CHUNK)

# 播放流 (3)
while len(data) > 0:
    stream.write(data)
    data = wf.readframes(CHUNK)

# 停止流 (4)
stream.stop_stream()
stream.close()

# 关闭 PyAudio (5)
p.terminate()
```

要使用 PyAudio，首先使用 pyaudio.PyAudio() (1) 实例化 PyAudio，这会设置 portaudio 系统。

要录制或播放音频，请使用

pyaudio.PyAudio.open() (2) 在所需设备上以所需音频参数打开流。这将设置一个pyaudio.Stream用于播放或录制音频。

通过使用pyaudio.Stream.write()向流中写入音频数据来播放音频，或通过pyaudio.Stream.read()从流中读取音频数据。(3)

注意，在“阻塞模式”下，每个pyaudio.Stream.write()或pyaudio.Stream.read()会阻塞，直到所有给定/请求的帧被播放/录制完毕。或者，为了动态生成音频数据或立即处理录制的音频数据，请使用“回调模式”（参见回调模式示例）

signifying whether there are more frames to play/record.

Start processing the audio stream using **pyaudio.Stream.start\_stream()** (4), which will call the callback function repeatedly until that function returns **pyaudio.paComplete**.

To keep the stream active, the main thread must not terminate, e.g., by sleeping (5).

## Section 182.2: Blocking Mode Audio I/O

"""PyAudio Example: Play a wave file."""

```
import pyaudio
import wave
import sys

CHUNK = 1024

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# instantiate PyAudio (1)
p = pyaudio.PyAudio()

# open stream (2)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                 channels=wf.getnchannels(),
                 rate=wf.getframerate(),
                 output=True)

# read data
data = wf.readframes(CHUNK)

# play stream (3)
while len(data) > 0:
    stream.write(data)
    data = wf.readframes(CHUNK)

# stop stream (4)
stream.stop_stream()
stream.close()

# close PyAudio (5)
p.terminate()
```

To use PyAudio, first instantiate PyAudio using **pyaudio.PyAudio()** (1), which sets up the portaudio system.

To record or play audio, open a stream on the desired device with the desired audio parameters using **pyaudio.PyAudio.open()** (2). This sets up a **pyaudio.Stream** to play or record audio.

Play audio by writing audio data to the stream using **pyaudio.Stream.write()**, or read audio data from the stream using **pyaudio.Stream.read()**. (3)

Note that in “blocking mode”，each **pyaudio.Stream.write()** or **pyaudio.Stream.read()** blocks until all the given/requested frames have been played/recorded. Alternatively, to generate audio data on the fly or immediately process recorded audio data, use the “callback mode”(refer the example on call back mode)

使用pyaudio.Stream.stop\_stream()暂停播放/录制，使用pyaudio.Stream.close()终止流。 (4)

最后，使用pyaudio.PyAudio.terminate()终止portaudio会话。 (5)

Use pyaudio.Stream.stop\_stream() to pause playing/recording, and **pyaudio.Stream.close()** to terminate the stream. (4)

Finally, terminate the portaudio session using **pyaudio.PyAudio.terminate()** (5)

# 第183章 : shelve

Shelve是一个用于将对象存储到文件中的Python模块。shelve模块实现了对任意可被pickle的Python对象的持久化存储，使用类似字典的API。当关系型数据库显得过于复杂时，shelve模块可以作为Python对象的简单持久化存储选项。shelf通过键访问，就像字典一样。值被pickle后写入由anydbm创建和管理的数据库中。

## 第183.1节 : 创建新的Shelf

使用shelve最简单的方法是通过DbfilenameShelf类。它使用anydbm来存储数据。你可以直接使用该类，或者简单调用shelve.open()：

```
import shelve

s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'示例数据' }
finally:
    s.close()
```

要再次访问数据，打开shelf并像使用字典一样使用它：

```
import shelve

s = shelve.open('test_shelf.db')
尝试:
existing = s['key1']
finally:
    s.close()

print existing
```

如果运行这两个示例脚本，你应该会看到：

```
$ python shelve_create.py
$ python shelve_existing.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

dbm模块不支持多个应用程序同时写入同一个数据库。如果你知道你的客户端不会修改shelf，你可以告诉shelve以只读方式打开数据库。

```
import shelve

s = shelve.open('test_shelf.db', flag='r')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

如果你的程序尝试在只读打开数据库时修改它，将会产生访问错误异常。异常类型取决于anydbm在创建数据库时选择的数据库模块。

# Chapter 183: shelve

Shelve is a python module used to store objects in a file. The shelve module implements persistent storage for arbitrary Python objects which can be pickled, using a dictionary-like API. The shelve module can be used as a simple persistent storage option for Python objects when a relational database is overkill. The shelf is accessed by keys, just as with a dictionary. The values are pickled and written to a database created and managed by anydbm.

## Section 183.1: Creating a new Shelf

The simplest way to use shelve is via the **DbfilenameShelf** class. It uses anydbm to store the data. You can use the class directly, or simply call **shelve.open()**:

```
import shelve

s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'Sample data' }
finally:
    s.close()
```

To access the data again, open the shelf and use it like a dictionary:

```
import shelve

s = shelve.open('test_shelf.db')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

If you run both sample scripts, you should see:

```
$ python shelve_create.py
$ python shelve_existing.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

The **dbm** module does not support multiple applications writing to the same database at the same time. If you know your client will not be modifying the shelf, you can tell shelve to open the database read-only.

```
import shelve

s = shelve.open('test_shelf.db', flag='r')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

If your program tries to modify the database while it is opened read-only, an access error exception is generated. The exception type depends on the database module selected by anydbm when the database was created.

## 第183.2节：shelve示例代码

要存储一个对象，首先导入模块，然后按如下方式赋值对象：

```
import shelve
database = shelve.open(filename.suffix)
object = Object()
database['key'] = object
```

## 第183.3节：总结接口（key是字符串，data是任意对象）：

```
import shelve

d = shelve.open(filename) # 打开——文件可能会被底层
                         # 库添加后缀

d[key] = data           # 在key处存储数据 (如果
                         # 使用已有key则覆盖旧数据)
data = d[key]           # 获取key处数据的副本 (如果
                         # 没有该key则抛出KeyError)
del d[key]              # 删除key处存储的数据 (如果
                         # 没有该key则抛出KeyError)

flag = key in d         # 如果key存在则为真
klist = list(d.keys())  # 所有现有key的列表 (慢！)

# 由于d是以非writeback=True方式打开，需注意：
d['xx'] = [0, 1, 2]    # 这按预期工作，但...
d['xx'].append(3)       # *这不行！* —— d['xx']仍然是[0, 1, 2]！

# 由于以非writeback=True方式打开d，需谨慎编写代码：
temp = d['xx']          # 提取副本
temp.append(5)           # 修改副本
d['xx'] = temp           # 将副本重新存回，以保持更改

# 或者，使用 d=shelve.open(filename,writeback=True) 你可以直接写代码
# d['xx'].append(5) 并且它会按预期工作，但这也会
# 占用更多内存并使 d.close() 操作变慢。

d.close()               # 关闭它
```

## 第183.4节：写回

默认情况下，货架（shelves）不会跟踪对易失性对象的修改。这意味着如果你更改了存储在货架中的某个项目的内容，必须通过再次存储该项目来显式更新货架。

```
import shelve

s = shelve.open('test_shelf.db')
try:
    print s['key1']
    s['key1']['new_value'] = '之前没有的内容'
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
尝试:
```

## Section 183.2: Sample code for shelve

To shelve an object, first import the module and then assign the object value as follows:

```
import shelve
database = shelve.open(filename.suffix)
object = Object()
database['key'] = object
```

## Section 183.3: To summarize the interface (key is a string, data is an arbitrary object):

```
import shelve

d = shelve.open(filename) # open -- file may get suffix added by low-level
                         # library

d[key] = data           # store data at key (overwrites old data if
                         # using an existing key)
data = d[key]           # retrieve a COPY of data at key (raise KeyError
                         # if no such key)
del d[key]              # delete data stored at key (raises KeyError
                         # if no such key)

flag = key in d         # true if the key exists
klist = list(d.keys())  # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]      # this works as expected, but...
d['xx'].append(3)        # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']          # extracts the copy
temp.append(5)           # mutates the copy
d['xx'] = temp           # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()               # close it
```

## Section 183.4: Write-back

Shelves do not track modifications to volatile objects, by default. That means if you change the contents of an item stored in the shelf, you must update the shelf explicitly by storing the item again.

```
import shelve

s = shelve.open('test_shelf.db')
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
```

```
print s['key1']
finally:
    s.close()
```

在此示例中，键为‘key1’的字典未被再次存储，因此当货架重新打开时，所做的更改未被保存。

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

要自动捕捉存储在shelf中的易变对象的更改，请启用写回（writeback）模式打开shelf。

writeback标志使shelf记住所有从数据库中检索的对象，使用内存缓存。

每个缓存对象在shelf关闭时也会写回数据库。

```
import shelve

s = shelve.open('test_shelf.db', writeback=True)
尝试:
    print s['key1']
s['key1']['new_value'] = '之前没有的内容'
    print s['key1']
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
尝试:
    print s['key1']
finally:
    s.close()
```

虽然它减少了程序员出错的可能性，并且可以使对象持久化更加透明，但在每种情况下使用写回模式可能并不理想。缓存会在shelf打开时消耗额外内存，并且在关闭时暂停将每个缓存对象写回数据库可能需要额外时间。由于无法判断缓存对象是否被修改，所有对象都会被写回。如果您的应用程序读取数据的次数多于写入，写回模式会增加您可能不希望的开销。

```
$ python shelve_create.py
$ python shelve_writeback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
```

```
print s['key1']
finally:
    s.close()
```

In this example, the dictionary at ‘key1’ is not stored again, so when the shelf is re-opened, the changes have not been preserved.

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py

{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

To automatically catch changes to volatile objects stored in the shelf, open the shelf with writeback enabled. The writeback flag causes the shelf to remember all of the objects retrieved from the database using an in-memory cache. Each cache object is also written back to the database when the shelf is closed.

```
import shelve

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
    print s['key1']
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

Although it reduces the chance of programmer error, and can make object persistence more transparent, using writeback mode may not be desirable in every situation. The cache consumes extra memory while the shelf is open, and pausing to write every cached object back to the database when it is closed can take extra time. Since there is no way to tell if the cached objects have been modified, they are all written back. If your application reads data more than it writes, writeback will add more overhead than you might want.

```
$ python shelve_create.py
$ python shelve_writeback.py

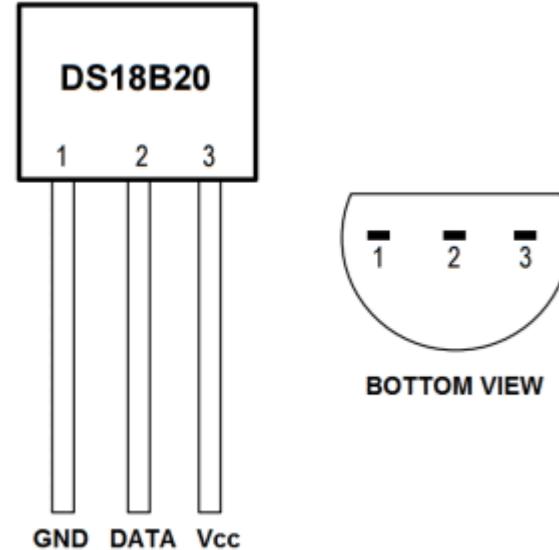
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
```

# 第184章：使用Python和树莓派进行物联网编程

## 第184.1节：示例 - 温度传感器

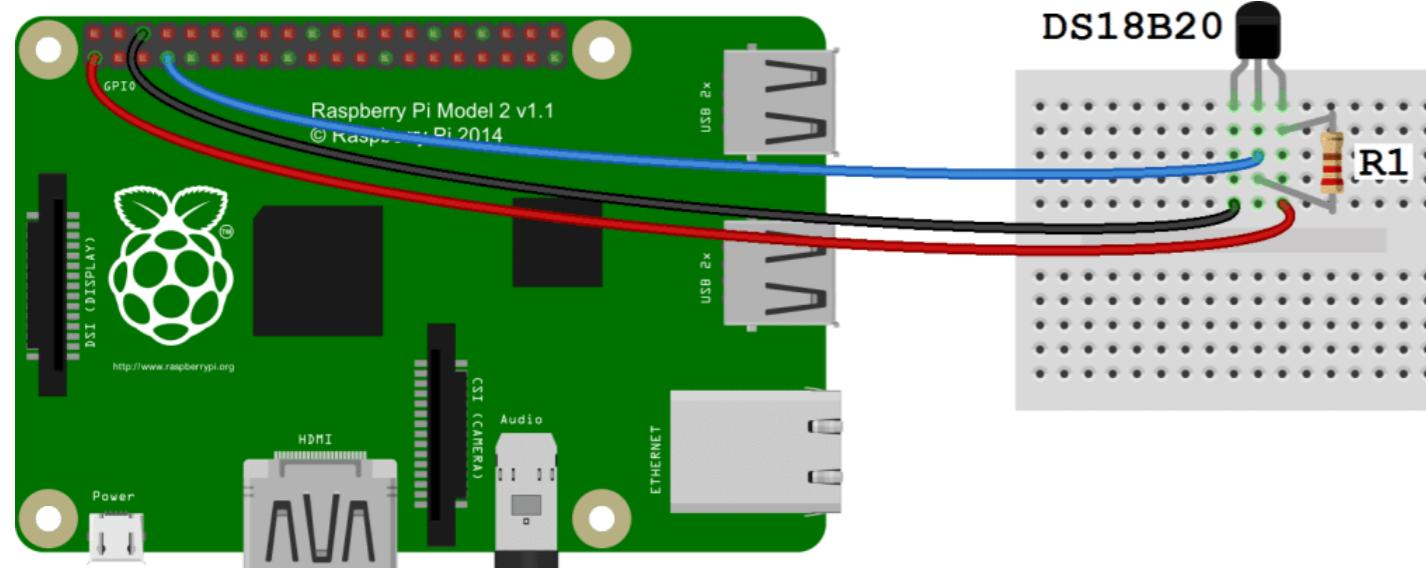
DS18B20与树莓派的接口

DS18B20与树莓派的连接



你可以看到有三个端子

1. Vcc
2. 地线
3. 数据 (单线协议)



fritzing

R1 是用于上拉电压的4.7k欧姆电阻

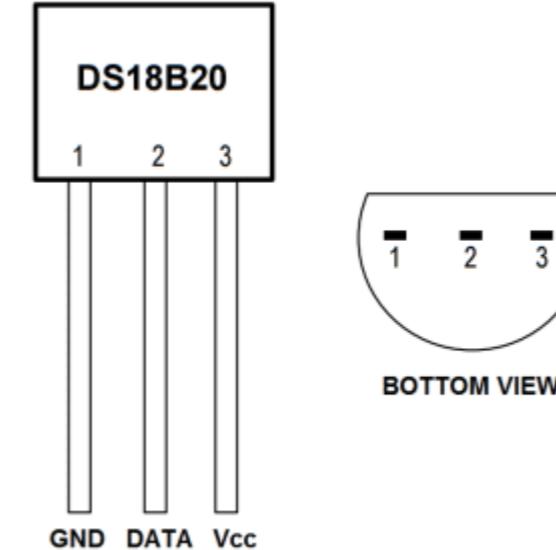
1. Vcc 应连接到树莓派的任一5V或3.3V引脚 (引脚：01, 02, 04, 17)。
2. Gnd 应连接到树莓派的任一地线引脚 (引脚：06, 09, 14, 20, 25)。

# Chapter 184: IoT Programming with Python and Raspberry PI

## Section 184.1: Example - Temperature sensor

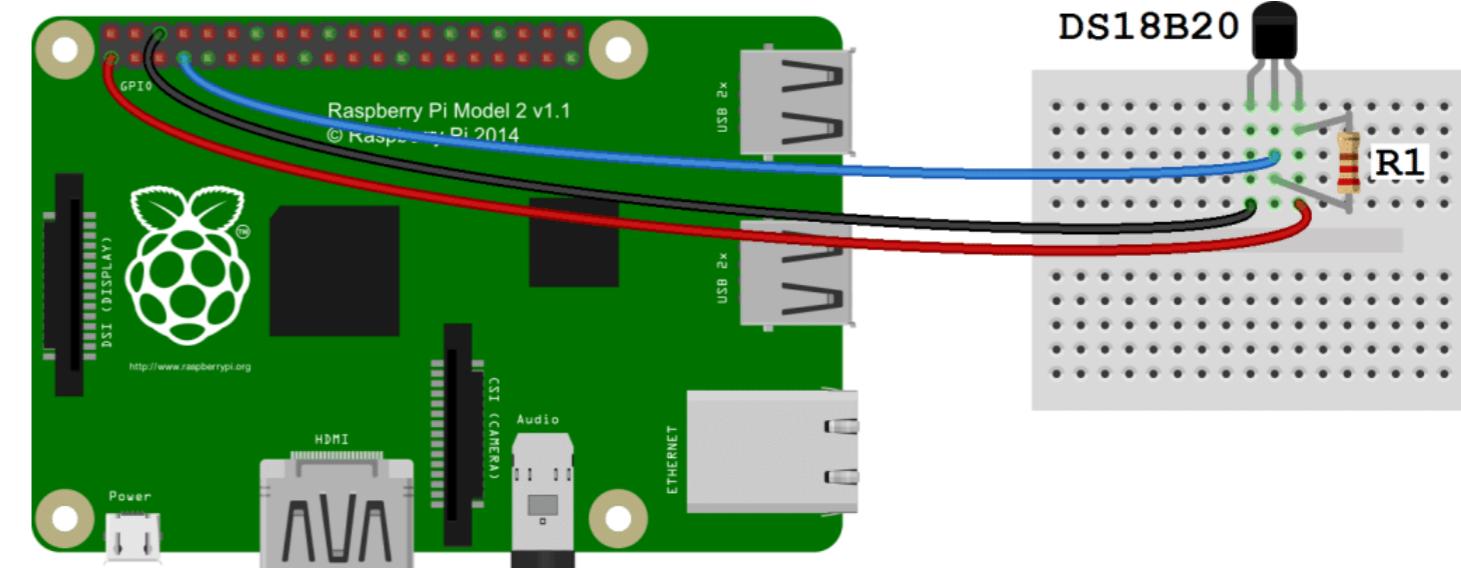
Interfacing of DS18B20 with Raspberry pi

Connection of DS18B20 with Raspberry pi



You can see there are three terminal

1. Vcc
2. Gnd
3. Data (One wire protocol)



fritzing

R1 is 4.7k ohm resistance for pulling up the voltage level

1. Vcc should be connected to any of the 5v or 3.3v pins of Raspberry pi (PIN : 01, 02, 04, 17).
2. Gnd should be connected to any of the Gnd pins of Raspberry pi (PIN : 06, 09, 14, 20, 25).

### 3. DATA 应连接到 (引脚 : 07)

#### 从树莓派端启用单线接口

4. 使用putty或其他Linux/Unix终端登录树莓派。
5. 登录后，在你喜欢的编辑器中打开 /boot/config.txt 文件。

```
nano /boot/config.txt
```

6. 现在在文件末尾添加这一行 dtoverlay=w1-gpio 。
7. 现在重启树莓派 sudo reboot 。
8. 登录树莓派，运行 sudo modprobe g1-gpio
9. 然后运行 sudo modprobe w1-therm
10. 现在进入目录 /sys/bus/w1/devices cd /sys/bus/w1/devices  
11. 现在你会发现一个以 28-\*\*\*\*\* 开头的温度传感器虚拟目录被创建。
12. 进入该目录 cd 28-\*\*\*\*\*
13. 现在有一个文件名为w1-slave，该文件包含温度和其他信息，如CRC。 cat w1-slave。

#### 现在用Python编写一个模块来读取温度

```
import glob
import time

RATE = 30
sensor_dirs = glob.glob("/sys/bus/w1/devices/28*")

if len(sensor_dirs) != 0:
    while True:
        time.sleep(RATE)
        for directories in sensor_dirs:
            temperature_file = open(directories + "/w1_slave")
            # 读取文件
            text = temperature_file.read()
            temperature_file.close()
            # 使用换行符 () 分割文本并选择第二行。
            second_line = text.split("\n")[1]
            # 将该行分割成单词，并选择第十个单词
            temperature_data = second_line.split(" ")[9]
            # 忽略前两个字符后读取数据。
            temperature = float(temperature_data[2:])
            # 现在通过除以1000来归一化温度。
            temperature = temperature / 1000
            print '地址 : '+str(directories.split('/')[-1])+'， 温度 : '+str(temperature)
```

上述Python模块将无限期打印温度与地址的对应关系。RATE参数用于更改或调整从传感器查询温度的频率。

#### GPIO引脚图

[1.[https://www.element14.com/community/servlet/liveServlet/previewBody/73950-102-11-339300/pi3\\_gpio.pn](https://www.element14.com/community/servlet/liveServlet/previewBody/73950-102-11-339300/pi3_gpio.pn)

### 3. DATA should be connected to (PIN : 07)

#### Enabling the one-wire interface from the RPi side

4. Login to Raspberry pi using putty or any other linux/unix terminal.
5. After login, open the /boot/config.txt file in your favourite browser.

```
nano /boot/config.txt
```

6. Now add the this line dtoverlay=w1-gpio to the end of the file.
7. Now reboot the Raspberry pi sudo reboot.
8. Log in to Raspberry pi, and run sudo modprobe g1-gpio
9. Then run sudo modprobe w1-therm
10. Now go to the directory /sys/bus/w1/devices cd /sys/bus/w1/devices
11. Now you will found out a virtual directory created of your temperature sensor starting from 28-\*\*\*\*\*.
12. Go to this directory cd 28-\*\*\*\*\*
13. Now there is a file name w1-slave, This file contains the temperature and other information like CRC. cat w1-slave.

#### Now write a module in python to read the temperature

```
import glob
import time

RATE = 30
sensor_dirs = glob.glob("/sys/bus/w1/devices/28*")

if len(sensor_dirs) != 0:
    while True:
        time.sleep(RATE)
        for directories in sensor_dirs:
            temperature_file = open(directories + "/w1_slave")
            # Reading the files
            text = temperature_file.read()
            temperature_file.close()
            # Split the text with new lines (\n) and select the second line.
            second_line = text.split("\n")[1]
            # Split the line into words, and select the 10th word
            temperature_data = second_line.split(" ")[9]
            # We will read after ignoring first two character.
            temperature = float(temperature_data[2:])
            # Now normalise the temperature by dividing 1000.
            temperature = temperature / 1000
            print 'Address : '+str(directories.split('/')[-1])+'， Temperature : '+str(temperature)
```

Above python module will print the temperature vs address for infinite time. RATE parameter is defined to change or adjust the frequency of temperature query from the sensor.

#### GPIO pin diagram

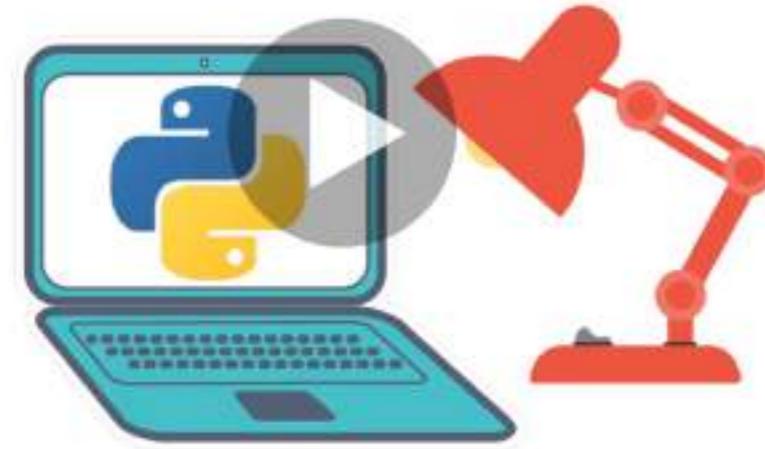
1. [[https://www.element14.com/community/servlet/liveServlet/previewBody/73950-102-11-339300/pi3\\_gpio.pn](https://www.element14.com/community/servlet/liveServlet/previewBody/73950-102-11-339300/pi3_gpio.pn)



# 视频：完整的Python训练营：从零开始成为Python 3高手

像专业人士一样学习Python！从基础开始，直到创建你自己的应用程序和游戏！

## COMPLETE PYTHON 3 BOOTCAMP



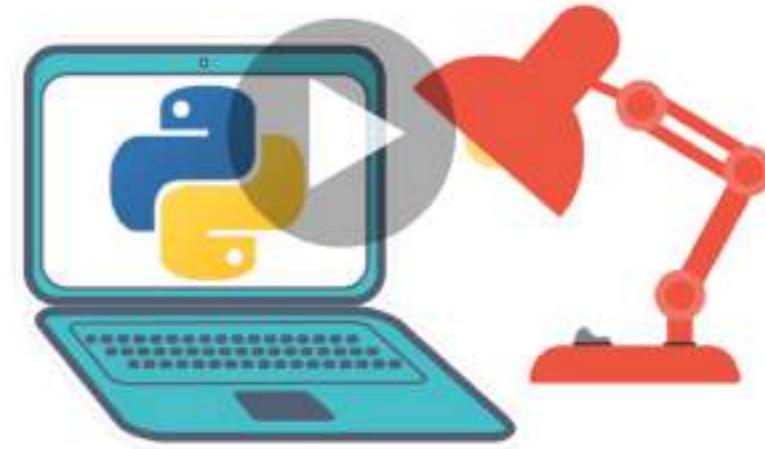
- ✓ 学习专业使用Python，掌握Python 2和Python 3！
- ✓ 用Python创建游戏，比如井字棋和二十一点！
- ✓ 学习高级Python功能，如collections模块和时间戳的使用！
- ✓ 学习使用面向对象编程和类！
- ✓ 理解复杂主题，如装饰器。
- ✓ 理解如何使用Jupyter Notebook并创建.py文件
- ✓ 了解如何在Jupyter Notebook系统中创建图形用户界面（GUI）！
- ✓ 从零开始全面掌握Python！

今天观看 →

# VIDEO: Complete Python Bootcamp: Go from zero to hero in Python 3

Learn Python like a Professional! Start from the basics and go all the way to creating your own applications and games!

## COMPLETE PYTHON 3 BOOTCAMP



- ✓ Learn to use Python professionally, learning both Python 2 and Python 3!
- ✓ Create games with Python, like Tic Tac Toe and Blackjack!
- ✓ Learn advanced Python features, like the collections module and how to work with timestamps!
- ✓ Learn to use Object Oriented Programming with classes!
- ✓ Understand complex topics, like decorators.
- ✓ Understand how to use both the Jupyter Notebook and create .py files
- ✓ Get an understanding of how to create GUIs in the Jupyter Notebook system!
- ✓ Build a complete understanding of Python from the ground up!

Watch Today →

# 第185章：kivy - 跨平台Python 用于NUI开发的框架

NUI：自然用户界面（NUI）是一种人机交互系统，用户通过与自然、日常人类行为相关的直观动作进行操作。

Kivy是一个Python库，用于开发支持多点触控的多媒体丰富应用程序，可安装在不同设备上。多点触控指的是触摸感应表面（通常是触摸屏或触控板）能够同时检测或感知两个或多个接触点的输入能力。

## 第185.1节：第一个应用程序

创建一个 kivy 应用程序

1. 子类化app类
2. 实现build方法，该方法将返回控件。
3. 实例化该类并调用 run。

```
from kivy.app import App
from kivy.uix.label import Label

class Test(App):
    def build(self):
        return Label(text='Hello world')

if __name__ == '__main__':
    Test().run()
```

### 说明

```
from kivy.app import App
```

上述语句将导入父类**app**。该类位于您的安装目录  
your\_installtion\_directory/kivy/app.py中

```
from kivy.uix.label import Label
```

上述语句将导入 ux 元素**Label**。所有的 ux 元素都存在于你的安装目录  
your\_installation\_directory/kivy/uix/中。

```
class Test(App):
```

上述语句用于创建你的应用程序，类名将作为你的应用名称。该类继承了父类 App 类。

```
def build(self):
```

上述语句重写了 App 类的 build 方法。该方法将返回在启动应用时需要显示的控件。

```
return Label(text='Hello world')
```

上述语句是 build 方法的主体。它返回一个文本为Hello world的 Label。

# Chapter 185: kivy - Cross-platform Python Framework for NUI Development

NUI : A natural user interface (NUI) is a system for human-computer interaction that the user operates through intuitive actions related to natural, everyday human behavior.

Kivy is a Python library for development of multi-touch enabled media rich applications which can be installed on different devices. Multi-touch refers to the ability of a touch-sensing surface (usually a touch screen or a trackpad) to detect or sense input from two or more points of contact simultaneously.

## Section 185.1: First App

To create an kivy application

1. sub class the **app** class
2. Implement the **build** method, which will return the widget.
3. Instantiate the class an invoke the **run**.

```
from kivy.app import App
from kivy.uix.label import Label

class Test(App):
    def build(self):
        return Label(text='Hello world')

if __name__ == '__main__':
    Test().run()
```

### Explanation

```
from kivy.app import App
```

The above statement will import the parent class **app**. This will be present in your installation directory  
your\_installtion\_directory/kivy/app.py

```
from kivy.uix.label import Label
```

The above statement will import the ux element **Label**. All the ux element are present in your installation directory  
your\_installation\_directory/kivy/uix/.

```
class Test(App):
```

The above statement is for to create your app and class name will be your app name. This class is inherited the parent app class.

```
def build(self):
```

The above statement override the build method of app class. Which will return the widget that needs to be shown when you will start the app.

```
return Label(text='Hello world')
```

The above statement is the body of the build method. It is returning the Label with its text **Hello world**.

```
if __name__ == '__main__':
```

上述语句是 Python 解释器开始执行你的应用程序的入口点。

```
Test().run()
```

上述语句通过创建 Test 类的实例来初始化它，并调用 App 类的 run() 方法。

您的应用程序将如下图所示。



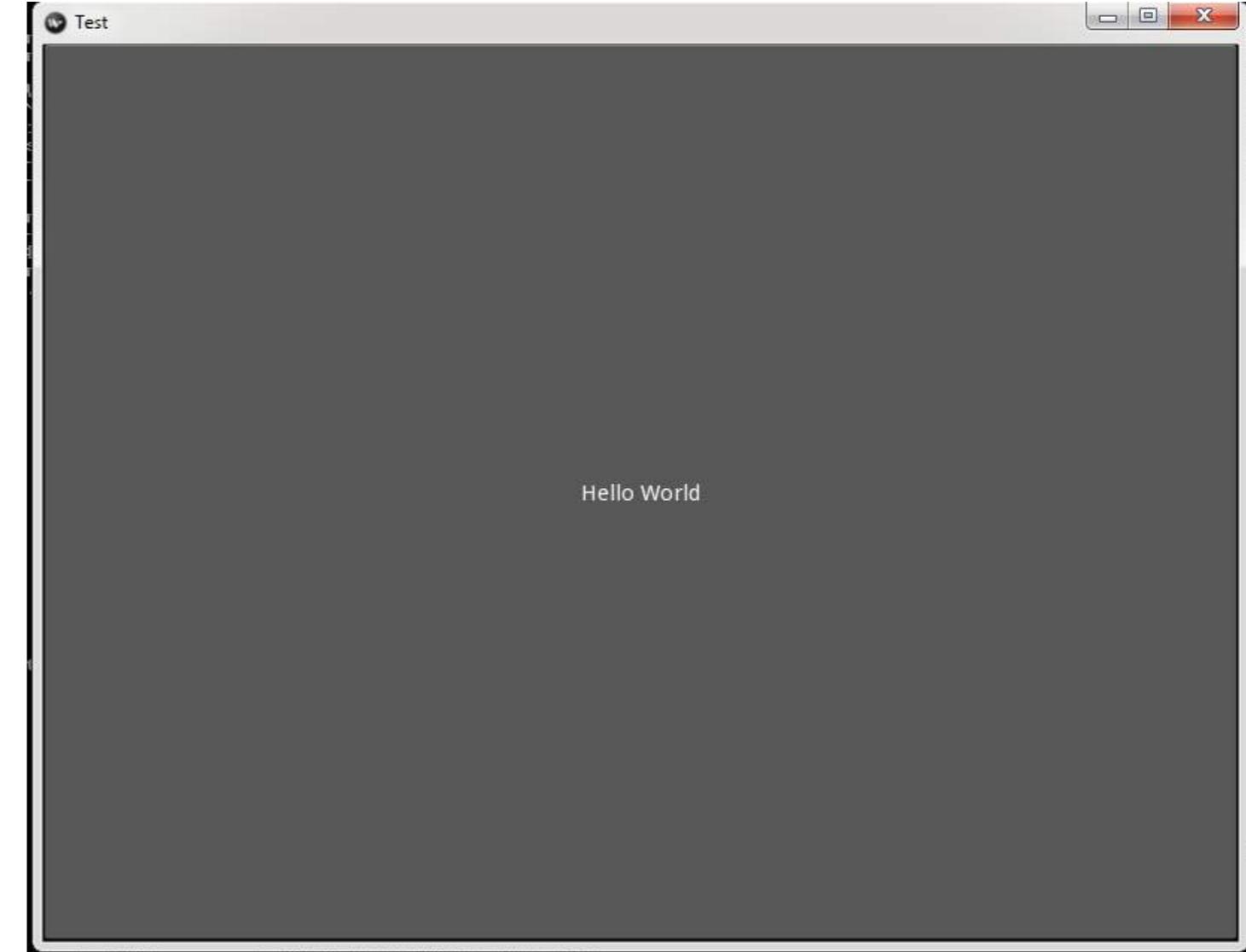
```
if __name__ == '__main__':
```

The above statement is the entry point from where python interpreter start executing your app.

```
Test().run()
```

The above statement Initialise your Test class by creating its instance. And invoke the app class function run().

Your app will look like the below picture.



# 第186章：Pandas Transform：对分组执行操作并连接结果

## 第186.1节：简单转换

首先，让我们创建一个示例数据框

我们假设一个客户可以有n个订单，一个订单可以有m个商品，且商品可以被多次订购

```
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3, 3]
orders_df['order_id'] = [1, 1, 1, 2, 2, 3, 3, 4, 5, 6, 6, 6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                    'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry', 'strawberry']
```

# 这是数据框的样子：

```
print(orders_df)
#   customer_id  order_id      item
# 0            1        1    苹果
# 1            1        1  巧克力
# 2            1        1  巧克力
# 3            1        2    咖啡
# 4            1        2    咖啡
# 5            2        3    苹果
# 6            2        3  香蕉
# 7            3        4    咖啡
# 8            3        5  奶昔
# 9            3        6  巧克力
# 10           3        6  草莓
# 11           3        6  草莓
```

现在，我们将使用 pandas 的 transform 函数来计算每个客户的订单数量

# 首先，我们定义一个函数，该函

数将应用于每个 customer\_id

```
# 现在，我们可以使用上述定义的逻辑对每个分组进行转换
orders_df['number_of_orders_per_client'] = (          # 将结果放入一个名为 'number_of_orders_per_client' 的新列中
    orders_df.groupby(['customer_id'])['order_i
d'] # 为每个 customer_id 创建一个单独的分组并选择 order_id
    .transform(count_number_of_orders)) # 对每个分组单独应用该函数
```

```
# 检查结果 ...
print(orders_df)
#   customer_id  order_id      item  number_of_orders_per_client
# 0            1        1    apples                2
# 1            1        1  chocolate               2
# 2            1        1  chocolate               2
# 3            1        2    coffee                2
# 4            1        2    coffee                2
```

# Chapter 186: Pandas Transform: Perform operations on groups and concatenate the results

## Section 186.1: Simple transform

First, Let's create a dummy dataframe

We assume that a customer can have n orders, an order can have m items, and items can be ordered more multiple times

```
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3, 3]
orders_df['order_id'] = [1, 1, 1, 2, 2, 3, 3, 4, 5, 6, 6, 6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                    'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry', 'strawberry']
```

# And this is how the dataframe looks like:

```
print(orders_df)
#   customer_id  order_id      item
# 0            1        1    apples
# 1            1        1  chocolate
# 2            1        1  chocolate
# 3            1        2    coffee
# 4            1        2    coffee
# 5            2        3    apples
# 6            2        3  bananas
# 7            3        4    coffee
# 8            3        5  milkshake
# 9            3        6  chocolate
# 10           3        6  strawberry
# 11           3        6  strawberry
```

Now, we will use pandas transform function to count the number of orders per customer

```
# First, we define the function that will be applied per customer_id
count_number_of_orders = lambda x: len(x.unique())
```

```
# And now, we can transform each group using the logic defined above
orders_df['number_of_orders_per_client'] = (          # Put the results into a new column that
    is called 'number_of_orders_per_client'
    orders_df # Take the original dataframe
    .groupby(['customer_id'])['order_id'] # Create a separate group for each
    customer_id & select the order_id
    .transform(count_number_of_orders)) # Apply the function to each group
separately
```

# Inspecting the results ...

```
print(orders_df)
#   customer_id  order_id      item  number_of_orders_per_client
# 0            1        1    apples                2
# 1            1        1  chocolate               2
# 2            1        1  chocolate               2
# 3            1        2    coffee                2
# 4            1        2    coffee                2
```

```

# 5      2      3      apples      1
# 6      2      3      bananas     1
# 7      3      4      coffee      3
# 8      3      5      milkshake   3
# 9      3      6      chocolate   3
# 10     3      6      strawberry  3
# 11     3      6      strawberry  3

```

## 第186.2节：每组的多个结果

使用transform函数返回每个组的子计算结果在前面的例子中，我们得到每个客户一个结果。然而，也可以应用返回组内不同值的函数。

```

# 创建一个虚拟数据框
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3, 3]
orders_df['order_id'] = [1, 1, 1, 2, 2, 3, 3, 4, 5, 6, 6, 6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                     'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry', 'strawberry']

```

# 让我们尝试查看每个订单中商品是否被多次订购

```

# 首先，我们定义一个将应用于每个组的函数
def 每个订单多件商品(_items):
    # 应用.duplicated, 若商品出现多次则返回True。
    多件商品布尔值 = _items.duplicated(keep=False)
    return(多件商品布尔值)

# 然后，根据定义的函数转换每个组
orders_df['item_duplicated_per_order'] = (
    orders_df
    .groupby(['order_id'])['item']      # 为每个order_id创建单独组并选择item
    .transform(multiple_items_per_order)) # 对每个分组分别应用定义的函数

```

```

# 检查结果 ...
print(orders_df)
#   customer_id  order_id      item  item_duplicated_per_order
# 0            1       1    苹果        False
# 1            1       1  巧克力       True
# 2            1       1  巧克力       True
# 3            1       2    咖啡       True
# 4            1       2    咖啡       True
# 5            2       3    苹果        False
# 6            2       3  香蕉        False
# 7            3       4    咖啡        False
# 8            3       5   奶昔        False
# 9            3       6  巧克力       False
# 10           3       6  草莓       True
# 11           3       6  草莓       True

```

```

# 5      2      3      apples      1
# 6      2      3      bananas     1
# 7      3      4      coffee      3
# 8      3      5      milkshake   3
# 9      3      6      chocolate   3
# 10     3      6      strawberry  3
# 11     3      6      strawberry  3

```

## Section 186.2: Multiple results per group

### Using transform functions that return sub-calculations per group

In the previous example, we had one result per client. However, functions returning different values for the group can also be applied.

```

# Create a dummy dataframe
orders_df = pd.DataFrame()
orders_df['customer_id'] = [1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3, 3]
orders_df['order_id'] = [1, 1, 1, 2, 2, 3, 3, 4, 5, 6, 6, 6]
orders_df['item'] = ['apples', 'chocolate', 'chocolate', 'coffee', 'coffee', 'apples',
                     'bananas', 'coffee', 'milkshake', 'chocolate', 'strawberry', 'strawberry']

```

# Let's try to see if the items were ordered more than once in each orders

```

# First, we define a function that will be applied per group
def multiple_items_per_order(_items):
    # Apply .duplicated, which will return True if the item occurs more than once.
    multiple_item_bool = _items.duplicated(keep=False)
    return(multiple_item_bool)

# Then, we transform each group according to the defined function
orders_df['item_duplicated_per_order'] = (
    orders_df
    .groupby(['order_id'])['item']      # Take the orders dataframe
    .transform(multiple_items_per_order)) # Create a separate group for each
                                         # Put the results into a new column
                                         # Order_id & select the item
                                         # transform(multiple_items_per_order)) # Apply the defined function to each
                                         # group separately

```

# Inspecting the results ...

```

print(orders_df)
#   customer_id  order_id      item  item_duplicated_per_order
# 0            1       1    apples        False
# 1            1       1  chocolate      True
# 2            1       1  chocolate      True
# 3            1       2    coffee       True
# 4            1       2    coffee       True
# 5            2       3    apples        False
# 6            2       3  bananas        False
# 7            3       4    coffee        False
# 8            3       5  milkshake     False
# 9            3       6  chocolate      False
# 10           3       6  strawberry     True
# 11           3       6  strawberry     True

```

# 第187章：语法上的相似性， 意义上的差异：Python 与 JavaScript

有时两种语言会对相同或相似的语法表达赋予不同的含义。当程序员对这两种语言都感兴趣时，澄清这些分歧点有助于更好地理解两种语言的基础和细微差别。

## 第187.1节：列表中的`in`

2 in [2, 3]

在Python中，这个表达式的结果为True，但在JavaScript中为false。这是因为Python中的in用于检查一个值是否包含在列表中，所以2是[2, 3]的第一个元素。而在JavaScript中，in用于对象，检查对象是否包含由值表示的属性名。因此，JavaScript将[2, 3]视为一个对象或键值映射，如下所示：

{'0': 2, '1': 3}

并检查它是否有属性或键名为'2'。整数2会被自动转换为字符串'2'。

# Chapter 187: Similarities in syntax, Differences in meaning: Python vs. JavaScript

It sometimes happens that two languages put different meanings on the same or similar syntax expression. When the both languages are of interest for a programmer, clarifying these bifurcation points helps to better understand the both languages in their basics and subtleties.

## Section 187.1: `in` with lists

2 in [2, 3]

In Python this evaluates to True, but in JavaScript to false. This is because in Python in checks if a value is contained in a list, so 2 is in [2, 3] as its first element. In JavaScript in is used with objects and checks if an object contains the property with the name expressed by the value. So JavaScript considers [2, 3] as an object or a key-value map like this:

{'0': 2, '1': 3}

and checks if it has a property or a key '2' in it. Integer 2 is silently converted to string '2'.

# 第188章：从C#调用Python

文档提供了C#与  
Python脚本之间进程间通信的示例实现。

## 第188.1节：由C#应用程序调用的Python脚本

```
import sys
import json

# 从文本文件加载输入参数
filename = sys.argv[ 1 ]
with open( filename ) as data_file:
    input_args = json.loads( data_file.read() )

# 将字符串转换为浮点数
x, y = [ float(input_args.get( key )) for key in [ 'x', 'y' ] ]

print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```

## 第188.2节：调用Python脚本的C#代码

```
using MongoDB.Bson;
using System;
using System.Diagnostics;
using System.IO;

namespace python_csharp
{
    class Program
    {
        static void Main(string[] args)
        {
            // .py 文件的完整路径
            string pyScriptPath = "...../sum.py";
            // 将输入参数转换为 JSON 字符串
            BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");

            bool saveInputFile = false;

            string argsFile = string.Format("{0}\\{1}.txt", Path.GetDirectoryName(pyScriptPath),
Guid.NewGuid());

            string outputString = null;
            // 创建新的进程启动信息
            ProcessStartInfo prcStartInfo = new ProcessStartInfo
            {
                // Python解释器"python.exe"的完整路径
                FileName = "python.exe", // string.Format(@"""{0}""", "python.exe"),
                UseShellExecute = false,
                RedirectStandardOutput = true,
                CreateNoWindow = false
            };

            try
            {
                // 将输入参数写入.txt文件
                using (StreamWriter sw = new StreamWriter(argsFile))
```

# Chapter 188: Call Python from C#

The documentation provides a sample implementation of the inter-process communication between C# and Python scripts.

## Section 188.1: Python script to be called by C# application

```
import sys
import json

# load input arguments from the text file
filename = sys.argv[ 1 ]
with open( filename ) as data_file:
    input_args = json.loads( data_file.read() )

# cast strings to floats
x, y = [ float(input_args.get( key )) for key in [ 'x', 'y' ] ]

print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```

## Section 188.2: C# code calling Python script

```
using MongoDB.Bson;
using System;
using System.Diagnostics;
using System.IO;

namespace python_csharp
{
    class Program
    {
        static void Main(string[] args)
        {
            // full path to .py file
            string pyScriptPath = "...../sum.py";
            // convert input arguments to JSON string
            BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");

            bool saveInputFile = false;

            string argsFile = string.Format("{0}\\{1}.txt", Path.GetDirectoryName(pyScriptPath),
Guid.NewGuid());

            string outputString = null;
            // create new process start info
            ProcessStartInfo prcStartInfo = new ProcessStartInfo
            {
                // full path of the Python interpreter 'python.exe'
                FileName = "python.exe", // string.Format(@"""{0}""", "python.exe"),
                UseShellExecute = false,
                RedirectStandardOutput = true,
                CreateNoWindow = false
            };

            try
            {
                // write input arguments to .txt file
                using (StreamWriter sw = new StreamWriter(argsFile))
```

```

    {
sw.WriteLine(argsBson);
prcStartInfo.Arguments = string.Format("{0} {1}", string.Format(@"""{0}""", 
pyScriptPath), string.Format(@"""{0}""", argsFile));
    }
    // 启动进程
    using (Process process = Process.Start(prcStartInfo))
    {
        // 读取标准输出的 JSON 字符串
        using (StreamReader myStreamReader = process.StandardOutput)
        {
outputString = myStreamReader.ReadLine();
            process.WaitForExit();
        }
    }
    finally
    {
        // 删除/保存临时 .txt 文件
        if (!saveInputFile)
        {
File.Delete(argsFile);
        }
    }
Console.WriteLine(outputString);
}
}

```

```

    {
        sw.WriteLine(argsBson);
        prcStartInfo.Arguments = string.Format("{0} {1}", string.Format(@"""{0}""", 
pyScriptPath), string.Format(@"""{0}""", argsFile));
    }
    // start process
    using (Process process = Process.Start(prcStartInfo))
    {
        // read standard output JSON string
        using (StreamReader myStreamReader = process.StandardOutput)
        {
outputString = myStreamReader.ReadLine();
            process.WaitForExit();
        }
    }
    finally
    {
        // delete/save temporary .txt file
        if (!saveInputFile)
        {
            File.Delete(argsFile);
        }
    }
    Console.WriteLine(outputString);
}
}

```

# 第189章：ctypes

ctypes 是一个 Python 内置库，用于调用本地编译库中导出的函数。

注意：由于该库处理的是编译代码，因此相对依赖操作系统。

## 第189.1节：ctypes 数组

正如任何优秀的 C 程序员所知道的，单个值远远不够。真正让我们起步的是数组！

```
>>> c_int * 16
<class '__main__.c_long_Array_16'>
```

这不是一个真正的数组，但非常接近！我们创建了一个表示包含16个int的数组的类。

现在我们所需要做的就是初始化它：

```
>>> arr = (c_int * 16)(*range(16))
>>> arr
<__main__.c_long_Array_16 object at 0xbaddcafe>
```

现在arr是一个真正的数组，包含从0到15的数字。

它们可以像任何列表一样访问：

```
>>> arr[5]
5
>>> arr[5] = 20
>>> arr[5]
20
```

就像任何其他ctypes对象一样，它也有大小和位置：

```
>>> sizeof(arr)
64 # sizeof(c_int) * 16
>>> hex(addressof(arr))
'0xc00010ff'
```

## 第189.2节：ctypes的函数封装

在某些情况下，C函数接受一个函数指针。作为热衷的ctypes用户，我们希望使用这些函数，甚至将Python函数作为参数传递。

让我们定义一个函数：

```
>>> def max(x, y):
    return x if x >= y else y
```

现在，该函数接受两个参数并返回相同类型的结果。为了举例说明，假设该类型是int。

就像我们在数组示例中所做的那样，我们可以定义一个表示该原型的对象：

```
>>> CFUNCTYPE(c_int, c_int, c_int)
```

# Chapter 189: ctypes

ctypes is a python built-in library that invokes exported functions from native compiled libraries.

**Note:** Since this library handles compiled code, it is relatively OS dependent.

## Section 189.1: ctypes arrays

As any good C programmer knows, a single value won't get you that far. What will really get us going are arrays!

```
>>> c_int * 16
<class '__main__.c_long_Array_16'>
```

This is not an actual array, but it's pretty darn close! We created a class that denotes an array of 16 ints.

Now all we need to do is to initialize it:

```
>>> arr = (c_int * 16)(*range(16))
>>> arr
<__main__.c_long_Array_16 object at 0xbaddcafe>
```

Now arr is an actual array that contains the numbers from 0 to 15.

They can be accessed just like any list:

```
>>> arr[5]
5
>>> arr[5] = 20
>>> arr[5]
20
```

And just like any other ctypes object, it also has a size and a location:

```
>>> sizeof(arr)
64 # sizeof(c_int) * 16
>>> hex(addressof(arr))
'0xc00010ff'
```

## Section 189.2: Wrapping functions for ctypes

In some cases, a C function accepts a function pointer. As avid ctypes users, we would like to use those functions, and even pass python function as arguments.

Let's define a function:

```
>>> def max(x, y):
    return x if x >= y else y
```

Now, that function takes two arguments and returns a result of the same type. For the sake of the example, let's assume that type is an int.

Like we did on the array example, we can define an object that denotes that prototype:

```
>>> CFUNCTYPE(c_int, c_int, c_int)
```

```
<CFunctionType object at 0xdeadbeef>
```

该原型表示一个返回c\_int (第一个参数) 的函数，并接受两个c\_int参数 (其他参数)。

现在让我们来包装这个函数：

```
>>> CFUNCTYPE(c_int, c_int, c_int)(max)
<CFunctionType object at 0xdeadbeef>
```

函数原型还有一个用途：它们可以包装ctypes函数（如libc.ntohl），并在调用函数时验证使用了正确的参数。

```
>>> libc.ntohl() # 垃圾进 - 垃圾出
>>> CFUNCTYPE(c_int, c_int)(libc.ntohl)()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 此函数至少需要1个参数 (给定了0个)
```

## 第189.3节：基本用法

假设我们想使用libc的ntohl函数。

首先，我们必须加载libc.so：

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle baadf00d at 0xdeadbeef>
```

然后，我们获取函数对象：

```
>>> ntohs = libc.ntohl
>>> ntohs
<_FuncPtr object at 0xbaadf00d>
```

现在，我们可以简单地调用该函数：

```
>>> ntohs(0x6C)
1811939328
>>> hex(_)
'0x6c000000'
```

这正是我们期望它执行的操作。

## 第189.4节：常见陷阱

### 无法加载文件

第一个可能的错误是加载库失败。在这种情况下，通常会引发OSError。

这要么是因为文件不存在（或者操作系统找不到该文件）：

```
>>> cdll.LoadLibrary("foobar.so")
追踪 (最近的调用最后一次) :
文件 "<stdin>", 第 1 行, 在 <module>
```

```
<CFunctionType object at 0xdeadbeef>
```

That prototype denotes a function that returns an c\_int (the first argument), and accepts two c\_int arguments (the other arguments).

Now let's wrap the function:

```
>>> CFUNCTYPE(c_int, c_int, c_int)(max)
<CFunctionType object at 0xdeadbeef>
```

Function prototypes have one more usage: They can wrap ctypes function (like libc.ntohl) and verify that the correct arguments are used when invoking the function.

```
>>> libc.ntohl() # garbage in - garbage out
>>> CFUNCTYPE(c_int, c_int)(libc.ntohl)()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: this function takes at least 1 argument (0 given)
```

## Section 189.3: Basic usage

Let's say we want to use libc's ntohs function.

First, we must load libc.so:

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle baadf00d at 0xdeadbeef>
```

Then, we get the function object:

```
>>> ntohs = libc.ntohl
>>> ntohs
<_FuncPtr object at 0xbaadf00d>
```

And now, we can simply invoke the function:

```
>>> ntohs(0x6C)
1811939328
>>> hex(_)
'0x6c000000'
```

Which does exactly what we expect it to do.

## Section 189.4: Common pitfalls

### Failing to load a file

The first possible error is failing to load the library. In that case an OSError is usually raised.

This is either because the file doesn't exist (or can't be found by the OS):

```
>>> cdll.LoadLibrary("foobar.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
文件 "/usr/lib/python3.5/ctypes/__init__.py", 第 425 行, 在 LoadLibrary
    返回 self._dlltype(name)
文件 "/usr/lib/python3.5/ctypes/__init__.py", 第 347 行, 在 __init__
    self._handle = _dlopen(self._name, mode)
OSErr: foobar.so: 无法 打开 共享 对象文件: 没有那个 文件 或 目录
```

如你所见，错误信息清晰且具有很强的指示性。

第二个原因是文件被找到，但格式不正确。

```
>>> cdll.LoadLibrary("libc.so")
追踪 (最近的调用最后) :
文件 "<stdin>", 第 1 行, 在 <模块>
文件 "/usr/lib/python3.5/ctypes/__init__.py", 第 425 行, 在 LoadLibrary
    返回 self._dlltype(name)
文件 "/usr/lib/python3.5/ctypes/__init__.py", 第 347 行, 在 __init__
    self._handle = _dlopen(self._name, mode)
OSErr: /usr/lib/i386-linux-gnu/libc.so: 无效的 ELF 头
```

在这种情况下，文件是一个脚本文件，而不是一个 .so 文件。当尝试在 Linux 机器上打开一个 .dll 文件，或者在 32 位 Python 解释器上打开一个 64 位文件时，也可能发生这种情况。如你所见，这种情况下错误信息比较模糊，需要进行一些调查。

## 访问函数失败

假设我们成功加载了 .so 文件，那么接下来需要像第一个

示例中那样访问我们的函数。

当使用不存在的函数时，会引发一个 `AttributeError`：

```
>>> libc.foo
追踪 (最近一次调用最后) :
文件 "<stdin>", 第 1 行, 在 <模块>
文件 "/usr/lib/python3.5/ctypes/__init__.py", 第 360 行, 在 __getattr__
    func = self.__getitem__(name)
文件 "/usr/lib/python3.5/ctypes/__init__.py", 第 365 行, 在 __getitem__
    func = self._FuncPtr((name_or_ordinal, self))
AttributeError: /lib/i386-linux-gnu/libc.so.6: 未定义的符号: foo
```

## 第189.5节：基本的ctypes对象

最基本的对象是一个int：

```
>>> obj = ctypes.c_int(12)
>>> obj
c_long(12)
```

现在，`obj` 指向一块包含值12的内存。

该值可以被直接访问，甚至可以被修改：

```
>>> obj.value
12
>>> obj.value = 13
>>> obj
c_long(13)
```

```
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSErr: foobar.so: cannot open shared object file: No such file or directory
```

As you can see, the error is clear and pretty indicative.

The second reason is that the file is found, but is not of the correct format.

```
>>> cdll.LoadLibrary("libc.so")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 347, in __init__
    self._handle = _dlopen(self._name, mode)
OSErr: /usr/lib/i386-linux-gnu/libc.so: invalid ELF header
```

In this case, the file is a script file and not a .so file. This might also happen when trying to open a .dll file on a Linux machine or a 64bit file on a 32bit python interpreter. As you can see, in this case the error is a bit more vague, and requires some digging around.

## Failing to access a function

Assuming we successfully loaded the .so file, we then need to access our function like we've done on the first example.

When a non-existing function is used, an `AttributeError` is raised:

```
>>> libc.foo
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 360, in __getattr__
    func = self.__getitem__(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 365, in __getitem__
    func = self._FuncPtr((name_or_ordinal, self))
AttributeError: /lib/i386-linux-gnu/libc.so.6: undefined symbol: foo
```

## Section 189.5: Basic ctypes object

The most basic object is an int:

```
>>> obj = ctypes.c_int(12)
>>> obj
c_long(12)
```

Now, `obj` refers to a chunk of memory containing the value 12.

That value can be accessed directly, and even modified:

```
>>> obj.value
12
>>> obj.value = 13
>>> obj
c_long(13)
```

由于obj指的是一块内存，我们也可以找出它的大小和位置：

```
>>> sizeof(obj)
4
>>> hex(addressof(obj))
'0xdeadbeef'
```

## 第189.6节：复杂用法

让我们将以上所有示例结合成一个复杂场景：使用libc的lfind函数。

有关该函数的更多细节，请阅读[the man page](#)。我建议你在继续之前先阅读它。

首先，我们定义正确的原型：

```
>>> compar_proto = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> lfind_proto = CFUNCTYPE(c_void_p, c_void_p, c_void_p, POINTER(c_uint), c_uint, compar_proto)
```

然后，让我们创建变量：

```
>>> key = c_int(12)
>>> arr = (c_int * 16)(*range(16))
>>> nmemb = c_uint(16)
```

现在我们定义比较函数：

```
>>> def compar(x, y):
    return x.contents.value - y.contents.value
```

注意到 x 和 y 是 POINTER(c\_int)，因此我们需要对它们进行解引用并取值，才能实际比较存储在内存中的值。

现在我们可以将所有内容组合在一起：

```
>>> lfind = lfind_proto(libc.lfind)
>>> ptr = lfind(byref(key), byref(arr), byref(nmemb), sizeof(c_int), compar_proto(compar))
```

ptr 是返回的 void 指针。如果 key 没有在 arr 中找到，值将是 None，但在此情况下我们得到了一个有效的值。

现在我们可以转换它并访问该值：

```
>>> cast(ptr, POINTER(c_int)).contents
c_long(12)
```

此外，我们可以看到 ptr 指向 arr 中的正确值：

```
>>> addressof(arr) + 12 * sizeof(c_int) == ptr
True
```

Since obj refers to a chunk of memory, we can also find out its size and location:

```
>>> sizeof(obj)
4
>>> hex(addressof(obj))
'0xdeadbeef'
```

## Section 189.6: Complex usage

Let's combine all of the examples above into one complex scenario: using libc's lfind function.

For more details about the function, read [the man page](#). I urge you to read it before going on.

First, we'll define the proper prototypes:

```
>>> compar_proto = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> lfind_proto = CFUNCTYPE(c_void_p, c_void_p, c_void_p, POINTER(c_uint), c_uint, compar_proto)
```

Then, let's create the variables:

```
>>> key = c_int(12)
>>> arr = (c_int * 16)(*range(16))
>>> nmemb = c_uint(16)
```

And now we define the comparison function:

```
>>> def compar(x, y):
    return x.contents.value - y.contents.value
```

Notice that x, and y are POINTER(c\_int), so we need to dereference them and take their values in order to actually compare the value stored in the memory.

Now we can combine everything together:

```
>>> lfind = lfind_proto(libc.lfind)
>>> ptr = lfind(byref(key), byref(arr), byref(nmemb), sizeof(c_int), compar_proto(compar))
```

ptr is the returned void pointer. If key wasn't found in arr, the value would be `None`, but in this case we got a valid value.

Now we can convert it and access the value:

```
>>> cast(ptr, POINTER(c_int)).contents
c_long(12)
```

Also, we can see that ptr points to the correct value inside arr:

```
>>> addressof(arr) + 12 * sizeof(c_int) == ptr
True
```

# 视频：Python数据科学与机器学习训练营

学习如何使用NumPy、Pandas、Seaborn、Matplotlib、Plotly、Scikit-Learn、机器学习、Tensorflow等！



- ✓ 使用Python进行数据科学和机器学习
- ✓ 使用Spark进行大数据分析
- ✓ 实现机器学习算法
- ✓ 学习使用NumPy进行数值数据处理
- ✓ 学习使用Pandas进行数据分析
- ✓ 学习使用Matplotlib进行Python绘图
- ✓ 学习使用Seaborn进行统计图表绘制
- ✓ 使用Plotly进行交互式动态可视化
- ✓ 使用SciKit-Learn完成机器学习任务
- ✓ K均值聚类
- ✓ 逻辑回归
- ✓ 线性回归
- ✓ 随机森林和决策树
- ✓ 神经网络
- ✓ 支持向量机今天观看→

# VIDEO: Python for Data Science and Machine Learning Bootcamp

Learn how to use NumPy, Pandas, Seaborn, Matplotlib , Plotly, Scikit-Learn , Machine Learning, Tensorflow, and more!



- ✓ Use Python for Data Science and Machine Learning
- ✓ Use Spark for Big Data Analysis
- ✓ Implement Machine Learning Algorithms
- ✓ Learn to use NumPy for Numerical Data
- ✓ Learn to use Pandas for Data Analysis
- ✓ Learn to use Matplotlib for Python Plotting
- ✓ Learn to use Seaborn for statistical plots
- ✓ Use Plotly for interactive dynamic visualizations
- ✓ Use SciKit-Learn for Machine Learning Tasks
- ✓ K-Means Clustering
- ✓ Logistic Regression
- ✓ Linear Regression
- ✓ Random Forest and Decision Trees
- ✓ Neural Networks
- ✓ Support Vector Machines

**Watch Today →**

# 第190章：编写扩展

## 第190.1节：使用C扩展的Hello World

下面的C源文件（为了演示我们称之为hello.c）生成一个名为hello的扩展模块，包含一个函数greet()：

```
#include <Python.h>
#include <stdio.h>

#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif

static PyObject *hello_greet(PyObject *self, PyObject *args)
{
    const char *input;
    if (!PyArg_ParseTuple(args, "s", &input)) {
        return NULL;
    }
    printf("%s", input);
    Py_RETURN_NONE;
}

static PyMethodDef HelloMethods[] = {
    {"greet", hello_greet, METH_VARARGS, "向用户问好" },
    { NULL, NULL, 0, NULL }
};

#endif IS_PY3K
static struct PyModuleDef hellomodule = {
    PyModuleDef_HEAD_INIT, "hello", NULL, -1, HelloMethods
};

PyMODINIT_FUNC PyInit_hello(void)
{
    return PyModule_Create(&hellomodule);
}
#else
PyMODINIT_FUNC inithello(void)
{
    (void) Py_InitModule("hello", HelloMethods);
}
#endif
```

要使用gcc编译器编译该文件，请在您喜欢的终端中运行以下命令：

```
gcc /path/to/your/file/hello.c -o /path/to/your/file/hello
```

要执行我们之前编写的greet()函数，在同一目录下创建一个文件，命名为hello.py

```
import hello      # 导入已编译的库
hello.greet("Hello!") # 使用"Hello!"作为参数运行greet()函数
```

## 第190.2节：使用C++和Boost的C扩展

这是一个使用C++和Boost的基本C扩展示例。

# Chapter 190: Writing extensions

## Section 190.1: Hello World with C Extension

The following C source file (which we will call hello.c for demonstration purposes) produces an extension module named hello that contains a single function greet():

```
#include <Python.h>
#include <stdio.h>

#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif

static PyObject *hello_greet(PyObject *self, PyObject *args)
{
    const char *input;
    if (!PyArg_ParseTuple(args, "s", &input)) {
        return NULL;
    }
    printf("%s", input);
    Py_RETURN_NONE;
}

static PyMethodDef HelloMethods[] = {
    {"greet", hello_greet, METH_VARARGS, "Greet the user" },
    { NULL, NULL, 0, NULL }
};

#endif IS_PY3K
static struct PyModuleDef hellomodule = {
    PyModuleDef_HEAD_INIT, "hello", NULL, -1, HelloMethods
};

PyMODINIT_FUNC PyInit_hello(void)
{
    return PyModule_Create(&hellomodule);
}
#else
PyMODINIT_FUNC inithello(void)
{
    (void) Py_InitModule("hello", HelloMethods);
}
#endif
```

To compile the file with the gcc compiler, run the following command in your favourite terminal:

```
gcc /path/to/your/file/hello.c -o /path/to/your/file/hello
```

To execute the greet() function that we wrote earlier, create a file in the same directory, and call it hello.py

```
import hello      # imports the compiled library
hello.greet("Hello!") # runs the greet() function with "Hello!" as an argument
```

## Section 190.2: C Extension Using C++ and Boost

This is a basic example of a C Extension using C++ and [Boost](#).

## C++代码

放在hello.cpp中的C++代码：

```
#include <boost/python/module.hpp>
#include <boost/python/list.hpp>
#include <boost/python/class.hpp>
#include <boost/python/def.hpp>

// 返回一个hello world字符串。
std::string get_hello_function()
{
    return "Hello world!";
}

// 可以返回包含指定数量“hello world”字符串的列表的类。
class hello_class
{
public:
    // 在构造函数中接收问候消息。
    hello_class(std::string message) : _message(message) {}

    // 返回包含消息的列表，列表长度为count。
    boost::python::list as_list(int count)
    {
        boost::python::list res;
        for (int i = 0; i < count; ++i) {
            res.append(_message);
        }
        return res;
    }

private:
    std::string _message;
};

// 定义一个名为“hello”的Python模块。
BOOST_PYTHON_MODULE(hello)
{
    // 在这里声明应该在模块中暴露的函数和类。

    // get_hello_function作为函数暴露给Python。
    boost::python::def("get_hello", get_hello_function);

    // hello_class作为类暴露给Python。
    boost::python::class_<hello_class>("Hello", boost::python::init<std::string>())
        .def("as_list", &hello_class::as_list)
    ;
}
}
```

要将此编译为Python模块，您需要Python头文件和Boost库。此示例是在Ubuntu 12.04上使用Python 3.4和gcc制作的。Boost支持多种平台。在Ubuntu的情况下，所需的软件包是通过以下命令安装的：

```
sudo apt-get install gcc libboost-dev libpython3.4-dev
```

将源文件编译成一个.so文件，之后只要该文件在python路径中即可作为模块导入：

## C++ Code

C++ code put in hello.cpp:

```
#include <boost/python/module.hpp>
#include <boost/python/list.hpp>
#include <boost/python/class.hpp>
#include <boost/python/def.hpp>

// Return a hello world string.
std::string get_hello_function()
{
    return "Hello world!";
}

// hello class that can return a list of count hello world strings.
class hello_class
{
public:
    // Taking the greeting message in the constructor.
    hello_class(std::string message) : _message(message) {}

    // Returns the message count times in a python list.
    boost::python::list as_list(int count)
    {
        boost::python::list res;
        for (int i = 0; i < count; ++i) {
            res.append(_message);
        }
        return res;
    }

private:
    std::string _message;
};

// Defining a python module naming it to "hello".
BOOST_PYTHON_MODULE(hello)
{
    // Here you declare what functions and classes that should be exposed on the module.

    // The get_hello_function exposed to python as a function.
    boost::python::def("get_hello", get_hello_function);

    // The hello_class exposed to python as a class.
    boost::python::class_<hello_class>("Hello", boost::python::init<std::string>())
        .def("as_list", &hello_class::as_list)
    ;
}
}
```

To compile this into a python module you will need the python headers and the boost libraries. This example was made on Ubuntu 12.04 using python 3.4 and gcc. Boost is supported on many platforms. In case of Ubuntu the needed packages were installed using:

```
sudo apt-get install gcc libboost-dev libpython3.4-dev
```

Compiling the source file into a .so-file that can later be imported as a module provided it is on the python path:

```
gcc -shared -o hello.so -fPIC -I/usr/include/python3.4 hello.cpp -lboost_python-py34 -lboost_system  
-l:libpython3.4m.so
```

example.py文件中的python代码：

```
import hello  
  
print(hello.get_hello())  
  
h = hello.Hello("World hello!")  
print(h.as_list(3))
```

然后运行python3 example.py将输出如下内容：

```
你好，世界！  
['World hello!', 'World hello!', 'World hello!']
```

## 第190.3节：向C扩展传递打开的文件

将Python中的打开文件对象传递给C扩展代码。

您可以使用PyObject\_AsFileDescriptor函数将文件转换为整数文件描述符：

```
PyObject *fobj;  
int fd = PyObject_AsFileDescriptor(fobj);  
if (fd < 0){  
    return NULL;  
}
```

要将整数文件描述符转换回Python对象，请使用PyFile\_FromFd。

```
int fd; /* 现有的文件描述符 */  
PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NULL,1);
```

```
gcc -shared -o hello.so -fPIC -I/usr/include/python3.4 hello.cpp -lboost_python-py34 -lboost_system  
-l:libpython3.4m.so
```

The python code in the file example.py:

```
import hello  
  
print(hello.get_hello())  
  
h = hello.Hello("World hello!")  
print(h.as_list(3))
```

Then python3 example.py will give the following output:

```
Hello world!  
['World hello!', 'World hello!', 'World hello!']
```

## Section 190.3: Passing an open file to C Extensions

Pass an open file object from Python to C extension code.

You can convert the file to an integer file descriptor using PyObject\_AsFileDescriptor function:

```
PyObject *fobj;  
int fd = PyObject_AsFileDescriptor(fobj);  
if (fd < 0){  
    return NULL;  
}
```

To convert an integer file descriptor back into a python object, use PyFile\_FromFd.

```
int fd; /* Existing file descriptor */  
PyObject *fobj = PyFile_FromFd(fd, "filename", "r", -1, NULL, NULL, NULL, 1);
```

# 第191章：Python Lex-Yacc

PLY是流行的编译器构造工具lex和yacc的纯Python实现。

## 第191.1节：PLY入门

要在您的计算机上为python2/3安装PLY，请按照以下步骤操作：

1. 从 [here](#) 下载源代码。
2. 解压下载的压缩文件
3. 进入解压后的 `ply-3.10` 文件夹
4. 在终端运行以下命令：`python setup.py install`

如果完成了以上所有步骤，你现在应该可以使用 PLY 模块了。你可以通过打开一个 python 解释器并输入 `import ply.lex` 来测试它。

注意：不要使用 `pip` 来安装 PLY，它会在你的机器上安装一个损坏的版本。

## 第191.2节：PLY 的“Hello, World！”——一个简单的计算器

让我们用一个简单的例子来展示 PLY 的强大功能：这个程序将接受一个算术表达式作为字符串输入，并尝试计算它。

打开你喜欢的编辑器，复制以下代码：

```
from ply import lex
import ply.yacc as yacc

tokens = (
    'PLUS',
    'MINUS',
    'TIMES',
    'DIV',
    'LPAREN',
    'RPAREN',
    'NUMBER',
)
t_ignore = ' '

t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIV     = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

def t_NUMBER( t ):
    r'[0-9]+'
    t.value = int( t.value )
    return t

def t_newline( t ):
    r'\n'
    t.lineno += len( t.value )

def t_error( t ):
```

# Chapter 191: Python Lex-Yacc

PLY is a pure-Python implementation of the popular compiler construction tools lex and yacc.

## Section 191.1: Getting Started with PLY

To install PLY on your machine for python2/3, follow the steps outlined below:

1. Download the source code from [here](#).
2. Unzip the downloaded zip file
3. Navigate into the unzipped `ply-3.10` folder
4. Run the following command in your terminal: `python setup.py install`

If you completed all the above, you should now be able to use the PLY module. You can test it out by opening a python interpreter and typing `import ply.lex`.

Note: Do *not* use pip to install PLY, it will install a broken distribution on your machine.

## Section 191.2: The "Hello, World!" of PLY - A Simple Calculator

Let's demonstrate the power of PLY with a simple example: this program will take an arithmetic expression as a string input, and attempt to solve it.

Open up your favourite editor and copy the following code:

```
from ply import lex
import ply.yacc as yacc

tokens = (
    'PLUS',
    'MINUS',
    'TIMES',
    'DIV',
    'LPAREN',
    'RPAREN',
    'NUMBER',
)
t_ignore = ' '

t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIV     = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

def t_NUMBER( t ):
    r'[0-9]+'
    t.value = int( t.value )
    return t

def t_newline( t ):
    r'\n'
    t.lineno += len( t.value )

def t_error( t ):
```

```

print("无效的标记:", t.value[0])
t.lexer.skip(1)

lexer = lex.lex()

precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIV'),
    ('nonassoc', 'UMINUS')
)

def p_add( p ):
    'expr : expr PLUS expr'
    p[0] = p[1] + p[3]

def p_sub( p ):
    'expr : expr MINUS expr'
    p[0] = p[1] - p[3]

def p_expr2uminus( p ):
    'expr : MINUS expr %prec UMINUS'
    p[0] = - p[2]

def p_mult_div( p ):
    '''expr : expr TIMES expr
           | expr DIV expr'''

    if p[2] == '*':
        p[0] = p[1] * p[3]
    else:
        if p[3] == 0:
            print("不能除以0")
            raise ZeroDivisionError('整数除以0')
        p[0] = p[1] / p[3]

def p_expr2NUM( p ):
    'expr : NUMBER'
    p[0] = p[1]

def p_parens( p ):
    'expr : LPAREN expr RPAREN'
    p[0] = p[2]

def p_error( p ):
    print("输入中存在语法错误！")

parser = yacc.yacc()

res = parser.parse("-4*(3-5)") # 输入
print(res)

```

将此文件保存为calc.py并运行。

输出：

-8

以下表达式的正确答案是-4 \* - (3 - 5)。

```

print("Invalid Token:", t.value[0])
t.lexer.skip(1)

lexer = lex.lex()

precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIV'),
    ('nonassoc', 'UMINUS')
)

def p_add( p ):
    'expr : expr PLUS expr'
    p[0] = p[1] + p[3]

def p_sub( p ):
    'expr : expr MINUS expr'
    p[0] = p[1] - p[3]

def p_expr2uminus( p ):
    'expr : MINUS expr %prec UMINUS'
    p[0] = - p[2]

def p_mult_div( p ):
    '''expr : expr TIMES expr
           | expr DIV expr'''

    if p[2] == '*':
        p[0] = p[1] * p[3]
    else:
        if p[3] == 0:
            print("Can't divide by 0")
            raise ZeroDivisionError('integer division by 0')
        p[0] = p[1] / p[3]

def p_expr2NUM( p ):
    'expr : NUMBER'
    p[0] = p[1]

def p_parens( p ):
    'expr : LPAREN expr RPAREN'
    p[0] = p[2]

def p_error( p ):
    print("Syntax error in input!")

parser = yacc.yacc()

res = parser.parse("-4*(-3-5)") # the input
print(res)

```

Save this file as calc.py and run it.

Output:

-8

Which is the right answer for -4 \* - (3 - 5).

## 第191.3节：第1部分：使用Lex进行输入的词法分析

示例1中的代码执行了两个步骤：第一个是词法分析输入，即查找构成算术表达式的符号，第二个步骤是语法分析，涉及分析提取的词法单元并计算结果。

本节提供了一个如何词法分析用户输入的简单示例，并逐行进行解析。

```
import ply.lex as lex

# 令牌名称列表。这是必需的
tokens = [
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
]

# 简单标记的正则表达式规则
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# 一个带有部分动作代码的正则表达式规则
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# 定义一个规则以便我们跟踪行号
def t_newline(t):
    r'+'
    t.lineno += len(t.value)

# 一个包含被忽略字符（空格和制表符）的字符串
t_ignore = ' '

# 错误处理规则
def t_error(t):
    print("非法字符 '%s'" % t.value[0])
    t.skip(1)

# 构建词法分析器
lexer = lex.lex()

# 给词法分析器输入一些数据
lexer.input(data)

# 词法分析
while True:
    tok = lexer.token()
    if not tok:
        break      # 没有更多输入
    print(tok)
```

## Section 191.3: Part 1: Tokenizing Input with Lex

There are two steps that the code from example 1 carried out: one was *tokenizing* the input, which means it looked for symbols that constitute the arithmetic expression, and the second step was *parsing*, which involves analysing the extracted tokens and evaluating the result.

This section provides a simple example of how to *tokenize* user input, and then breaks it down line by line.

```
import ply.lex as lex

# List of token names. This is always required
tokens = [
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
]

# Regular expression rules for simple tokens
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.skip(1)

# Build the lexer
lexer = lex.lex()

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    print(tok)
```

将此文件保存为calclex.py。我们在构建Yacc解析器时会用到它。

## 拆解

1. 使用import ply.lex导入模块
2. 所有词法分析器必须提供一个名为tokens的列表，定义词法分析器可能产生的所有标记名称由词法分析器生成。此列表始终是必需的。

```
tokens = [
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
]
```

tokens也可以是字符串元组（而非字符串），其中每个字符串表示一个标记，如前所述。

3. 每个字符串的正则表达式规则可以定义为字符串或函数。无论哪种情况，变量名应以 t\_ 为前缀，以表示它是用于匹配标记的规则。

- 对于简单的标记，正则表达式可以指定为字符串：t\_PLUS = r'\+'
- 如果需要执行某种操作，则可以将标记规则指定为函数。

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

注意，规则作为函数内的文档字符串指定。该函数接受一个参数，该参数是一个 LexToken 实例，执行某些操作后返回该参数。

如果你想使用外部字符串作为函数的正则表达式规则，而不是指定文档字符串，请参考以下示例：

```
@TOKEN(identifier)      # identifier 是一个保存正则表达式的字符串
def t_ID(t):
...      # 操作
```

- 一个 LexToken 对象的实例（我们称该对象为 t）具有以下属性：

1. t.type 是标记类型（字符串）（例如：'NUMBER', 'PLUS' 等）。默认情况下，t.type 被设置为以 t\_ 前缀开头的名称。
2. t.value 是词素（实际匹配的文本）
3. t.lineno 是当前行号（这不会自动更新，因为词法分析器不知道行号）。使用名为 t\_newline 的函数来更新 lineno。

```
def t_newline(t):
    r'+'
    t.lexer.lineno += len(t.value)
```

Save this file as calclex.py. We'll be using this when building our Yacc parser.

## Breakdown

1. Import the module using `import ply.lex`
2. All lexers must provide a list called tokens that defines all of the possible token names that can be produced by the lexer. This list is always required.

```
tokens = [
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
]
```

tokens could also be a tuple of strings (rather than a string), where each string denotes a token as before.

3. The regex rule for each string may be defined either as a string or as a function. In either case, the variable name should be prefixed by t\_ to denote it is a rule for matching tokens.

- For simple tokens, the regular expression can be specified as strings: t\_PLUS = r'\+'
- If some kind of action needs to be performed, a token rule can be specified as a function.

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Note, the rule is specified as a doc string within the function. The function accepts one argument which is an instance of LexToken, performs some action and then returns back the argument.

If you want to use an external string as the regex rule for the function instead of specifying a doc string, consider the following example:

```
@TOKEN(identifier)      # identifier is a string holding the regex
def t_ID(t):
...      # actions
```

- An instance of LexToken object (let's call this object t) has the following attributes:
  1. t.type which is the token type (as a string) (eg: 'NUMBER', 'PLUS', etc). By default, t.type is set to the name following the t\_ prefix.
  2. t.value which is the lexeme (the actual text matched)
  3. t.lineno which is the current line number (this is not automatically updated, as the lexer knows nothing of line numbers). Update lineno using a function called t\_newline.

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

#### 4. t.lexpos 是相对于输入文本开头的标记位置。

- 如果正则表达式规则函数没有返回任何内容，则该标记会被丢弃。如果你想丢弃一个标记，你也可以通过给正则表达式规则变量添加 `t_ignore_` 前缀来实现，而不是为同一规则定义函数。

```
def t_COMMENT(t):
r'#.+'  
    通过  
    # 无返回值。令牌被丢弃
```

...等同于：

```
t_ignore_COMMENT = r'#.+'  
  
如果你在遇到注释时需要执行某些操作，这当然是无效的。在这种情况下，使用函数来定义正则表达式规则。  
。
```

如果你没有为某些字符定义令牌但仍想忽略它，使用 `t_ignore = "<要忽略的字符>"`（这些前缀是必须的）：

```
t_ignore_COMMENT = r'#.+'  
t_ignore = ' '      # 忽略空格和制表符
```

- 在构建主正则表达式时，lex 会按如下方式添加文件中指定的正则表达式：

- 由函数定义的令牌按它们在文件中出现的顺序添加。
- 由字符串定义的令牌按定义该令牌正则表达式的字符串长度递减顺序添加。

如果你在同一文件中匹配 `==` 和 `=`，请利用这些规则。

- 字面量是按原样返回的标记。类型 (`t.type`) 和值 (`t.value`) 都将被设置为字符本身。定义一个字面量列表如下：

```
字面量 = [ '+', '-', '*', '/' ]
```

或者，

```
字面量 = "+-*/"
```

可以编写在匹配字面量时执行额外操作的标记函数。

但是，你需要适当设置标记类型。例如：

```
字面量 = [ '{', '}' ]  
  
def t_lbrace(t):
r'{'
t.type = '{' # 将标记类型设置为预期的字面量（如果这是一个字面量，则绝对必须）
    返回 t
```

#### 4. `t.lexpos` which is the position of the token relative to the beginning of the input text.

- If nothing is returned from a regex rule function, the token is discarded. If you want to discard a token, you can alternatively add `t_ignore_` prefix to a regex rule variable instead of defining a function for the same rule.

```
def t_COMMENT(t):
r'#.+'  
pass  
# No return value. Token discarded
```

...Is the same as:

```
t_ignore_COMMENT = r'#.+'  
  
This is of course invalid if you're carrying out some action when you see a comment. In which case, use a function to define the regex rule.
```

If you haven't defined a token for some characters but still want to ignore it, use `t_ignore = "<characters to ignore>"` (these prefixes are necessary):

```
t_ignore_COMMENT = r'#.+'  
t_ignore = ' \t'      # ignores spaces and tabs
```

- When building the master regex, lex will add the regexes specified in the file as follows:

- Tokens defined by functions are added in the same order as they appear in the file.
- Tokens defined by strings are added in decreasing order of the string length of the string defining the regex for that token.

If you are matching `==` and `=` in the same file, take advantage of these rules.

- Literals are tokens that are returned as they are. Both `t.type` and `t.value` will be set to the character itself. Define a list of literals as such:

```
literals = [ '+', '-', '*', '/' ]
```

or,

```
literals = "+-*/"
```

It is possible to write token functions that perform additional actions when literals are matched. However, you'll need to set the token type appropriately. For example:

```
literals = [ '{', '}' ]  
  
def t_lbrace(t):
r'{'
t.type = '{' # Set token type to the expected literal (ABSOLUTE MUST if this is a literal)
    return t
```

- 使用 `t_error` 函数处理错误。

```
# 错误处理规则
def t_error(t):
    print("非法字符 '%s'" % t.value[0])
    t.lexer.skip(1) # 跳过非法标记 (不处理它)
```

通常, `t.lexer.skip(n)` 会跳过输入字符串中的 `n` 个字符。

#### 4. 最终准备工作：

使用 `lexer = lex.lex()` 构建词法分析器。

你也可以将所有内容放入一个类中, 并通过该类的实例来定义词法分析器。例如:

```
import ply.lex as lex
class MyLexer(object):
    ... # 与标记规则和错误处理相关的所有内容通常都写在这里

    # 构建词法分析器
    def build(self, **kwargs):
        self.lexer = lex.lex(module=self, **kwargs)

    def test(self, data):
        self.lexer.input(data)
        for token in self.lexer.token():
            print(token)

    # 构建词法分析器并进行测试

m = MyLexer()
m.build()      # 构建词法分析器
m.test("3 + 4") #
```

使用 `lexer.input(data)` 提供输入, 其中 `data` 是字符串要获取

词法单元, 使用 `lexer.token()`, 它返回匹配的词法单元。你可以像下面这样在循环中迭代 `lexer`:

```
for i in lexer:
    print(i)
```

## 第191.4节：第2部分：使用 Yacc 解析词法单元化的输入

本节解释了如何处理第1部分的分词输入——这是通过上下文无关文法 (CFGs) 完成的。必须指定文法, 并根据文法处理这些标记。在底层, 解析器使用的是LALR解析器。

```
# Yacc 示例

import ply.yacc as yacc

# 从词法分析器获取标记映射。这是必需的。
from calclex import tokens
```

- Handle errors with `t_error` function.

```
# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1) # skip the illegal token (don't process it)
```

In general, `t.lexer.skip(n)` skips `n` characters in the input string.

#### 4. Final preparations:

Build the lexer using `lexer = lex.lex()`.

You can also put everything inside a class and call use instance of the class to define the lexer. Eg:

```
import ply.lex as lex
class MyLexer(object):
    ... # everything relating to token rules and error handling comes here as usual

    # Build the lexer
    def build(self, **kwargs):
        self.lexer = lex.lex(module=self, **kwargs)

    def test(self, data):
        self.lexer.input(data)
        for token in self.lexer.token():
            print(token)

    # Build the lexer and try it out

m = MyLexer()
m.build()      # Build the lexer
m.test("3 + 4") #
```

Provide input using `lexer.input(data)` where `data` is a string

To get the tokens, use `lexer.token()` which returns tokens matched. You can iterate over `lexer` in a loop as in:

```
for i in lexer:
    print(i)
```

## Section 191.4: Part 2: Parsing Tokenized Input with Yacc

This section explains how the tokenized input from Part 1 is processed - it is done using Context Free Grammars (CFGs). The grammar must be specified, and the tokens are processed according to the grammar. Under the hood, the parser uses an LALR parser.

```
# Yacc example

import ply.yacc as yacc

# Get the token map from the lexer. This is required.
from calclex import tokens
```

```

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    '表达式 : 项'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term 除以 factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    '项 : 因子'
    p[0] = p[1]

def p_factor_num(p):
    '因子 : 数字'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# 语法错误的错误规则
def p_error(p):
    print("输入中存在语法错误！")

# 构建解析器
parser = yacc.yacc()

while True:
    尝试:
        s = raw_input('calc > ')
        except EOFError:
            break
        if not s: continue
        result = parser.parse(s)
        print(result)

```

## 拆解

- 每个语法规则由一个函数定义，该函数的文档字符串包含相应的上下文无关文法规范。构成函数体的语句实现该规则的语义动作。每个函数接受一个参数 p，该参数是一个序列，包含对应规则中每个语法符号的值。p[i] 的值映射到语法符号，如下所示：

```

def p_expression_plus(p):
    'expression : expression PLUS term'
    # ^          ^          ^          ^
    # p[0]      p[1]      p[2] p[3]

```

```

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")

# Build the parser
parser = yacc.yacc()

while True:
    try:
        s = raw_input('calc > ')
    except EOFError:
        break
    if not s: continue
    result = parser.parse(s)
    print(result)

```

## Breakdown

- Each grammar rule is defined by a function where the docstring to that function contains the appropriate context-free grammar specification. The statements that make up the function body implement the semantic actions of the rule. Each function accepts a single argument p that is a sequence containing the values of each grammar symbol in the corresponding rule. The values of p[i] are mapped to grammar symbols as shown here:

```

def p_expression_plus(p):
    'expression : expression PLUS term'
    # ^          ^          ^          ^
    # p[0]      p[1]      p[2] p[3]

```

```
p[0] = p[1] + p[3]
```

- 对于终结符，相应的 `p[i]` 的“值”与词法分析器模块中分配的 `p.value` 属性相同。因此，PLUS 的值将是 `+`。
- 对于非终结符，值由放置在 `p[0]` 中的内容决定。如果没有放置，值为 `None`。此外，`p[-1]` 不同于 `p[3]`，因为 `p` 不是简单的列表 (`p[-1]` 可以指定嵌入动作 (此处不讨论))。

注意，函数名可以是任意名称，只要前面加上 `p_` 即可。

- `p_error(p)` 规则用于捕获语法错误 (与 `yacc/bison` 中的 `yyerror` 相同)。
- 多个语法规则可以合并到一个函数中，如果产生式结构相似，这是一个好主意。

```
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
        term      : term TIMES factor
                  | term DIVIDE factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

- 字符串常量可以代替标记 (tokens)。

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
        term      : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

当然，常量必须在词法分析模块中指定。

- 空产生式的形式为 `'''symbol : '''`
- 要显式设置起始符号，使用 `start = 'foo'`，其中 `foo` 是某个非终结符。
- 可以使用优先级变量来设置优先级和结合性。

```
优先级 = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'), # 非结合运算符
```

```
p[0] = p[1] + p[3]
```

- For tokens, the "value" of the corresponding `p[i]` is the same as the `p.value` attribute assigned in the lexer module. So, PLUS will have the value `+`.
- For non-terminals, the value is determined by whatever is placed in `p[0]`. If nothing is placed, the value is `None`. Also, `p[-1]` is not the same as `p[3]`, since `p` is not a simple list (`p[-1]` can specify embedded actions (not discussed here)).

Note that the function can have any name, as long as it is preceded by `p_`.

- The `p_error(p)` rule is defined to catch syntax errors (same as `yyerror` in `yacc/bison`).
- Multiple grammar rules can be combined into a single function, which is a good idea if productions have a similar structure.

```
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
        term      : term TIMES factor
                  | term DIVIDE factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

- Character literals can be used instead of tokens.

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
        term      : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

Of course, the literals must be specified in the lexer module.

- Empty productions have the form `'''symbol : '''`
- To explicitly set the start symbol, use `start = 'foo'`, where `foo` is some non-terminal.
- Setting precedence and associativity can be done using the precedence variable.

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'), # Nonassociative operators
```

```
('left', 'PLUS', 'MINUS'),
('left', 'TIMES', 'DIVIDE'),
('right', 'UMINUS'),      # 一元负号运算符
)
```

标记按优先级从低到高排序。nonassoc 表示这些标记不结合。

这意味着像  $a < b < c$  这样的表达式是非法的，而  $a < b$  仍然是合法的。

- parser.out 是一个调试文件，当 yacc 程序首次执行时创建。每当发生移入/规约冲突时，解析器总是选择移入。

```
('left', 'PLUS', 'MINUS'),
('left', 'TIMES', 'DIVIDE'),
('right', 'UMINUS'),      # Unary minus operator
)
```

Tokens are ordered from lowest to highest precedence. nonassoc means that those tokens do not associate.

This means that something like  $a < b < c$  is illegal whereas  $a < b$  is still legal.

- parser.out is a debugging file that is created when the yacc program is executed for the first time. Whenever a shift/reduce conflict occurs, the parser always shifts.

# 第192章：单元测试

## 第192.1节：在unittest.TestCase中进行测试设置和拆卸

有时我们希望为每个测试准备一个上下文。setUp方法会在类中的每个测试之前运行。tearDown会在每个测试结束时运行。这些方法是可选的。请记住，TestCase通常用于协作式多重继承，因此在这些方法中应始终小心调用super，以确保基类的setUp和tearDown方法也被调用。TestCase的基类实现提供了空的setUp和tearDown方法，因此调用它们不会引发异常：

```
import unittest

class SomeTest(unittest.TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.mock_data = [1, 2, 3, 4, 5]

    def test(self):
        self.assertEqual(len(self.mock_data), 5)

    def tearDown(self):
        super(SomeTest, self).tearDown()
        self.mock_data = []

if __name__ == '__main__':
    unittest.main()
```

请注意，在 Python 2.7 及以上版本中，还有一个 addCleanup 方法，用于注册在测试运行后调用的函数。与仅在 setUp 成功时才调用的 tearDown 不同，通过 addCleanup 注册的函数即使在 setUp 中发生未处理异常时也会被调用。举一个具体的例子，这个方法经常用于移除测试运行期间注册的各种模拟对象：

```
import unittest
import some_module

class SomeOtherTest(unittest.TestCase):
    def setUp(self):
        super(SomeOtherTest, self).setUp()

        # 用 mock.Mock 替换 `some_module.method`
        my_patch = mock.patch.object(some_module, 'method')
        my_patch.start()

        # 测试运行结束后，恢复原始方法。
        self.addCleanup(my_patch.stop)
```

以这种方式注册清理函数的另一个好处是，它允许程序员将清理代码放在设置代码旁边，并且在子类忘记在 tearDown 中调用 super 时保护你。

## 第192.2节：断言异常

你可以通过内置的 unittest 用两种不同的方法测试函数是否抛出异常。

# Chapter 192: Unit Testing

## Section 192.1: Test Setup and Teardown within a unittest.TestCase

Sometimes we want to prepare a context for each test to be run under. The setUp method is run prior to each test in the class. tearDown is run at the end of every test. These methods are optional. Remember that TestCases are often used in cooperative multiple inheritance so you should be careful to always call `super` in these methods so that base class's setUp and tearDown methods also get called. The base implementation of TestCase provides empty setUp and tearDown methods so that they can be called without raising exceptions:

```
import unittest

class SomeTest(unittest.TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.mock_data = [1, 2, 3, 4, 5]

    def test(self):
        self.assertEqual(len(self.mock_data), 5)

    def tearDown(self):
        super(SomeTest, self).tearDown()
        self.mock_data = []

if __name__ == '__main__':
    unittest.main()
```

Note that in python2.7+, there is also the `addCleanup` method that registers functions to be called after the test is run. In contrast to tearDown which only gets called if setUp succeeds, functions registered via addCleanup will be called even in the event of an unhandled exception in setUp. As a concrete example, this method can frequently be seen removing various mocks that were registered while the test was running:

```
import unittest
import some_module

class SomeOtherTest(unittest.TestCase):
    def setUp(self):
        super(SomeOtherTest, self).setUp()

        # Replace `some_module.method` with a `mock.Mock`
        my_patch = mock.patch.object(some_module, 'method')
        my_patch.start()

        # When the test finishes running, put the original method back.
        self.addCleanup(my_patch.stop)
```

Another benefit of registering cleanups this way is that it allows the programmer to put the cleanup code next to the setup code and it protects you in the event that a subclasser forgets to call `super` in tearDown.

## Section 192.2: Asserting on Exceptions

You can test that a function throws an exception with the built-in unittest through two different methods.

## 使用上下文管理器

```
def division_function(dividend, divisor):
    return dividend / divisor

class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError):
            x = division_function(1, 0)
```

这将运行上下文管理器内的代码，如果代码成功执行，则测试失败，因为异常未被抛出。如果代码抛出了正确类型的异常，测试将继续进行。

如果需要，你也可以获取抛出异常的内容，以便对其进行额外的断言。

```
class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError) as ex:
            x = division_function(1, 0)

        self.assertEqual(ex.message, '整数除法或取模除以零')
```

## 通过提供一个可调用函数

```
def division_function(dividend, divisor):
    """
    除以两个数字。
    :type dividend: int
    :type divisor: int

    :raises: 如果除数为零(0)，则抛出ZeroDivisionError。
    :rtype: int
    """
    return dividend / divisor
```

```
class MyTestCase(unittest.TestCase):
    def test_passing_function(self):
        self.assertRaises(ZeroDivisionError, division_function, 1, 0)
```

要检查的异常必须是第一个参数，第二个参数必须传入一个可调用函数。任何其他指定的参数将直接传递给被调用的函数，允许你指定触发异常的参数。

## 第192.3节：测试异常

程序在输入错误时会抛出错误。因此，需要确保在实际输入错误时抛出错误。为此，我们需要检查特定的异常，以下示例中我们将使用以下异常：

```
class WrongInputException(Exception):
    通过
```

当输入错误时会抛出此异常，以下场景中我们总是期望输入的文本是数字。

## Using a context manager

```
def division_function(dividend, divisor):
    return dividend / divisor

class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError):
            x = division_function(1, 0)
```

This will run the code inside of the context manager and, if it succeeds, it will fail the test because the exception was not raised. If the code raises an exception of the correct type, the test will continue.

You can also get the content of the raised exception if you want to execute additional assertions against it.

```
class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError) as ex:
            x = division_function(1, 0)

        self.assertEqual(ex.message, 'integer division or modulo by zero')
```

## By providing a callable function

```
def division_function(dividend, divisor):
    """
    Dividing two numbers.

    :type dividend: int
    :type divisor: int

    :raises: ZeroDivisionError if divisor is zero (0).
    :rtype: int
    """
    return dividend / divisor
```

```
class MyTestCase(unittest.TestCase):
    def test_passing_function(self):
        self.assertRaises(ZeroDivisionError, division_function, 1, 0)
```

The exception to check for must be the first parameter, and a callable function must be passed as the second parameter. Any other parameters specified will be passed directly to the function that is being called, allowing you to specify the parameters that trigger the exception.

## Section 192.3: Testing Exceptions

Programs throw errors when for instance wrong input is given. Because of this, one needs to make sure that an error is thrown when actual wrong input is given. Because of that we need to check for an exact exception, for this example we will use the following exception:

```
class WrongInputException(Exception):
    pass
```

This exception is raised when wrong input is given, in the following context where we always expect a number as text input.

```

def convert2number(random_input):
    尝试:
        my_input = int(random_input)
    except ValueError:
        raise WrongInputException("Expected an integer!")
    return my_input

```

为了检查是否抛出了异常，我们使用assertRaises来检测该异常。assertRaises有两种用法：

1. 使用常规函数调用。第一个参数是异常类型，第二个参数是可调用对象（通常是函数），其余参数传递给该可调用对象。
2. 使用带有with子句，仅向函数提供异常类型。这的优点是可以编写更多代码应该执行，但应谨慎使用，因为多个函数可能使用相同的异常，这可能会导致问题。举个例子：使用 self.assertRaises(WrongInputException): convert2number("not a number")

这首先已在以下测试用例中实现：

```

import unittest

类 ExceptionTestCase(unittest.TestCase):

    定义 test_wrong_input_string(self):
        self.assertRaises(WrongInputException, convert2number, "not a number")

    定义 test_correct_input(self):
        尝试:
            结果 = convert2number("56")
            self.assertIsInstance(结果, int)
        except WrongInputException:
            self.fail()

```

也可能需要检查不应抛出的异常。然而，当异常被抛出时，测试会自动失败，因此这可能根本不必要。仅为展示选项，第二个测试方法展示了如何检查异常不被抛出的情况。基本上，这是捕获异常然后使用fail方法使测试失败。

```

def convert2number(random_input):
    try:
        my_input = int(random_input)
    except ValueError:
        raise WrongInputException("Expected an integer!")
    return my_input

```

To check whether an exception has been raised, we use assertRaises to check for that exception. assertRaises can be used in two ways:

1. Using the regular function call. The first argument takes the exception type, second a callable (usually a function) and the rest of arguments are passed to this callable.
2. Using a `with` clause, giving only the exception type to the function. This has as advantage that more code can be executed, but should be used with care since multiple functions can use the same exception which can be problematic. An example: with self.assertRaises(WrongInputException): convert2number("not a number")

This first has been implemented in the following test case:

```

import unittest

class ExceptionTestCase(unittest.TestCase):

    def test_wrong_input_string(self):
        self.assertRaises(WrongInputException, convert2number, "not a number")

    def test_correct_input(self):
        try:
            result = convert2number("56")
            self.assertIsInstance(result, int)
        except WrongInputException:
            self.fail()

```

There also may be a need to check for an exception which should not have been thrown. However, a test will automatically fail when an exception is thrown and thus may not be necessary at all. Just to show the options, the second test method shows a case on how one can check for an exception not to be thrown. Basically, this is catching the exception and then failing the test using the fail method.

## 第192.4节：在单元测试中选择断言

虽然 Python 有一个 `assert` 语句，但 Python 单元测试框架有更适合测试的断言：它们在失败时提供更多信息，并且不依赖于执行的调试模式。

也许最简单的断言是 `assertTrue`，可以这样使用：

```

import unittest

类 SimplisticTest(unittest.TestCase):
    定义 test_basic(self):
        self.assertTrue(1 + 1 == 2)

```

这将正常运行，但将上面那行替换为

```
self.assertTrue(1 + 1 == 3)
```

将会失败。

## Section 192.4: Choosing Assertions Within Unitests

While Python has an `assert statement`, the Python unit testing framework has better assertions specialized for tests: they are more informative on failures, and do not depend on the execution's debug mode.

Perhaps the simplest assertion is `assertTrue`, which can be used like this:

```

import unittest

class SimplisticTest(unittest.TestCase):
    定义 test_basic(self):
        self.assertTrue(1 + 1 == 2)

```

This will run fine, but replacing the line above with

```
self.assertTrue(1 + 1 == 3)
```

will fail.

`assertTrue` 断言很可能是最通用的断言，因为任何被测试的内容都可以转换为某种布尔条件，但通常有更好的替代方案。当测试相等性时，如上所示，最好写成

```
self.assertEqual(1 + 1, 3)
```

当前者失败时，消息是

```
=====
```

失败：测试 (`__main__.TruthTest`)

```
=====
```

回溯（最近一次调用最后）：

文件 "stuff.py"，第6行，在 test 中

```
self.assertTrue(1 + 1 == 3)
```

断言错误：False 不是 True

但当后者失败时，消息是

```
=====
```

失败：测试 (`__main__.TruthTest`)

```
=====
```

回溯（最近一次调用最后）：

文件 "stuff.py"，第6行，在 test 中

```
self.assertEqual(1 + 1, 3)
```

断言错误：2 != 3

这更具信息性（它实际上计算了左侧的结果）。

你可以在标准文档中找到断言列表。一般来说，[选择最具体符合条件的断言是个好主意](#)。因此，如上所示，对于断言`1 + 1 == 2`，最好使用`assertEqual`而不是`assertTrue`。同样，对于断言`a` 是`None`，最好使用`assertIsNone`而不是`assertEqual`。

请注意，断言也有否定形式。因此，`assertEqual` 有其否定对应 `assertNotEqual`，而 `assertIsNone` 有其否定对应 `assertIsNotNone`。再次强调，适当使用否定对应形式，将使错误信息更清晰。

## 第192.5节：使用pytest进行单元测试

安装pytest：

```
pip install pytest
```

准备测试：

The `assertTrue` assertion is quite likely the most general assertion, as anything tested can be cast as some boolean condition, but often there are better alternatives. When testing for equality, as above, it is better to write

```
self.assertEqual(1 + 1, 3)
```

When the former fails, the message is

```
=====
```

FAIL: test (`__main__.TruthTest`)

```
=====
```

Traceback (most recent call last):

File "stuff.py", line 6, in test

```
self.assertTrue(1 + 1 == 3)
```

AssertionError: False is not true

but when the latter fails, the message is

```
=====
```

FAIL: test (`__main__.TruthTest`)

```
=====
```

Traceback (most recent call last):

File "stuff.py", line 6, in test

```
self.assertEqual(1 + 1, 3)
```

AssertionError: 2 != 3

which is more informative (it actually evaluated the result of the left hand side).

You can find the list of assertions [in the standard documentation](#). In general, it is a good idea to choose the assertion that is the most specifically fitting the condition. Thus, as shown above, for asserting that `1 + 1 == 2` it is better to use `assertEqual` than `assertTrue`. Similarly, for asserting that `a` is `None`, it is better to use `assertIsNone` than `assertEqual`.

Note also that the assertions have negative forms. Thus `assertEqual` has its negative counterpart `assertNotEqual`, and `assertIsNone` has its negative counterpart `assertIsNotNone`. Once again, using the negative counterparts when appropriate, will lead to clearer error messages.

## Section 192.5: Unit tests with pytest

installing pytest:

```
pip install pytest
```

getting the tests ready:

```
mkdir tests  
touch tests/test_docker.py
```

需要测试的函数位于 docker\_something/helpers.py :

```
from subprocess import Popen, PIPE  
# 这个Popen被夹具'all_popens'进行了猴子补丁  
  
def copy_file_to_docker(src, dest):  
    尝试:  
        result = Popen(['docker', 'cp', src, 'something_cont:{}'.format(dest)], stdout=PIPE,  
                      stderr=PIPE)  
        err = result.stderr.read()  
        if err:  
            raise Exception(err)  
        except Exception as e:  
            print(e)  
    return result  
  
def docker_exec_something(something_file_string):  
    f1 = Popen(["docker", "exec", "-i", "something_cont", "something"], stdin=PIPE, stdout=PIPE,  
              stderr=PIPE)  
    f1.stdin.write(something_file_string)  
    f1.stdin.close()  
    err = f1.stderr.read()  
    f1.stderr.close()  
    if err:  
        print(err)  
        exit()  
    result = f1.stdout.read()  
    print(result)
```

测试导入 test\_docker.py :

```
import os  
from tempfile import NamedTemporaryFile  
import pytest  
from subprocess import Popen, PIPE  
  
from docker_something import helpers  
copy_file_to_docker = helpers.copy_file_to_docker  
docker_exec_something = helpers.docker_exec_something
```

在 test\_docker.py 中模拟类似文件的对象：

```
class MockBytes():  
    '''用于收集字节  
    ...  
    all_read = []  
    all_write = []  
    all_close = []  
  
    def read(self, *args, **kwargs):  
        # print('read', args, kwargs, dir(self))  
        self.all_read.append((self, args, kwargs))  
  
    def write(self, *args, **kwargs):  
        # print('wrote', args, kwargs)  
        self.all_write.append((self, args, kwargs))
```

```
mkdir tests  
touch tests/test_docker.py
```

Functions to test in docker\_something/helpers.py:

```
from subprocess import Popen, PIPE  
# this Popen is monkeypatched with the fixture 'all_popens'  
  
def copy_file_to_docker(src, dest):  
    尝试:  
        result = Popen(['docker', 'cp', src, 'something_cont:{}'.format(dest)], stdout=PIPE,  
                      stderr=PIPE)  
        err = result.stderr.read()  
        if err:  
            raise Exception(err)  
        except Exception as e:  
            print(e)  
    return result  
  
def docker_exec_something(something_file_string):  
    f1 = Popen(["docker", "exec", "-i", "something_cont", "something"], stdin=PIPE, stdout=PIPE,  
              stderr=PIPE)  
    f1.stdin.write(something_file_string)  
    f1.stdin.close()  
    err = f1.stderr.read()  
    f1.stderr.close()  
    if err:  
        print(err)  
        exit()  
    result = f1.stdout.read()  
    print(result)
```

The test imports test\_docker.py:

```
import os  
from tempfile import NamedTemporaryFile  
import pytest  
from subprocess import Popen, PIPE  
  
from docker_something import helpers  
copy_file_to_docker = helpers.copy_file_to_docker  
docker_exec_something = helpers.docker_exec_something
```

mocking a file like object in test\_docker.py:

```
class MockBytes():  
    '''Used to collect bytes  
    ...  
    all_read = []  
    all_write = []  
    all_close = []  
  
    def read(self, *args, **kwargs):  
        # print('read', args, kwargs, dir(self))  
        self.all_read.append((self, args, kwargs))  
  
    def write(self, *args, **kwargs):  
        # print('wrote', args, kwargs)  
        self.all_write.append((self, args, kwargs))
```

```

def close(self, *args, **kwargs):
    # print('closed', self, args, kwargs)
    self.all_close.append((self, args, kwargs))

def get_all_mock_bytes(self):
    return self.all_read, self.all_write, self.all_close

```

使用 pytest 在 test\_docker.py 中进行猴子补丁：

```

@pytest.fixture
def all_popens(monkeypatch):
    """此来真覆盖/模拟内置的 Popen
    并用 MockBytes 对象替换 stdin、stdout、stderr

    注意：monkeypatch 是自动导入的
    ...
    all_popens = []

    class MockPopen(object):
        def __init__(self, args, stdout=None, stdin=None, stderr=None):
            all_popens.append(self)
            self.args = args
            self.byte_collection = MockBytes()
            self.stdin = self.byte_collection
            self.stdout = self.byte_collection
            self.stderr = self.byte_collection
    通过
    monkeypatch.setattr(helpers, 'Popen', MockPopen)

    return all_popens

```

示例测试，必须以前缀 test\_ 开头，位于 test\_docker.py 文件中：

```

def test_docker_install():
    p = Popen(['which', 'docker'], stdout=PIPE, stderr=PIPE)
    result = p.stdout.read()
    assert 'bin/docker' in result

def test_copy_file_to_docker(all_popens):
    result = copy_file_to_docker('asdf', 'asdf')
    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert mock_read
    assert result.args == ['docker', 'cp', 'asdf', 'something_cont:asdf']

def test_docker_exec_something(all_popens):
    docker_exec_something(something_file_string)

    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert len(mock_read) == 3
    something_template_stdin = mock_write[0][1][0]
    these = [os.environ['USER'], os.environ['password_prod'], 'table_name_here', 'test_vdm',
    'col_a', 'col_b', '/tmp/test.tsv']
    assert all([x in something_template_stdin for x in these])

```

一次运行测试：

```

def close(self, *args, **kwargs):
    # print('closed', self, args, kwargs)
    self.all_close.append((self, args, kwargs))

def get_all_mock_bytes(self):
    return self.all_read, self.all_write, self.all_close

```

Monkey patching with pytest in test\_docker.py:

```

@pytest.fixture
def all_popens(monkeypatch):
    """This fixture overrides / mocks the builtin Popen
    and replaces stdin, stdout, stderr with a MockBytes object

    note: monkeypatch is magically imported
    ...
    all_popens = []

    class MockPopen(object):
        def __init__(self, args, stdout=None, stdin=None, stderr=None):
            all_popens.append(self)
            self.args = args
            self.byte_collection = MockBytes()
            self.stdin = self.byte_collection
            self.stdout = self.byte_collection
            self.stderr = self.byte_collection
    pass
    monkeypatch.setattr(helpers, 'Popen', MockPopen)

    return all_popens

```

Example tests, must start with the prefix test\_ in the test\_docker.py file:

```

def test_docker_install():
    p = Popen(['which', 'docker'], stdout=PIPE, stderr=PIPE)
    result = p.stdout.read()
    assert 'bin/docker' in result

def test_copy_file_to_docker(all_popens):
    result = copy_file_to_docker('asdf', 'asdf')
    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert mock_read
    assert result.args == ['docker', 'cp', 'asdf', 'something_cont:asdf']

def test_docker_exec_something(all_popens):
    docker_exec_something(something_file_string)

    collected_popen = all_popens.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert len(mock_read) == 3
    something_template_stdin = mock_write[0][1][0]
    these = [os.environ['USER'], os.environ['password_prod'], 'table_name_here', 'test_vdm',
    'col_a', 'col_b', '/tmp/test.tsv']
    assert all([x in something_template_stdin for x in these])

```

running the tests one at a time:

```
py.test -k test_docker_install tests  
py.test -k test_copy_file_to_docker tests  
py.test -k test_docker_exec_something tests
```

运行 tests 文件夹中的所有测试：

```
py.test -k test_ tests
```

## 第192.6节：使用 unittest.mock.create\_autospec 模拟函数

模拟函数的一种方法是使用create\_autospec函数，它会根据对象的规格进行模拟。对于函数，我们可以用它来确保函数被正确调用。

在custom\_math.py中有一个函数multiply：

```
def multiply(a, b):  
    return a * b
```

以及在process\_math.py中有一个函数multiples\_of：

```
from custom_math import multiply
```

```
def multiples_of(integer, *args, num_multiples=0, **kwargs):
```

```
    """  
    :rtype: list  
    """
```

```
multiples = []
```

```
    for x in range(1, num_multiples + 1):  
        """
```

传入 args 和 kwargs 只有在向 multiples\_of 函数传递了值时才会引发 TypeError，否则会被忽略。这样我们可以测试 multiples\_of 是否被正确使用。这里是为了演示 create\_autospec 的工作原理。不推荐用于生产代码。

```
    """
```

```
multiple = multiply(integer, x, *args, **kwargs)  
multiples.append(multiple)
```

```
return multiples
```

我们可以通过模拟 multiply 来单独测试 multiples\_of。下面的示例使用了 Python 标准库 unittest，但这也可以用于其他测试框架，如 pytest 或 nose：

```
from unittest.mock import create_autospec  
import unittest  
  
# 我们导入整个模块以便模拟 multiply  
import custom_math  
custom_math.multiply = create_autospec(custom_math.multiply)  
from process_math import multiples_of  
  
class TestCustomMath(unittest.TestCase):  
    def test_multiples_of(self):  
        multiples = multiples_of(3, num_multiples=1)  
        custom_math.multiply.assert_called_with(3, 1)
```

```
py.test -k test_docker_install tests  
py.test -k test_copy_file_to_docker tests  
py.test -k test_docker_exec_something tests
```

running all the tests in the tests folder:

```
py.test -k test_ tests
```

## Section 192.6: Mocking functions with unittest.mock.create\_autospec

One way to mock a function is to use the create\_autospec function, which will mock out an object according to its specs. With functions, we can use this to ensure that they are called appropriately.

With a function multiply in custom\_math.py:

```
def multiply(a, b):  
    return a * b
```

And a function multiples\_of in process\_math.py:

```
from custom_math import multiply
```

```
def multiples_of(integer, *args, num_multiples=0, **kwargs):
```

```
    """  
    :rtype: list  
    """
```

```
multiples = []
```

```
    for x in range(1, num_multiples + 1):  
        """
```

Passing in args and kwargs here will only raise TypeError if values were passed to multiples\_of function, otherwise they are ignored. This way we can test that multiples\_of is used correctly. This is here for an illustration of how create\_autospec works. Not recommended for production code.

```
    """  
    multiple = multiply(integer, x, *args, **kwargs)  
    multiples.append(multiple)
```

```
return multiples
```

We can test multiples\_of alone by mocking out multiply. The below example uses the Python standard library unittest，but this can be used with other testing frameworks as well, like pytest or nose:

```
from unittest.mock import create_autospec  
import unittest  
  
# we import the entire module so we can mock out multiply  
import custom_math  
custom_math.multiply = create_autospec(custom_math.multiply)  
from process_math import multiples_of
```

```
class TestCustomMath(unittest.TestCase):  
    def test_multiples_of(self):  
        multiples = multiples_of(3, num_multiples=1)  
        custom_math.multiply.assert_called_with(3, 1)
```

```
def test_multiples_of_with_bad_inputs(self):
    with self.assertRaises(TypeError) as e:
        multiples_of(1, "extra arg", num_multiples=1) # this should raise a TypeError
```

```
def test_multiples_of_with_bad_inputs(self):
    with self.assertRaises(TypeError) as e:
        multiples_of(1, "extra arg", num_multiples=1) # this should raise a TypeError
```

# 第193章：py.test

## 第193.1节：设置py.test

py.test 是Python可用的几个第三方测试库之一。它可以使用pip安装，命令为

```
pip install pytest
```

### 测试代码

假设我们正在测试位于projectroot/module/code.py中的加法函数：

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

### 测试代码

我们在projectroot/tests/test\_code.py中创建一个测试文件。该文件必须以test\_开头，才能被识别为测试文件。

```
# projectroot/tests/test_code.py
from module import code

def test_add():
    assert code.add(1, 2) == 3
```

### 运行测试

从projectroot我们只需运行py.test：

```
# 确保我们有这些模块
$ touch tests/__init__.py
$ touch module/__init__.py
$ py.test
=====
===== test 会话开始
=====
平台 darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
根目录: /projectroot, 配置文件:
收集了 1 个测试项

tests/test_code.py .

=====
===== 1 通过 用时 0.01 秒
=====
```

## 第193.2节：测试夹具简介

更复杂的测试有时需要在运行你想测试的代码之前进行一些设置。这可以在测试函数本身中完成，但这样会导致测试函数变得很大，做了太多事情，难以区分设置部分和测试部分的界限。你也可能在不同的测试函数之间出现大量重复的设置代码。

# Chapter 193: py.test

## Section 193.1: Setting up py.test

py.test is one of several [third party testing libraries](#) that are available for Python. It can be installed using pip with

```
pip install pytest
```

### The Code to Test

Say we are testing an addition function in projectroot/module/code.py:

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

### The Testing Code

We create a test file in projectroot/tests/test\_code.py. The file **must begin with test\_** to be recognized as a testing file.

```
# projectroot/tests/test_code.py
from module import code

def test_add():
    assert code.add(1, 2) == 3
```

### Running The Test

From projectroot we simply run py.test:

```
# ensure we have the modules
$ touch tests/__init__.py
$ touch module/__init__.py
$ py.test
=====
===== test session starts
=====
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py .

=====
===== 1 passed in 0.01 seconds
=====
```

## Section 193.2: Intro to Test Fixtures

More complicated tests sometimes need to have things set up before you run the code you want to test. It is possible to do this in the test function itself, but then you end up with large test functions doing so much that it is difficult to tell where the setup stops and the test begins. You can also get a lot of duplicate setup code between your various test functions.

我们的代码文件：

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
        self.bar = 2
```

我们的测试文件：

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff

def test_foo_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000

def test_bar_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

这些例子相当简单，但如果我们的Stuff对象需要更多的初始化，代码会变得难以管理。我们看到测试用例之间有一些重复代码，所以先将其重构成一个单独的函数。

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff

def get_prep_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff

def test_foo_updates():
    my_stuff = get_prep_stuff()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000

def test_bar_updates():
    my_stuff = get_prep_stuff()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

这样看起来更好，但我们仍然有`my_stuff = get_prep_stuff()`的调用，使测试函数显得杂乱。

**py.test的fixture来帮忙！**

Our code file:

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
        self.bar = 2
```

Our test file:

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff

def test_foo_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000

def test_bar_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

These are pretty simple examples, but if our Stuff object needed a lot more setup, it would get unwieldy. We see that there is some duplicated code between our test cases, so let's refactor that into a separate function first.

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff

def get_prep_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff

def test_foo_updates():
    my_stuff = get_prep_stuff()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000

def test_bar_updates():
    my_stuff = get_prep_stuff()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

This looks better but we still have the `my_stuff = get_prep_stuff()` call cluttering up our test functions.

**py.test fixtures to the rescue!**

Fixture是比测试初始化函数更强大、更灵活的版本。它们能做的远不止我们这里利用的这些，但我们会一步步来。

首先，我们将`get_preppeed_stuff`改成一个名为`prepped_stuff`的fixture。你应该用名词来命名fixture，而不是动词，因为fixture最终会在测试函数中以名词形式使用。`@pytest.fixture`表示这个特定函数应被视为fixture，而不是普通函数。

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

现在我们应该更新测试函数，使它们使用夹具。这是通过在定义中添加一个与夹具名称完全匹配的参数来完成的。当`py.test`执行时，它会在运行测试之前先运行夹具，然后通过该参数将夹具的返回值传递给测试函数。（注意，夹具不需要返回值；它们也可以执行其他设置操作，比如调用外部资源、安排文件系统上的内容、将值放入数据库，或者测试所需的任何设置）

```
def test_foo_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 1 == my_stuff.foo
    my_stuff.foo = 3000
    assert my_stuff.foo == 3000
```

```
def test_bar_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

现在你可以理解为什么我们用名词来命名它了。那行`my_stuff = prepped_stuff`基本上没什么用，所以我们直接使用`prepped_stuff`就好了。

```
def test_foo_updates(prepped_stuff):
    assert 1 == prepped_stuff.foo
    prepped_stuff.foo = 3000
    assert prepped_stuff.foo == 3000
```

```
def test_bar_updates(prepped_stuff):
    assert 2 == prepped_stuff.bar
    prepped_stuff.bar = 42
    assert 42 == prepped_stuff.bar
```

现在我们正在使用夹具！我们可以更进一步，通过更改夹具的作用域（使其每个测试模块或测试套件执行会话只运行一次，而不是每个测试函数运行一次）、构建使用其他夹具的夹具、参数化夹具（使夹具及所有使用该夹具的测试多次运行，每次使用传递给夹具的一个参数）、夹具从调用它们的模块读取值……如前所述，夹具比普通的设置函数拥有更多的功能和灵活性。

## 测试完成后的清理工作。

假设我们的代码已经增长，现在我们的`Stuff`对象需要特殊的清理。

```
# projectroot/module/stuff.py
```

Fixtures are much more powerful and flexible versions of test setup functions. They can do a lot more than we're leveraging here, but we'll take it one step at a time.

First we change `get_preppeed_stuff` to a fixture called `prepped_stuff`. You want to name your fixtures with nouns rather than verbs because of how the fixtures will end up being used in the test functions themselves later. The `@pytest.fixture` indicates that this specific function should be handled as a fixture rather than a regular function.

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

Now we should update the test functions so that they use the fixture. This is done by adding a parameter to their definition that exactly matches the fixture name. When `py.test` executes, it will run the fixture before running the test, then pass the return value of the fixture into the test function through that parameter. (Note that fixtures don't **need** to return a value; they can do other setup things instead, like calling an external resource, arranging things on the filesystem, putting values in a database, whatever the tests need for setup)

```
def test_foo_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 1 == my_stuff.foo
    my_stuff.foo = 3000
    assert my_stuff.foo == 3000
```

```
def test_bar_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Now you can see why we named it with a noun. but the `my_stuff = prepped_stuff` line is pretty much useless, so let's just use `prepped_stuff` directly instead.

```
def test_foo_updates(prepped_stuff):
    assert 1 == prepped_stuff.foo
    prepped_stuff.foo = 3000
    assert prepped_stuff.foo == 3000
```

```
def test_bar_updates(prepped_stuff):
    assert 2 == prepped_stuff.bar
    prepped_stuff.bar = 42
    assert 42 == prepped_stuff.bar
```

Now we're using fixtures! We can go further by changing the scope of the fixture (so it only runs once per test module or test suite execution session instead of once per test function), building fixtures that use other fixtures, parameterizing the fixture (so that the fixture and all tests using that fixture are run multiple times, once for each parameter given to the fixture), fixtures that read values from the module that calls them... as mentioned earlier, fixtures have a lot more power and flexibility than a normal setup function.

## Cleaning up after the tests are done.

Let's say our code has grown and our `Stuff` object now needs special clean up.

```
# projectroot/module/stuff.py
```

```
class Stuff(object):
def prep(self):
    self.foo = 1
    self.bar = 2

def finish(self):
    self.foo = 0
    self.bar = 0
```

我们可以在每个测试函数的底部添加一些代码来调用清理操作，但夹具（fixtures）提供了更好的方法来实现这一点。如果你在夹具中添加一个函数并将其注册为finalizer，finalizer函数中的代码将在使用该夹具的测试完成后被调用。如果夹具的作用域大于单个函数（例如模块或会话），finalizer将在该作用域内的所有测试完成后执行，也就是说，在模块运行结束或整个测试会话结束时执行。

```
@pytest.fixture
def prepped_stuff(request): # 我们需要传入request以使用finalizers
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    def fin(): # finalizer函数
        # 在这里执行所有清理操作
        my_stuff.finish()
    request.addfinalizer(fin) # 将fin()注册为finalizer
    # 如果你真的想，可以在这里做更多的设置
    return my_stuff
```

在函数内部使用finalizer函数一开始可能有点难以理解，尤其是当你有更复杂的夹具时。你也可以使用yield夹具来实现同样的功能，且执行流程更易于理解。唯一的区别是，我们不是使用return，而是在夹具完成设置并将控制权交给测试函数的部分使用yield，然后在yield之后添加所有清理代码。我们还将其装饰为yield\_fixture，以便py.test知道如何处理它。

```
@pytest.yield_fixture
def prepped_stuff(): # 现在不需要 request 了！
    # 执行初始化
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    # 初始化完成，交出控制权给测试函数
    yield my_stuff
    # 执行清理
    my_stuff.finish()
```

这就是测试夹具入门的全部内容！

更多信息请参见官方 [py.test 夹具文档](#) 和官方 [yield 夹具文档](#)

### 第 193.3 节：失败的测试

失败的测试会提供有用的输出，说明出了什么问题：

```
# projectroot/tests/test_code.py
from module import code

def test_add_failing():
    assert code.add(10, 11) == 33
```

```
class Stuff(object):
    def prep(self):
        self.foo = 1
        self.bar = 2

    def finish(self):
        self.foo = 0
        self.bar = 0
```

We could add some code to call the clean up at the bottom of every test function, but fixtures provide a better way to do this. If you add a function to the fixture and register it as a **finalizer**, the code in the finalizer function will get called after the test using the fixture is done. If the scope of the fixture is larger than a single function (like module or session), the finalizer will be executed after all the tests in scope are completed, so after the module is done running or at the end of the entire test running session.

```
@pytest.fixture
def prepped_stuff(request): # we need to pass in the request to use finalizers
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    def fin(): # finalizer function
        # do all the cleanup here
        my_stuff.finish()
    request.addfinalizer(fin) # register fin() as a finalizer
    # you can do more setup here if you really want to
    return my_stuff
```

Using the finalizer function inside a function can be a bit hard to understand at first glance, especially when you have more complicated fixtures. You can instead use a **yield fixture** to do the same thing with a more human readable execution flow. The only real difference is that instead of using `return` we use a `yield` at the part of the fixture where the setup is done and control should go to a test function, then add all the cleanup code after the `yield`. We also decorate it as a `yield_fixture` so that py.test knows how to handle it.

```
@pytest.yield_fixture
def prepped_stuff(): # it doesn't need request now!
    # do setup
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    # setup is done, pass control to the test functions
    yield my_stuff
    # do cleanup
    my_stuff.finish()
```

And that concludes the Intro to Test Fixtures!

For more information, see the [official py.test fixture documentation](#) and the [official yield fixture documentation](#)

### Section 193.3: Failing Tests

A failing test will provide helpful output as to what went wrong:

```
# projectroot/tests/test_code.py
from module import code

def test_add__failing():
    assert code.add(10, 11) == 33
```

结果：

```
$ py.test
=====
 test 会话开始
=====
平 台 darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
根目录: /projectroot, 配置文件:
收集了 1 个项目

tests/test_code.py F
=====
失败
=====
    test_add_failing

def test_add_failing():
>     断言 code.add(10, 11) == 33
E     断言 21 == 33
E     + 其中 21 = <函数 add 位于 0x105d4d6e0>(10, 11)
E     +     其中 <函数 add 位于 0x105d4d6e0> = code.add

tests/test_code.py:5: 断言错误
===== 1 失败 耗时 0.01 秒
=====
```

Results:

```
$ py.test
=====
 test session starts
=====
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, ini file:
collected 1 items

tests/test_code.py F
=====
 FAILURES
=====
    test_add_failing

def test_add_failing():
>     assert code.add(10, 11) == 33
E     assert 21 == 33
E     + where 21 = <function add at 0x105d4d6e0>(10, 11)
E     +     where <function add at 0x105d4d6e0> = code.add

tests/test_code.py:5: AssertionError
===== 1 failed in 0.01 seconds
=====
```

# 第194章：性能分析

## 第194.1节：IPython中的%%timeit和%timeit

字符串拼接的性能分析：

```
输入 [1]: 导入 string

输入 [2]: %%timeit s="" ; long_list=list(string.ascii_letters)*50
....: 对于 substring 在 long_list 中:
....:     ....:
....:     s+=substring
....:
1000 循环次数, 3次中最佳：每次循环570微秒

输入 [3]: %%timeit long_list=list(string.ascii_letters)*50
....: s="".join(long_list)
....:
100000 循环,3次中最佳: 每次循环16.1微秒
```

对可迭代对象和列表的循环进行性能分析：

```
在 [4]: %timeit for i in range(100000):pass
100 循环,3次中最佳: 每次循环2.82毫秒

在 [5]: %timeit for i in list(range(100000)):pass
100 循环,3次中最佳: 每次循环3.95毫秒
```

## 第194.2节：使用cProfile（首选性能分析器）

Python包含一个名为cProfile的性能分析器。通常推荐使用它而不是timeit。

它会分析整个脚本，并针对脚本中的每个方法告诉你：

- ncalls: 方法被调用的次数
- tottime: 在给定函数中花费的总时间（不包括调用子函数的时间）
- percall: 每次调用花费的时间。或者是 tottime 除以 ncalls 的商
- cumtime: 在该函数及所有子函数中累计花费的时间（从调用到退出）。即使是递归函数，该数值也是准确的。
- percall: 是 cumtime 除以原始调用次数的商
- filename:lineno(function): 提供每个函数的相应数据

cProfiler 可以通过命令行轻松调用：

```
$ python -m cProfile main.py
```

要按方法中花费的时间对返回的分析方法列表进行排序：

```
$ python -m cProfile -s time main.py
```

## 第194.3节：timeit() 函数

分析数组中元素的重复情况

```
>>> import timeit
```

# Chapter 194: Profiling

## Section 194.1: %%timeit and %timeit in IPython

Profiling string concatenation:

```
In [1]: import string

In [2]: %%timeit s="" ; long_list=list(string.ascii_letters)*50
....: for substring in long_list:
....:     s+=substring
....:
1000 loops, best of 3: 570 us per loop

In [3]: %%timeit long_list=list(string.ascii_letters)*50
....: s="".join(long_list)
....:
100000 loops, best of 3: 16.1 us per loop
```

Profiling loops over iterables and lists:

```
In [4]: %timeit for i in range(100000):pass
100 loops, best of 3: 2.82 ms per loop

In [5]: %timeit for i in list(range(100000)):pass
100 loops, best of 3: 3.95 ms per loop
```

## Section 194.2: Using cProfile (Preferred Profiler)

Python includes a profiler called cProfile. This is generally preferred over using timeit.

It breaks down your entire script and for each method in your script it tells you:

- ncalls: The number of times a method was called
- tottime: Total time spent in the given function (excluding time made in calls to sub-functions)
- percall: Time spent per call. Or the quotient of tottime divided by ncalls
- cumtime: The cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.
- percall: is the quotient of cumtime divided by primitive calls
- filename:lineno(function): provides the respective data of each function

The cProfiler can be easily called on Command Line using:

```
$ python -m cProfile main.py
```

To sort the returned list of profiled methods by the time taken in the method:

```
$ python -m cProfile -s time main.py
```

## Section 194.3: timeit() function

Profiling repetition of elements in an array

```
>>> import timeit
```

```
>>> timeit.timeit('list(itertools.repeat("a", 100))', 'import itertools', number = 10000000)
10.997665435877963
>>> timeit.timeit('["a"]*100', number = 10000000)
7.118789926862576
```

## 第194.4节：timeit命令行

数字连接的性能分析

```
python -m timeit "'-'join(str(n) for n in range(100))"
10000 循环, 3次中最佳: 29.2 微秒每次循环
```

```
python -m timeit "'-'join(map(str,range(100)))"
100000 循环, 3次中最佳: 19.4 微秒每次循环
```

## 第194.5节：命令行中的line\_profiler

带有@profile指令的源代码，放在我们想要分析的函数前：

```
import requests

@profile
def slow_func():
    s = requests.session()
    html=s.get("https://en.wikipedia.org/").text
    sum([pow(ord(x),3.1) for x in list(html)])

for i in range(50):
    slow_func()
```

使用kernprof命令逐行计算性能分析

```
$ kernprof -lv so6.py
已将分析结果写入 so6.py.lprof
计时单位：4.27654e-07 秒

总时间：22.6427 秒
文件：so6.py
函数：slow_func, 第4行

第 # 行 命中次数 每次命中时间 占用时间百分比 行内容
=====
4 @profile
5 定义 slow_func() 函数：
6 50 20729 414.6 0.0 s = requests.session()
7 50 47618627 952372.5 89.9 html=s.get("https://en.wikipedia.org/").text
8 50 5306958 106139.2 10.0 sum([pow(ord(x),3.1) for x in list(html)])
```

页面请求几乎总是比基于页面信息的任何计算都要慢。

```
>>> timeit.timeit('list(itertools.repeat("a", 100))', 'import itertools', number = 10000000)
10.997665435877963
>>> timeit.timeit('["a"]*100', number = 10000000)
7.118789926862576
```

## Section 194.4: timeit command line

Profiling concatenation of numbers

```
python -m timeit "'-'join(str(n) for n in range(100))"
10000 loops, best of 3: 29.2 usec per loop
```

```
python -m timeit "'-'join(map(str,range(100)))"
100000 loops, best of 3: 19.4 usec per loop
```

## Section 194.5: line\_profiler in command line

The source code with @profile directive before the function we want to profile:

```
import requests

@profile
def slow_func():
    s = requests.session()
    html=s.get("https://en.wikipedia.org/").text
    sum([pow(ord(x),3.1) for x in list(html)])

for i in range(50):
    slow_func()
```

Using kernprof command to calculate profiling line by line

```
$ kernprof -lv so6.py
Wrote profile results to so6.py.lprof
Timer unit: 4.27654e-07 s

Total time: 22.6427 s
File: so6.py
Function: slow_func at line 4

Line # Hits Time Per Hit % Time Line Contents
=====
4 @profile
5 def slow_func():
6 50 20729 414.6 0.0 s = requests.session()
7 50 47618627 952372.5 89.9 html=s.get("https://en.wikipedia.org/").text
8 50 5306958 106139.2 10.0 sum([pow(ord(x),3.1) for x in list(html)])
```

Page request is almost always slower than any calculation based on the information on the page.

# 视频：机器学习 A-Z：动手 Python数据科学

学习如何从两位数据科学专家那里用Python创建机器学习算法。包含代码模板。



- ✓ 精通Python上的机器学习
- ✓ 对多种机器学习模型有良好的直觉
- ✓ 做出准确的预测
- ✓ 进行强有力的分析
- ✓ 制作稳健的机器学习模型
- ✓ 为您的业务创造强大的附加价值
- ✓ 将机器学习用于个人目的
- ✓ 处理强化学习、自然语言处理和深度学习等特定主题
- ✓ 掌握降维等高级技术
- ✓ 了解针对每种问题应选择哪种机器学习模型
- ✓ 构建一支强大的机器学习模型军团，并知道如何组合它们以解决任何问题

今天观看 →

# VIDEO: Machine Learning A-Z: Hands-On Python In Data Science

Learn to create Machine Learning Algorithms in Python from two Data Science experts. Code templates included.



- ✓ Master Machine Learning on Python
- ✓ Have a great intuition of many Machine Learning models
- ✓ Make accurate predictions
- ✓ Make powerful analysis
- ✓ Make robust Machine Learning models
- ✓ Create strong added value to your business
- ✓ Use Machine Learning for personal purpose
- ✓ Handle specific topics like Reinforcement Learning, NLP and Deep Learning
- ✓ Handle advanced techniques like Dimensionality Reduction
- ✓ Know which Machine Learning model to choose for each type of problem
- ✓ Build an army of powerful Machine Learning models and know how to combine them to solve any problem

Watch Today →

# 第195章：Python程序速度

## 第195.1节：双端队列操作

双端队列是一种两端都可以操作的队列。

```
类 Deque :  
def __init__(self):  
    self.items = []  
  
def isEmpty(self):  
    return self.items == []  
  
def addFront(self, item):  
    self.items.append(item)  
  
def addRear(self, item):  
    self.items.insert(0,item)  
  
def removeFront(self):  
    return self.items.pop()  
  
def removeRear(self):  
    return self.items.pop(0)  
  
def size(self):  
    return len(self.items)
```

操作：平均情况（假设参数是随机生成的）

追加：O(1)

左侧追加：O(1)

复制：O(n)

扩展：O(k)

左侧扩展：O(k)

弹出：O(1)

左侧弹出：O(1)

移除：O(n)

旋转：O(k)

## 第195.2节：算法符号

任何计算机语言的优化都有一定的原则，Python也不例外。不要

**边写边优化**：编写程序时不要考虑可能的优化，重点是

确保代码干净、正确且易于理解。如果完成后程序太大或太慢，

那时你可以考虑进行优化。

记住80/20法则：在许多领域，你可以用20%的努力获得80%的结果（也称为

# Chapter 195: Python speed of program

## Section 195.1: Deque operations

A deque is a double-ended queue.

```
class Deque:  
def __init__(self):  
    self.items = []  
  
def isEmpty(self):  
    return self.items == []  
  
def addFront(self, item):  
    self.items.append(item)  
  
def addRear(self, item):  
    self.items.insert(0,item)  
  
def removeFront(self):  
    return self.items.pop()  
  
def removeRear(self):  
    return self.items.pop(0)  
  
def size(self):  
    return len(self.items)
```

Operations : Average Case (assumes parameters are randomly generated)

Append : O(1)

Appendleft : O(1)

Copy : O(n)

Extend : O(k)

Extendleft : O(k)

Pop : O(1)

Popleft : O(1)

Remove : O(n)

Rotate : O(k)

## Section 195.2: Algorithmic Notations

There are certain principles that apply to optimization in any computer language, and Python is no exception. **Don't**

**optimize as you go**: Write your program without regard to possible optimizations, concentrating instead on

making sure that the code is clean, correct, and understandable. If it's too big or too slow when you've finished,

then you can consider optimizing it.

**Remember the 80/20 rule**: In many fields you can get 80% of the result with 20% of the effort (also called the

90/10法则——这取决于你和谁交流）。每当你准备优化代码时，使用性能分析来找出那80%的执行时间花费在哪里，这样你就知道该把精力集中在哪。

始终运行“前后”基准测试：否则你怎么知道你的优化是否真的有效？如果优化后的代码仅比原版稍快或稍小，撤销你的更改，回到原本清晰的代码。

使用合适的算法和数据结构：不要用 $O(n^2)$ 的冒泡排序算法去排序一千个元素，当有 $O(n \log n)$ 的快速排序可用时。同样，不要将一千个项目存储在需要 $O(n)$ 搜索的数组中，当你可以使用 $O(\log n)$ 的二叉树，或者 $O(1)$ 的Python哈希表时。

更多内容请访问以下链接... [Python加速](#)

以下三种渐近符号主要用于表示算法的时间复杂度。

1.  $\Theta$  符号：Theta 符号对函数进行上下界的限制，因此它定义了精确的渐近行为。

获得表达式的 Theta 符号的简单方法是去掉低阶项并忽略前导常数。例如，考虑以下表达式。 $3n^3 + 6n^2 + 6000 = \Theta(n^3)$  去掉低阶项总是可以的，因为总会存在一个 $n_0$ ，使得  $\Theta(n^3)$  的值大于  $\Theta(n^2)$

无论涉及的常数如何。对于给定函数 $g(n)$ ，我们表示  $\Theta(g(n))$  是以下函数集合。 $\Theta(g(n)) = \{f(n) : \text{存在正常数 } c_1, c_2 \text{ 和 } n_0, \text{ 使得对所有 } n \geq n_0 \text{ 有 } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$  上述定义意味着，如果 $f(n)$ 是 $g(n)$ 的 theta，则 $f(n)$ 的值在大 $n$ 值 ( $n \geq n_0$ ) 时总是在 $c_1 g(n)$ 和 $c_2 g(n)$ 之间。theta 的定义还要求 $f(n)$ 在 $n > n_0$ 时必须非负。

2. 大O符号：大O 符号定义了算法的上界，它只对函数进行上界限制。

例如，考虑插入排序的情况。它在最好情况下是线性时间，最坏情况下是二次时间。我们可以安全地说插入排序的时间复杂度是 $O(n^2)$ 。注意 $O(n^2)$ 也包含线性时间。如果我们用 $\Theta$  符号表示插入排序的时间复杂度，则需要分别用两条语句表示最好和最坏情况：

1. 插入排序的最坏情况时间复杂度是  $\Theta(n^2)$ 。

2. 插入排序的最佳时间复杂度是 $\Theta(n)$ 。

大O 符号在我们仅知道算法时间复杂度的上界时非常有用。很多时候，我们可以通过简单地观察算法轻松找到一个上界。 $O(g(n)) = \{f(n) : \text{存在正常数 } c \text{ 和 } n_0, \text{ 使得对于所有 } n \geq n_0, \text{ 有 } 0 \leq f(n) \leq cg(n)\}$

0.  $\Omega$  符号：正如大O 符号提供函数的渐近上界， $\Omega$  符号提供

一个渐近下界。 $\Omega$  符号在我们知道算法时间复杂度的下界时非常有用。如前所述，算法的最好情况性能通常没有太大用处，Omega 符号是三种符号中使用最少的。对于给定函数 $g(n)$ ，我们用

$\Omega(g(n))$  表示函数集。 $\Omega(g(n)) = \{f(n) : \text{存在正常数 } c \text{ 和 } n_0, \text{ 使得对于所有 } n \geq n_0, \text{ 有 } 0 \leq cg(n) \leq f(n)\}$ 。这里我们考虑同样的插入排序例子。插入排序的时间复杂度可以写成 $\Omega(n)$ ，但这对插入排序来说不是很有用的信息，因为我们通常更关心最坏情况，有时也关心平均情况。

## 第195.3节：符号

### 基本概念

描述 Python 程序运行速度时使用的符号称为大O 符号。假设你有一个函数：

90/10 rule - it depends on who you talk to). Whenever you're about to optimize code, use profiling to find out where that 80% of execution time is going, so you know where to concentrate your effort.

**Always run "before" and "after" benchmarks:** How else will you know that your optimizations actually made a difference? If your optimized code turns out to be only slightly faster or smaller than the original version, undo your changes and go back to the original, clear code.

Use the right algorithms and data structures: Don't use an  $O(n^2)$  bubble sort algorithm to sort a thousand elements when there's an  $O(n \log n)$  quicksort available. Similarly, don't store a thousand items in an array that requires an  $O(n)$  search when you could use an  $O(\log n)$  binary tree, or an  $O(1)$  Python hash table.

For more visit the link below... [Python Speed Up](#)

The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

1.  **$\Theta$  Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behavior. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.  $3n^3 + 6n^2 + 6000 = \Theta(n^3)$  Dropping lower order terms is always fine because there will always be a  $n_0$  after which  $\Theta(n^3)$  has higher values than  $\Theta(n^2)$  irrespective of the constants involved. For a given function  $g(n)$ , we denote  $\Theta(g(n))$  is following set of functions.  $\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$  The above definition means, if  $f(n)$  is theta of  $g(n)$ , then the value  $f(n)$  is always between  $c_1 g(n)$  and  $c_2 g(n)$  for large values of  $n$  ( $n \geq n_0$ ). The definition of theta also requires that  $f(n)$  must be non-negative for values of  $n$  greater than  $n_0$ .

2. **Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is  $O(n^2)$ . Note that  $O(n^2)$  also covers linear time. If we use  $\Theta$  notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst case time complexity of Insertion Sort is  $\Theta(n^2)$ .
2. The best case time complexity of Insertion Sort is  $\Theta(n)$ .

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.  $O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$

0.  **$\Omega$  Notation:** Just as Big O notation provides an asymptotic upper bound on a function,  $\Omega$  notation provides an asymptotic lower bound.  $\Omega$  Notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three. For a given function  $g(n)$ , we denote by  $\Omega(g(n))$  the set of functions.  $\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$ . Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as  $\Omega(n)$ , but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

## Section 195.3: Notation

### Basic Idea

The notation used when describing the speed of your Python program is called Big-O notation. Let's say you have a function:

```

def list_check(to_check, the_list):
    for item in the_list:
        if to_check == item:
            return True
    return False

```

这是一个简单的函数，用于检查某个元素是否在列表中。描述该函数的复杂度时，通常说是  $O(n)$ 。这意味着“n阶”，因为  $O$  函数被称为阶函数。

$O(n)$  - 通常  $n$  是容器中元素的数量

$O(k)$  - 通常  $k$  是参数的值或参数中元素的数量

## 第195.4节：列表操作

操作：平均情况（假设参数是随机生成的）

追加 :  $O(1)$

复制 :  $O(n)$

删除切片 :  $O(n)$

删除元素 :  $O(n)$

插入 :  $O(n)$

获取元素 :  $O(1)$

设置元素 :  $O(1)$

迭代 :  $O(n)$

获取切片 :  $O(k)$

设置切片 :  $O(n + k)$

扩展 :  $O(k)$

排序 :  $O(n \log n)$

乘法 :  $O(nk)$

$x$  是否在  $s$  中 :  $O(n)$

最小值( $s$ )、最大值( $s$ ) :  $O(n)$

获取长度 :  $O(1)$

## 第195.5节：集合操作

操作：平均情况（假设参数随机生成）：最坏情况

$x$  在  $s$  中 :  $O(1)$

差集  $s - t$  :  $O(\text{len}(s))$

```

def list_check(to_check, the_list):
    for item in the_list:
        if to_check == item:
            return True
    return False

```

This is a simple function to check if an item is in a list. To describe the complexity of this function, you will say  $O(n)$ . This means "Order of n" as the  $O$  function is known as the Order function.

$O(n)$  - generally  $n$  is the number of items in container

$O(k)$  - generally  $k$  is the value of the parameter or the number of elements in the parameter

## Section 195.4: List operations

Operations : Average Case (assumes parameters are randomly generated)

Append :  $O(1)$

Copy :  $O(n)$

Del slice :  $O(n)$

Delete item :  $O(n)$

Insert :  $O(n)$

Get item :  $O(1)$

Set item :  $O(1)$

Iteration :  $O(n)$

Get slice :  $O(k)$

Set slice :  $O(n + k)$

Extend :  $O(k)$

Sort :  $O(n \log n)$

Multiply :  $O(nk)$

$x$  in  $s$  :  $O(n)$

$\min(s), \max(s)$  :  $O(n)$

Get length :  $O(1)$

## Section 195.5: Set operations

Operation : Average Case (assumes parameters generated randomly) : Worst case

$x$  in  $s$  :  $O(1)$

Difference  $s - t$  :  $O(\text{len}(s))$

交集 `s&t` :  $O(\min(\text{len}(s), \text{len}(t)))$  :  $O(\text{len}(s) * \text{len}(t))$

多重交集 `s1&s2&s3&...&sn` :  $(n-1) * O(l)$ , 其中  $l$  是  $\max(\text{len}(s1), \dots, \text{len}(sn))$

差集更新 `s.difference_update(t)` :  $O(\text{len}(t))$  :  $O(\text{len}(t) * \text{len}(s))$

`s.symmetric_difference_update(t)` :  $O(\text{len}(t))$

对称差集 `s^t` :  $O(\text{len}(s))$  :  $O(\text{len}(s) * \text{len}(t))$

并集 `s|t` :  $O(\text{len}(s) + \text{len}(t))$

Intersection `s&t` :  $O(\min(\text{len}(s), \text{len}(t)))$  :  $O(\text{len}(s) * \text{len}(t))$

Multiple intersection `s1&s2&s3&...&sn` :  $(n-1) * O(l)$  where  $l$  is  $\max(\text{len}(s1), \dots, \text{len}(sn))$

`s.difference_update(t)` :  $O(\text{len}(t))$  :  $O(\text{len}(t) * \text{len}(s))$

`s.symmetric_difference_update(t)` :  $O(\text{len}(t))$

Symmetric difference `s^t` :  $O(\text{len}(s))$  :  $O(\text{len}(s) * \text{len}(t))$

Union `s|t` :  $O(\text{len}(s) + \text{len}(t))$

# 第196章：性能优化

## 第196.1节：代码性能分析

首先，你应该能够找到脚本的瓶颈，并且要注意，没有任何优化能够弥补数据结构选择不当或算法设计缺陷。其次，不要在编码过程中过早地进行优化，以免牺牲代码的可读性、设计和质量。唐纳德·克努斯（Donald Knuth）曾对优化发表过如下声明：

“我们应该忽略小的效率提升，大约97%的时间都是如此：过早优化是万恶之源。但我们也不应错过那关键的3%的优化机会。”

要对代码进行性能分析，你可以使用多种工具：标准库中的cProfile（或较慢的profile）、line\_profiler和timeit。它们各自有不同的用途。

cProfile 是一个确定性分析器：函数调用、函数返回和异常事件都会被监控，并对这些事件之间的时间间隔进行精确计时（精度可达0.001秒）。该库的文档

([\[1\]](https://docs.python.org/2/library/profile.html)) 为我们提供了一个简单的使用示例

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

或者如果你更喜欢包裹现有代码的部分：

```
import cProfile, pstats, StringIO
pr = cProfile.Profile()
pr.enable()
# ... 执行一些操作 ...
# ... 很长时间 ...
pr.disable()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

这将生成如下表格形式的输出，您可以快速看到程序大部分时间花费的位置，并识别需要优化的函数。

3 次函数调用，耗时 0.000 秒

按标准名称排序  
ncalls tottime percall cumtime percall filename:lineno(function)  
1 0.000 0.000 0.000 :1(f)  
1 0.000 0.000 0.000 :1()  
1 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

模块 line\_profiler ([\[1\]](https://github.com/rkern/line_profiler)) 可用于对代码进行逐行分析。显然，这对于长脚本来说不可行，但适用于代码片段。更多详情请参阅文档。最简单的入门方法是使用包页面中说明的 kernprof 脚本，注意您需要手动指定要分析的函数。

# Chapter 196: Performance optimization

## Section 196.1: Code profiling

First and foremost you should be able to find the bottleneck of your script and note that no optimization can compensate for a poor choice in data structure or a flaw in your algorithm design. Secondly do not try to optimize too early in your coding process at the expense of readability/design/quality. Donald Knuth made the following statement on optimization:

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"

To profile your code you have several tools: cProfile (or the slower [profile](#)) from the standard library, line\_profiler and [timeit](#). Each of them serve a different purpose.

cProfile is a deterministic profiler: function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (up to 0.001s). The library documentation ([\[1\]](https://docs.python.org/2/library/profile.html)) provides us with a simple use case

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

Or if you prefer to wrap parts of your existing code:

```
import cProfile, pstats, StringIO
pr = cProfile.Profile()
pr.enable()
# ... do something ...
# ... long ...
pr.disable()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

This will create outputs looking like the table below, where you can quickly see where your program spends most of its time and identify the functions to optimize.

3 function calls in 0.000 seconds

Ordered by: standard name  
ncalls tottime percall cumtime percall filename:lineno(function)  
1 0.000 0.000 0.000 0.000 :1(f)  
1 0.000 0.000 0.000 0.000 :1()  
1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}

The module line\_profiler ([\[1\]](https://github.com/rkern/line_profiler)) is useful to have a line by line analysis of your code. This is obviously not manageable for long scripts but is aimed at snippets. See the documentation for more details. The easiest way to get started is to use the kernprof script as explained one the package page, note that you will need to specify manually the function(s) to profile.

```
$ kernprof -l script_to_profile.py
```

kernprof 将创建一个 LineProfiler 实例并将其插入到`__builtins__`命名空间中，名称为 profile。它被设计为装饰器使用，因此在你的脚本中，你可以用`@profile`装饰你想要分析的函数。

```
@profile  
def slow_function(a, b, c):  
    ...
```

kernprof 的默认行为是将结果保存到一个二进制文件`script_to_profile.py.lprof`中。你可以通过`[-v--view]`选项告诉kernprof 立即在终端查看格式化结果。否则，你可以稍后这样查看结果：

```
$ python -m line_profiler script_to_profile.py.lprof
```

最后，`timeit`提供了一种简单的方法来测试单行代码或小表达式，无论是在命令行还是 python 交互环境中。该模块可以回答诸如，将集合转换为列表时，使用列表推导式是否比内置的`list()`更快等问题。请查找`setup`关键字或`-s`选项以添加初始化代码。

```
>>> import timeit  
>>> timeit.timeit('"-'.join(str(n) for n in range(100))', number=10000)  
0.8187260627746582
```

来自终端

```
$ python -m timeit '"-'.join(str(n) for n in range(100))'  
10000 loops, best of 3: 40.3 usec per loop
```

```
$ kernprof -l script_to_profile.py
```

kernprof will create an instance of LineProfiler and insert it into the `__builtins__` namespace with the name `profile`. It has been written to be used as a decorator, so in your script, you decorate the functions you want to profile with `@profile`.

```
@profile  
def slow_function(a, b, c):  
    ...
```

The default behavior of kernprof is to put the results into a binary file `script_to_profile.py.lprof`. You can tell kernprof to immediately view the formatted results at the terminal with the `[-v--view]` option. Otherwise, you can view the results later like so:

```
$ python -m line_profiler script_to_profile.py.lprof
```

Finally `timeit` provides a simple way to test one liners or small expression both from the command line and the python shell. This module will answer question such as, is it faster to do a list comprehension or use the built-in `list()` when transforming a set into a list. Look for the `setup` keyword or `-s` option to add setup code.

```
>>> import timeit  
>>> timeit.timeit('"-'.join(str(n) for n in range(100))', number=10000)  
0.8187260627746582
```

from a terminal

```
$ python -m timeit '"-'.join(str(n) for n in range(100))'  
10000 loops, best of 3: 40.3 usec per loop
```

# 第197章：安全与密码学

Python作为计算机和网络安全中最流行的语言之一，在安全和密码学方面具有巨大潜力。本章节涉及Python中密码学的特性和实现，内容涵盖其在计算机和网络安全中的应用，以及哈希和加密/解密算法。

## 第197.1节：安全密码哈希

由`hashlib`模块提供的PBKDF2算法可用于执行安全的密码哈希。虽然该算法无法防止通过暴力破解攻击来恢复存储的哈希中的原始密码，但它使得此类攻击代价非常高昂。

```
import hashlib
import os

salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2可以与任何摘要算法配合使用，上述示例使用了通常推荐的SHA256。随机盐值应与哈希密码一起存储，您在比较输入密码与存储哈希时需要再次使用它。每个密码必须使用不同的盐进行哈希。至于迭代次数，建议根据您的应用将其设置得尽可能高。

如果您想要十六进制结果，可以使用`binascii`模块：

```
import binascii
hexhash = binascii.hexlify(hash)
```

注意：虽然 PBKDF2 并不差，`bcrypt` 尤其是 `scrypt` 被认为对抗暴力破解攻击更强。  
目前这两者都不是 Python 标准库的一部分。

## 第197.2节：计算消息摘要

`hashlib` 模块允许通过 `new` 方法创建消息摘要生成器。这些生成器可以将任意字符串转换为固定长度的摘要：

```
import hashlib

h = hashlib.new('sha256')
h.update(b'Nobody expects the Spanish Inquisition.')
h.digest()
# ==> b'.\xd9\xda\xdaVR[\x12\x90\xff\x16\xfb\x17D\xcf\xb4\x82\xdd)\x14\xff\xbc\xb6Iy\x0c\x0eX\x9eF='
```

注意，在调用 `digest` 之前，你可以任意次数调用 `update`，这对于分块哈希大文件非常有用。你也可以通过使用 `hexdigest` 来获取十六进制格式的摘要：

```
h.hexdigest()
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d'
```

## 第197.3节：可用的哈希算法

调用`hashlib.new`时需要提供算法名称以生成生成器。要查看当前Python解释器中可用的算法，请使用 `hashlib.algorithms_available`：

# Chapter 197: Security and Cryptography

Python, being one of the most popular languages in computer and network security, has great potential in security and cryptography. This topic deals with the cryptographic features and implementations in Python from its uses in computer and network security to hashing and encryption/decryption algorithms.

## Section 197.1: Secure Password Hashing

The [PBKDF2 algorithm](#) exposed by `hashlib` module can be used to perform secure password hashing. While this algorithm cannot prevent brute-force attacks in order to recover the original password from the stored hash, it makes such attacks very expensive.

```
import hashlib
import os

salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2 can work with any digest algorithm, the above example uses SHA256 which is usually recommended. The random salt should be stored along with the hashed password, you will need it again in order to compare an entered password to the stored hash. It is essential that each password is hashed with a different salt. As to the number of rounds, it is recommended to set it [as high as possible for your application](#).

If you want the result in hexadecimal, you can use the `binascii` module:

```
import binascii
hexhash = binascii.hexlify(hash)
```

Note: While PBKDF2 isn't bad, `bcrypt` and especially `scrypt` are considered stronger against brute-force attacks. Neither is part of the Python standard library at the moment.

## Section 197.2: Calculating a Message Digest

The `hashlib` module allows creating message digest generators via the `new` method. These generators will turn an arbitrary string into a fixed-length digest:

```
import hashlib

h = hashlib.new('sha256')
h.update(b'Nobody expects the Spanish Inquisition.')
h.digest()
# ==> b'.\xd9\xda\xdaVR[\x12\x90\xff\x16\xfb\x17D\xcf\xb4\x82\xdd)\x14\xff\xbc\xb6Iy\x0c\x0eX\x9eF='
```

Note that you can call `update` an arbitrary number of times before calling `digest` which is useful to hash a large file chunk by chunk. You can also get the digest in hexadecimal format by using `hexdigest`:

```
h.hexdigest()
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d'
```

## Section 197.3: Available Hashing Algorithms

`hashlib.new` requires the name of an algorithm when you call it to produce a generator. To find out what algorithms are available in the current Python interpreter, use `hashlib.algorithms_available`:

```
import hashlib
hashlib.algorithms_available
# ==> {'sha256', 'DSA-SHA', 'SHA512', 'SHA224', 'dsaWithSHA', 'SHA', 'RIPEMD160', 'ecdsa-with-SHA1',
'sha1', 'SHA384', 'md5', 'SHA1', 'MD5', 'MD4', 'SHA256', 'sha384', 'md4', 'ripemd160', 'sha224',
'sha512', 'DSA', 'dsaEncryption', 'sha', 'whirlpool'}
```

返回的列表会根据平台和解释器不同而有所变化；请确保检查你的算法是否可用。

还有一些算法是保证在所有平台和解释器上可用的，可以通过 `hashlib.algorithms_guaranteed` 获得：

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

## 第197.4节：文件哈希

哈希是一种将可变长度的字节序列转换为固定长度序列的函数。对文件进行哈希处理有许多优点。哈希值可以用来检查两个文件是否相同，或者验证文件内容是否未被损坏或更改。

你可以使用 `hashlib` 为文件生成哈希值：

```
import hashlib

hasher = hashlib.new('sha256')
使用 open('myfile', 'r') 作为 f:
contents = f.read()
hasher.update(contents)

print hasher.hexdigest()
```

对于较大的文件，可以使用固定长度的缓冲区：

```
import hashlib
SIZE = 65536
hasher = hashlib.new('sha256')
使用 open('myfile', 'r') 作为 f:
buffer = f.read(SIZE)
while len(buffer) > 0:
hasher.update(buffer)
    buffer = f.read(SIZE)
print(hasherhexdigest())
```

## 第197.5节：使用pycrypto生成RSA签名

RSA 可用于创建消息签名。只有拥有私有RSA密钥才能生成有效签名，而验证则仅需对应的公钥。因此，只要对方知道你的公钥，他们就可以验证消息确实由你签名且未被篡改——例如电子邮件中使用的这种方法。目前，实现此功能需要第三方模块如pycrypto。

```
import errno

from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
```

```
import hashlib
hashlib.algorithms_available
# ==> {'sha256', 'DSA-SHA', 'SHA512', 'SHA224', 'dsaWithSHA', 'SHA', 'RIPEMD160', 'ecdsa-with-SHA1',
'sha1', 'SHA384', 'md5', 'SHA1', 'MD5', 'MD4', 'SHA256', 'sha384', 'md4', 'ripemd160', 'sha224',
'sha512', 'DSA', 'dsaEncryption', 'sha', 'whirlpool'}
```

The returned list will vary according to platform and interpreter; make sure you check your algorithm is available.

还有一些算法是保证在所有平台和解释器上可用的，可以通过 `hashlib.algorithms_guaranteed` 获得：

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

## Section 197.4: File Hashing

A hash is a function that converts a variable length sequence of bytes to a fixed length sequence. Hashing files can be advantageous for many reasons. Hashes can be used to check if two files are identical or verify that the contents of a file haven't been corrupted or changed.

You can use `hashlib` to generate a hash for a file:

```
import hashlib

hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    contents = f.read()
    hasher.update(contents)

print hasher.hexdigest()
```

For larger files, a buffer of fixed length can be used:

```
import hashlib
SIZE = 65536
hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    buffer = f.read(SIZE)
    while len(buffer) > 0:
        hasher.update(buffer)
        buffer = f.read(SIZE)
print(hasherhexdigest())
```

## Section 197.5: Generating RSA signatures using pycrypto

RSA can be used to create a message signature. A valid signature can only be generated with access to the private RSA key, validating on the other hand is possible with merely the corresponding public key. So as long as the other side knows your public key they can verify the message to be signed by you and unchanged - an approach used for email for example. Currently, a third-party module like [pycrypto](#) is required for this functionality.

```
import errno

from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
```

```
message = b'这条消息来自我，我保证。'
```

尝试:

```

with open('privkey.pem', 'r') as f:
    key = RSA.importKey(f.read())
except IOError as e:
    if e.errno != errno.ENOENT:
        raise
    # 没有私钥，生成一个新的。这可能需要几秒钟。
    key = RSA.generate(4096)
    with open('privkey.pem', 'wb') as f:
        f.write(key.exportKey('PEM'))
    with open('pubkey.pem', 'wb') as f:
        f.write(key.publickey().exportKey('PEM'))

hasher = SHA256.new(message)
signer = PKCS1_v1_5.new(key)
signature = signer.sign(hasher)

```

验证签名的过程类似，但使用的是公钥而非私钥：

```

with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
    hasher = SHA256.new(message)
    verifier = PKCS1_v1_5.new(key)
    if verifier.verify(hasher, signature):
        print('太好了，签名有效！')
    else:
        print('不，消息是用错误的私钥签名的或已被篡改')

```

注意：上述示例使用的是非常常见的 PKCS#1 v1.5 签名算法。pycrypto 也实现了较新的 PKCS#1 PSS 算法，如果你想使用该算法，只需将示例中的 PKCS1\_v1\_5 替换为 PKCS1\_PSS 即可。目前似乎几乎没有理由使用它。

## 第197.6节：使用 pycrypto 进行非对称 RSA 加密

非对称加密的优点是消息可以在不与接收方交换密钥的情况下加密。发送方只需知道接收方的公钥，这允许以只有指定接收方（拥有对应私钥）才能解密的方式加密消息。

目前，实现此功能需要第三方模块，如 [pycrypto](#)。

```

from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

message = b'这是一条非常机密的消息。'

with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
    cipher = PKCS1_OAEP.new(key)
    encrypted = cipher.encrypt(message)

```

如果接收方拥有正确的私钥，则可以解密消息：

```

with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
    cipher = PKCS1_OAEP.new(key)
    decrypted = cipher.decrypt(encrypted)

```

```
message = b'This message is from me, I promise.'
```

try:

```

with open('privkey.pem', 'r') as f:
    key = RSA.importKey(f.read())
except IOError as e:
    if e.errno != errno.ENOENT:
        raise
    # No private key, generate a new one. This can take a few seconds.
    key = RSA.generate(4096)
    with open('privkey.pem', 'wb') as f:
        f.write(key.exportKey('PEM'))
    with open('pubkey.pem', 'wb') as f:
        f.write(key.publickey().exportKey('PEM'))

hasher = SHA256.new(message)
signer = PKCS1_v1_5.new(key)
signature = signer.sign(hasher)

```

Verifying the signature works similarly but uses the public key rather than the private key:

```

with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
    hasher = SHA256.new(message)
    verifier = PKCS1_v1_5.new(key)
    if verifier.verify(hasher, signature):
        print('Nice, the signature is valid!')
    else:
        print('No, the message was signed with the wrong private key or modified')

```

Note: The above examples use PKCS#1 v1.5 signing algorithm which is very common. pycrypto also implements the newer PKCS#1 PSS algorithm, replacing PKCS1\_v1\_5 by PKCS1\_PSS in the examples should work if you want to use that one. Currently there seems to be [little reason to use it](#) however.

## Section 197.6: Asymmetric RSA encryption using pycrypto

Asymmetric encryption has the advantage that a message can be encrypted without exchanging a secret key with the recipient of the message. The sender merely needs to know the recipient's public key, this allows encrypting the message in such a way that only the designated recipient (who has the corresponding private key) can decrypt it. Currently, a third-party module like [pycrypto](#) is required for this functionality.

```

from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

message = b'This is a very secret message.'

with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
    cipher = PKCS1_OAEP.new(key)
    encrypted = cipher.encrypt(message)

```

The recipient can decrypt the message then if they have the right private key:

```

with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
    cipher = PKCS1_OAEP.new(key)
    decrypted = cipher.decrypt(encrypted)

```

注意: 上述示例使用的是 PKCS#1 OAEP 加密方案。pycrypto 也实现了 PKCS#1 v1.5 加密方案，但由于已知的缺陷，不推荐在新协议中使用该方案。

## 第197.7节：使用 pycrypto 的对称加密

Python 内置的加密功能目前仅限于哈希。加密需要第三方模块，如 [pycrypto](#)。例如，它提供了[AES算法](#)，被认为是对称加密的先进技术。以下代码将使用密码短语加密给定的消息：

```
import hashlib
import math
import os

from Crypto.Cipher import AES

IV_SIZE = 16      # 128 位, AES 算法固定长度
KEY_SIZE = 32     # 256 位, 表示 AES-256, 也可以是 128 或 192 位
SALT_SIZE = 16    # 此大小为任意值

cleartext = b'Lorem ipsum'
password = b'高度安全的加密密码'
salt = os.urandom(SALT_SIZE)
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                               dklen=IV_SIZE + KEY_SIZE)

iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]

encrypted = salt + AES.new(key, AES.MODE_CFB, iv).encrypt(cleartext)
```

AES 算法需要三个参数：加密密钥、初始化向量（IV）和实际要加密的消息。如果你有一个随机生成的 AES 密钥，那么可以直接使用该密钥，并仅生成一个随机的初始化向量。然而，密码短语的长度不合适，且不建议直接使用，因为它不是真正随机的，因此熵较低。相反，我们使用内置的 PBKDF2 算法实现，从密码生成一个 128 位的初始化向量和 256 位的加密密钥。

请注意随机盐（salt），它对于每条加密消息生成不同的初始化向量和密钥非常重要。这特别确保了两个相同的消息不会产生相同的加密文本，同时也防止攻击者将猜测一个密码短语所花费的工作重复用于其他密码短语加密的消息。该盐必须与加密消息一起存储，以便在解密时推导出相同的初始化向量和密钥。

下面的代码将再次解密我们的消息：

```
salt = encrypted[0:SALT_SIZE]
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                               dklen=IV_SIZE + KEY_SIZE)

iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]
cleartext = AES.new(key, AES.MODE_CFB, iv).decrypt(encrypted[SALT_SIZE:])
```

Note: The above examples use PKCS#1 OAEP encryption scheme. pycrypto also implements PKCS#1 v1.5 encryption scheme, this one is not recommended for new protocols however due to [known caveats](#).

## Section 197.7: Symmetric encryption using pycrypto

Python's built-in crypto functionality is currently limited to hashing. Encryption requires a third-party module like [pycrypto](#). For example, it provides the [AES algorithm](#) which is considered state of the art for symmetric encryption. The following code will encrypt a given message using a passphrase:

```
import hashlib
import math
import os

from Crypto.Cipher import AES

IV_SIZE = 16      # 128 bit, fixed for the AES algorithm
KEY_SIZE = 32     # 256 bit meaning AES-256, can also be 128 or 192 bits
SALT_SIZE = 16    # This size is arbitrary

cleartext = b'Lorem ipsum'
password = b'highly secure encryption password'
salt = os.urandom(SALT_SIZE)
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                               dklen=IV_SIZE + KEY_SIZE)

iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]

encrypted = salt + AES.new(key, AES.MODE_CFB, iv).encrypt(cleartext)
```

The AES algorithm takes three parameters: encryption key, initialization vector (IV) and the actual message to be encrypted. If you have a randomly generated AES key then you can use that one directly and merely generate a random initialization vector. A passphrase doesn't have the right size however, nor would it be recommendable to use it directly given that it isn't truly random and thus has comparably little entropy. Instead, we use the built-in implementation of the PBKDF2 algorithm to generate a 128 bit initialization vector and 256 bit encryption key from the password.

Note the random salt which is important to have a different initialization vector and key for each message encrypted. This ensures in particular that two equal messages won't result in identical encrypted text, but it also prevents attackers from reusing work spent guessing one passphrase on messages encrypted with another passphrase. This salt has to be stored along with the encrypted message in order to derive the same initialization vector and key for decrypting.

The following code will decrypt our message again:

```
salt = encrypted[0:SALT_SIZE]
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                               dklen=IV_SIZE + KEY_SIZE)

iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]
cleartext = AES.new(key, AES.MODE_CFB, iv).decrypt(encrypted[SALT_SIZE:])
```

# 第198章：Python中的安全Shell连接

## 参数

hostname 该参数指定需要建立连接的主机

username 访问主机所需的用户名

端口 主机端口

password 账户密码

## 用法

### 第198.1节：ssh连接

```
from paramiko import client
ssh = client.SSHClient() # 创建一个新的SSHClient对象
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) # 自动接受未知主机密钥
ssh.connect(hostname, username=username, port=port, password=password) # 连接主机
stdin, stdout, stderr = ssh.exec_command(command) # 向ssh提交命令
print stdout.channel.recv_exit_status() #告诉状态 1 - 任务失败
```

# Chapter 198: Secure Shell Connection in Python

## Parameter

hostname This parameter tells the host to which the connection needs to be established

username username required to access the host

port host port

password password for the account

## Usage

### Section 198.1: ssh connection

```
from paramiko import client
ssh = client.SSHClient() # create a new SSHClient object
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) #auto-accept unknown host keys
ssh.connect(hostname, username=username, port=port, password=password) #connect with a host
stdin, stdout, stderr = ssh.exec_command(command) # submit a command to ssh
print stdout.channel.recv_exit_status() #tells the status 1 - job failed
```

# 第199章：Python反模式

## 第199.1节：过度热心的except子句

异常处理很强大，但一个过度热心的except子句可能在一行代码中破坏一切。

```
try: res = get_result() res = res[0] log('获得结果: %r' % res) except: if not res: res = " print('捕获异常')
```

这个例子展示了反模式的三个症状：

1. 没有指定异常类型的except（第5行）会捕获所有异常，包括KeyboardInterrupt。  
这会导致程序在某些情况下无法退出。
2. except块没有重新抛出错误，这意味着我们无法判断异常是来自get\_result内部，还是因为res是空列表。
3. 最糟糕的是，如果我们担心结果为空，我们反而引发了更严重的问题。如果get\_result失败，res将完全未被设置，而except块中对res的引用会引发NameError，完全掩盖了原始错误。

始终考虑你想处理的异常类型。阅读异常页面，了解基本异常的种类。

这是上述示例的修正版：

```
import traceback try: res = get_result() except Exception: log_exception(traceback.format_exc()) raise try: res = res[0] except IndexError: res = " log('got result: %r' % res)
```

我们捕获了更具体的异常，并在必要时重新抛出。代码多了几行，但正确得多。

## 第199.2节：在调用处理器密集型函数前先三思

程序很容易因为多次调用处理器密集型函数而浪费时间。

例如，考虑这样一个函数：如果输入value能产生一个整数，则返回该整数，否则

返回None：

```
def intensive_f(value): # int -> Optional[int]
    # 复杂且耗时的代码
    if process_has_failed:
        return None
    return integer_output
```

它可以按以下方式使用：

```
x = 5
if intensive_f(x) 不是 None:
    print(intensive_f(x) / 2)
else:
    print(x, "无法处理")

print(x)
```

虽然这样可以工作，但存在调用intensive\_f的问题，这会使代码运行时间加倍。更好的解决方案是事先获取函数的返回值。

# Chapter 199: Python Anti-Patterns

## Section 199.1: Overzealous except clause

Exceptions are powerful, but a single overzealous except clause can take it all away in a single line.

```
try: res = get_result() res = res[0] log('got result: %r' % res) except: if not res: res = " print('got exception')
```

This example demonstrates 3 symptoms of the antipattern:

1. The **except** with no exception type (line 5) will catch even healthy exceptions, including [KeyboardInterrupt](#). That will prevent the program from exiting in some cases.
2. The except block does not reraise the error, meaning that we won't be able to tell if the exception came from within `get_result` or because `res` was an empty list.
3. Worst of all, if we were worried about `result` being empty, we've caused something much worse. If `get_result` fails, `res` will stay completely unset, and the reference to `res` in the except block, will raise [NameError](#), completely masking the original error.

Always think about the type of exception you're trying to handle. Give [the exceptions page a read](#) and get a feel for what basic exceptions exist.

Here is a fixed version of the example above:

```
import traceback try: res = get_result() except Exception: log_exception(traceback.format_exc()) raise try: res = res[0] except IndexError: res = " log('got result: %r' % res)
```

We catch more specific exceptions, reraising where necessary. A few more lines, but infinitely more correct.

## Section 199.2: Looking before you leap with processor-intensive function

A program can easily waste time by calling a processor-intensive function multiple times.

For example, take a function which looks like this: it returns an integer if the input value can produce one, else `None`:

```
def intensive_f(value): # int -> Optional[int]
    # complex, and time-consuming code
    if process_has_failed:
        return None
    return integer_output
```

And it could be used in the following way:

```
x = 5
if intensive_f(x) is not None:
    print(intensive_f(x) / 2)
else:
    print(x, "could not be processed")

print(x)
```

Whilst this will work, it has the problem of calling `intensive_f`, which doubles the length of time for the code to run. A better solution would be to get the return value of the function beforehand.

```
x = 5
result = intensive_f(x)
if result 不是 None:
    print(result / 2)
else:
    print(x, "无法处理")
```

然而，更清晰且可能更符合Python风格的方法是使用异常，例如：

```
x = 5
尝试:
    print(intensive_f(x) / 2)
except TypeError: # 如果尝试 None + 1 会引发的异常
    print(x, "无法处理")
```

这里不需要临时变量。通常更倾向于使用assert语句，并捕获AssertionError异常。

## 字典键

一个常见的例子是在访问字典键时。例如比较：

```
bird_speeds = get_very_long_dictionary()

if "european swallow" in bird_speeds:
    speed = bird_speeds["european swallow"]
else:
    speed = input("一只未负重的燕子的空速是多少？")

print(speed)
```

含有：

```
bird_speeds = get_very_long_dictionary()

尝试:
speed = bird_speeds["european swallow"]
except KeyError:
speed = input("一只未负重的燕子的空速是多少？")

print(speed)
```

第一个例子需要两次遍历字典，由于字典很长，每次遍历可能会花费较长时间。第二个例子只需遍历一次字典，因此节省了大量处理器时间。

另一种方法是使用 `dict.get(key, default)`，但在键不存在的情况下，许多情况可能需要执行更复杂的操作

```
x = 5
result = intensive_f(x)
if result is not None:
    print(result / 2)
else:
    print(x, "could not be processed")
```

However, a clearer and [possibly more pythonic way](#) is to use exceptions, for example:

```
x = 5
try:
    print(intensive_f(x) / 2)
except TypeError: # The exception raised if None + 1 is attempted
    print(x, "could not be processed")
```

Here no temporary variable is needed. It may often be preferable to use a `assert` statement, and to catch the `AssertionError` instead.

## Dictionary keys

A common example of where this may be found is accessing dictionary keys. For example compare:

```
bird_speeds = get_very_long_dictionary()

if "european swallow" in bird_speeds:
    speed = bird_speeds["european swallow"]
else:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

with:

```
bird_speeds = get_very_long_dictionary()

try:
    speed = bird_speeds["european swallow"]
except KeyError:
    speed = input("What is the air-speed velocity of an unladen swallow?")

print(speed)
```

The first example has to look through the dictionary twice, and as this is a long dictionary, it may take a long time to do so each time. The second only requires one search through the dictionary, and thus saves a lot of processor time.

An alternative to this is to use `dict.get(key, default)`, however many circumstances may require more complex operations to be done in the case that the key is not present

# 视频：机器学习、数据科学和使用 Python 的深度学习

完整的动手机器学习教程，涵盖  
数据科学、Tensorflow、人工智能  
和神经网络



- ✓ 使用Tensorflow和Keras构建人工神经网络
- ✓ 使用深度学习对图像、数据和情感进行分类
- ✓ 使用线性回归、多项式回归和多元回归进行预测
- ✓ 使用Matplotlib和Seaborn进行数据可视化
- ✓ 使用 Apache Spark 的 MLLib 实现大规模机器学习
- ✓ 理解强化学习——以及如何构建一个吃豆人机器人
- ✓ 使用 K-Means 聚类、支持向量机 (SVM) 、KNN、决策树、朴素贝叶斯和主成分分析 (PCA) 对数据进行分类
- ✓ 使用训练/测试和 K 折交叉验证来选择和调整模型
- ✓ 使用基于物品和基于用户的协同过滤构建电影推荐系统

今天观看 →

# VIDEO: Machine Learning, Data Science and Deep Learning with Python

Complete hands-on machine learning tutorial with data science, Tensorflow, artificial intelligence, and neural networks



- ✓ Build artificial neural networks with Tensorflow and Keras
- ✓ Classify images, data, and sentiments using deep learning
- ✓ Make predictions using linear regression, polynomial regression, and multivariate regression
- ✓ Data Visualization with Matplotlib and Seaborn
- ✓ Implement machine learning at massive scale with Apache Spark's MLLib
- ✓ Understand reinforcement learning - and how to build a Pac-Man bot
- ✓ Classify data using K-Means clustering, Support Vector Machines (SVM), KNN, Decision Trees, Naive Bayes, and PCA
- ✓ Use train/test and K-Fold cross validation to choose and tune your models
- ✓ Build a movie recommender system using item-based and user-based collaborative filtering

Watch Today →

# 第200章：常见陷阱

Python 是一种旨在清晰易读、无歧义和意外行为的语言。

不幸的是，这些目标并非在所有情况下都能实现，因此 Python 确实存在一些边缘情况，可能会做出与你预期不同的行为。

本节将向你展示编写 Python 代码时可能遇到的一些问题。

## 第200.1节：列表乘法与常见引用问题

考虑通过乘法创建嵌套列表结构的情况：

```
li = [[]] * 3
print(li)
# 输出: [[], [], []]
```

乍一看，我们会认为这是一个包含3个不同嵌套列表的列表。让我们尝试向第一个

列表中追加1：

```
li[0].append(1)
print(li)
# 输出: [[1], [1], [1]]
```

1结果是1被追加到了li中的所有列表中。

原因是`[] * 3`并没有创建3个不同的列表。相反，它创建了一个列表，包含3个指向同一个列表对象的引用。因此，当我们向`li[0]`追加元素时，所有`li`的子元素都会显示出该变化。这等同于：

```
li = []
元素 = []
li = 元素 + 元素 + 元素
print(li)
# 输出: [[], [], []]
element.append(1)
print(li)
# 输出: [[1], [1], [1]]
```

如果我们使用 `id` 打印包含的列表的内存地址，可以进一步证实这一点：

```
li = [[]] * 3
print([id(inner_list) for inner_list in li])
# 输出: [6830760, 6830760, 6830760]
```

解决方案是用循环创建内部列表：

```
li = [[] for _ in range(3)]
```

我们不再创建一个单一的列表然后引用它3次，而是创建3个不同的独立列表。这一点同样可以通过使用 `id` 函数验证：

```
print([id(inner_list) for inner_list in li])
# 输出: [6331048, 6331528, 6331488]
```

# Chapter 200: Common Pitfalls

Python is a language meant to be clear and readable without any ambiguities and unexpected behaviors. Unfortunately, these goals are not achievable in all cases, and that is why Python does have a few corner cases where it might do something different than what you were expecting.

This section will show you some issues that you might encounter when writing Python code.

## Section 200.1: List multiplication and common references

Consider the case of creating a nested list structure by multiplying:

```
li = [[]] * 3
print(li)
# Out: [[], [], []]
```

At first glance we would think we have a list of containing 3 different nested lists. Let's try to append 1 to the first one:

```
li[0].append(1)
print(li)
# Out: [[1], [1], [1]]
```

1 got appended to all of the lists in `li`.

The reason is that `[] * 3` doesn't create a `list` of 3 different `lists`. Rather, it creates a `list` holding 3 references to the same `list` object. As such, when we append to `li[0]` the change is visible in all sub-elements of `li`. This is equivalent of:

```
li = []
element = []
li = element + element + element
print(li)
# Out: [[], [], []]
element.append(1)
print(li)
# Out: [[1], [1], [1]]
```

This can be further corroborated if we print the memory addresses of the contained `list` by using `id`:

```
li = [[]] * 3
print([id(inner_list) for inner_list in li])
# Out: [6830760, 6830760, 6830760]
```

The solution is to create the inner lists with a loop:

```
li = [[] for _ in range(3)]
```

Instead of creating a single `list` and then making 3 references to it, we now create 3 different distinct lists. This, again, can be verified by using the `id` function:

```
print([id(inner_list) for inner_list in li])
# Out: [6331048, 6331528, 6331488]
```

你也可以这样做。它会在每次 `append` 调用时创建一个新的空列表。

```
>>> li = []
>>> li.append([])
>>> li.append([])
>>> li.append([])
>>> li.append([])
...
...>>> for k in li: print(id(k))
```

You can also do this. It causes a new empty list to be created in each `append` call.

```
>>> li = []
>>> li.append([])
>>> li.append([])
>>> li.append([])
>>> li.append([])
...
...>>> for k in li: print(id(k))
...
4315469256
4315564552
4315564808
```

Don't use index to loop over a sequence.

**Don't:**

```
for i in range(len(tab)):
    print(tab[i])
```

**Do:**

```
for elem in tab:
    print(elem)
```

`for` will automate most iteration operations for you.

**Use enumerate if you really need both the index and the element.**

```
for i, elem in enumerate(tab):
    print((i, elem))
```

**Be careful when using "==" to check against True or False**

```
for i, elem in enumerate(tab):
    print((i, elem))
```

**使用 "==" 检查 True 或 False 时要小心**

```
if (var == True):
    # 如果 var 是 True 或 1、1.0、1L，则会执行此代码

if (var != True):
    # 如果 var 既不是 True 也不是 1，则会执行此代码

if (var == False):
    # 当 var 为 False 或 0 (或 0.0、0L、0j) 时执行

if (var == None):
    # 仅当 var 为 None 时执行

if var:
    # 当 var 是非空字符串/列表/字典/元组，非0等时执行

if not var:
    # 当 var 是 ""、{}、[]、()、0、None 等时执行

if var is True:
    # 仅当 var 是布尔值 True (非数字 1) 时执行

if var is False:
    # 仅当变量为布尔值 False 时执行，不包括 0

if var is None:
```

```
if (var == True):
    # this will execute if var is True or 1, 1.0, 1L

if (var != True):
    # this will execute if var is neither True nor 1

if (var == False):
    # this will execute if var is False or 0 (or 0.0, 0L, 0j)

if (var == None):
    # only execute if var is None

if var:
    # execute if var is a non-empty string/list/dictionary/tuple, non-0, etc

if not var:
    # execute if var is "", {}, [], (), 0, None, etc.

if var is True:
    # only execute if var is boolean True, not 1

if var is False:
    # only execute if var is boolean False, not 0

if var is None:
```

```
# 等同于 var == None
```

## 不要检查是否可以，直接执行并处理错误

Python 程序员通常说“请求原谅比请求许可更容易”。

**不要：**

```
if os.path.isfile(file_path):
    file = open(file_path)
else:
    # 执行某些操作
```

**要：**

**尝试：**

```
file = open(file_path)
except OSError as e:
    # 执行某些操作
```

或者更好的是使用Python2.6+：

```
with open(file_path) as file:
```

这更好，因为它更通用。你几乎可以对任何东西使用 **try/except**。你不需要关心如何预防错误，只需关心你可能遇到的错误。

## 不要检查类型

Python 是动态类型语言，因此检查类型会让你失去灵活性。相反，使用鸭子类型通过检查行为。如果你期望函数中传入字符串，则使用str()将任何对象转换为字符串。如果你期望传入列表，则使用list()将任何可迭代对象转换为列表。

**不要：**

```
def foo(name):
    if isinstance(name, str):
        print(name.lower())

def bar(listing):
    if isinstance(listing, list):
        listing.extend((1, 2, 3))
    return ", ".join(listing)
```

**要：**

```
def foo(name) :
    print(str(name).lower())

def bar(listing) :
    l = list(listing)
    l.extend((1, 2, 3))
    return ", ".join(l)
```

使用最后一种方式，foo 将接受任何对象。bar 将接受字符串、元组、集合、列表以及更多类型。廉价的DRY（不要重复自己）。

## 不要混用空格和制表符

```
# same as var == None
```

## Do not check if you can, just do it and handle the error

Pythonistas usually say "It's easier to ask for forgiveness than permission".

**Don't:**

```
if os.path.isfile(file_path):
    file = open(file_path)
else:
    # do something
```

**Do:**

```
try:
    file = open(file_path)
except OSError as e:
    # do something
```

Or even better with Python 2.6+:

```
with open(file_path) as file:
```

It is much better because it is much more generic. You can apply **try/except** to almost anything. You don't need to care about what to do to prevent it, just care about the error you are risking.

## Do not check against type

Python is dynamically typed, therefore checking for type makes you lose flexibility. Instead, use [duck typing](#) by checking behavior. If you expect a string in a function, then use **str()** to convert any object to a string. If you expect a list, use **list()** to convert any iterable to a list.

**Don't:**

```
def foo(name):
    if isinstance(name, str):
        print(name.lower())

def bar(listing):
    if isinstance(listing, list):
        listing.extend((1, 2, 3))
    return ", ".join(listing)
```

**Do:**

```
def foo(name) :
    print(str(name).lower())

def bar(listing) :
    l = list(listing)
    l.extend((1, 2, 3))
    return ", ".join(l)
```

Using the last way, foo will accept any object. bar will accept strings, tuples, sets and much more. Cheap DRY.

## Don't mix spaces and tabs

## 使用object作为第一个父级

这很棘手，但随着你的程序变大，它会给你带来麻烦。在Python2.x中有旧式和新式类。旧式类嘛，就是旧的。它们缺少一些特性，并且在继承时可能表现得很尴尬。为了可用，你的任何类都必须是“新式”的。为此，让它继承自object。

不要：

```
class Father:  
    pass  
  
class Child(Father):  
    pass
```

要：

```
class Father(object):  
    pass  
  
class Child(Father):  
    pass
```

在Python3.x中，所有类都是新式的，所以你不需要这样做。

## 不要在init方法外初始化类属性

来自其他语言的人会觉得这样做很诱人，因为在Java或PHP中就是这么做的。你写类名，然后列出属性并给它们一个默认值。在Python中这似乎也能工作，然而，事实并非你想象的那样。这样做会设置类属性（静态属性），然后当你尝试获取对象属性时，它会返回对象属性的值，除非该值为空。在那种情况下，它会返回类属性。这意味着两个重大风险：

- 如果类属性被修改，那么初始值也会被改变。
- 如果你将可变对象设为默认值，你将得到在实例间共享的同一个对象。

不要这样做（除非你想要静态属性）：

```
class Car(object):  
    color = "red"  
    wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

要：

```
class Car(object):  
    def __init__(self):  
        self.color = "red"  
        self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

## 第200.2节：可变默认参数

```
def foo(li=[]):  
    li.append(1)  
    print(li)
```

## Use object as first parent

This is tricky, but it will bite you as your program grows. There are old and new classes in Python 2.x. The old ones are, well, old. They lack some features, and can have awkward behavior with inheritance. To be usable, any of your class must be of the "new style". To do so, make it inherit from `object`.

**Don't:**

```
class Father:  
    pass  
  
class Child(Father):  
    pass
```

**Do:**

```
class Father(object):  
    pass  
  
class Child(Father):  
    pass
```

In Python 3.x all classes are new style so you don't need to do that.

## Don't initialize class attributes outside the init method

People coming from other languages find it tempting because that is what you do in Java or PHP. You write the class name, then list your attributes and give them a default value. It seems to work in Python, however, this doesn't work the way you think. Doing that will setup class attributes (static attributes), then when you will try to get the object attribute, it will give you its value unless it's empty. In that case it will return the class attributes. It implies two big hazards:

- If the class attribute is changed, then the initial value is changed.
- If you set a mutable object as a default value, you'll get the same object shared across instances.

**Don't (unless you want static):**

```
class Car(object):  
    color = "red"  
    wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

**Do:**

```
class Car(object):  
    def __init__(self):  
        self.color = "red"  
        self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

## Section 200.2: Mutable default argument

```
def foo(li=[]):  
    li.append(1)  
    print(li)
```

```
foo([2])  
# 输出: [2, 1]  
foo([3])  
# 输出: [3, 1]
```

这段代码表现如预期，但如果我们不传入参数会怎样？

```
foo()  
# 输出: [1] 如预期...  
  
foo()  
# 输出: [1, 1] 不如预期...
```

这是因为函数和方法的默认参数是在定义时而非运行时被求值的。  
所以我们实际上只有一个li列表的实例。

解决方法是只使用不可变类型作为默认参数：

```
def foo(li=None):  
    if not li:  
        li = []  
        li.append(1)  
    print(li)  
  
foo()  
# Out: [1]  
  
foo()  
# Out: [1]
```

虽然有所改进，且if not li能正确评估为False，但许多其他对象也会如此，例如零长度序列。以下示例参数可能导致意外结果：

```
x = []  
foo(li=x)  
# Out: [1]  
  
foo(li="")  
# Out: [1]  
  
foo(li=0)  
# Out: [1]
```

惯用做法是直接将参数与None对象进行比较：

```
def foo(li=None):  
    if li is None:  
        li = []  
        li.append(1)  
    print(li)  
  
foo()  
# Out: [1]
```

## 第200.3节：更改正在迭代的序列

一个for循环遍历一个序列，因此在循环内部修改该序列可能导致意想不到的结果  
(尤其是在添加或删除元素时)：

```
foo([2])  
# Out: [2, 1]  
foo([3])  
# Out: [3, 1]
```

This code behaves as expected, but what if we don't pass an argument?

```
foo()  
# Out: [1] As expected...  
  
foo()  
# Out: [1, 1] Not as expected...
```

This is because default arguments of functions and methods are evaluated at **definition** time rather than run time.  
So we only ever have a single instance of the li list.

The way to get around it is to use only immutable types for default arguments:

```
def foo(li=None):  
    if not li:  
        li = []  
        li.append(1)  
    print(li)  
  
foo()  
# Out: [1]  
  
foo()  
# Out: [1]
```

While an improvement and although if not li correctly evaluates to False, many other objects do as well, such as zero-length sequences. The following example arguments can cause unintended results:

```
x = []  
foo(li=x)  
# Out: [1]  
  
foo(li="")  
# Out: [1]  
  
foo(li=0)  
# Out: [1]
```

The idiomatic approach is to directly check the argument against the None object:

```
def foo(li=None):  
    if li is None:  
        li = []  
        li.append(1)  
    print(li)  
  
foo()  
# Out: [1]
```

## Section 200.3: Changing the sequence you are iterating over

A for loop iterates over a sequence, so **altering this sequence inside the loop could lead to unexpected results** (especially when adding or removing elements):

```

alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Out: [1]

```

注意：list.pop() 用于从列表中移除元素。

第二个元素没有被删除，因为迭代是按索引顺序进行的。上述循环迭代了两次，结果如下：

```

# 第1次迭代
index = 0
alist = [0, 1, 2]
alist.pop(0) # 移除 '0'

# 第2次迭代
index = 1
alist = [1, 2]
alist.pop(1) # 移除 '2'

# 循环终止，但 alist 仍不为空：
alist = [1]

```

这个问题产生的原因是索引在按递增方向迭代时发生了变化。为避免这个问题，你可以**反向迭代循环**：

```

alist = [1,2,3,4,5,6,7]
for index, item in reversed(list(enumerate(alist))):
    # 删除所有偶数项
    if item % 2 == 0:
        alist.pop(index)
print(alist)
# 输出: [1, 3, 5, 7]

```

通过从末尾开始迭代循环，当项目被移除（或添加）时，不会影响列表中较早项目的索引。因此，这个例子将正确地从list中移除所有偶数项。

**当你在迭代一个列表时插入或追加元素时，会出现类似的问题，这可能导致无限循环：**

```

alist = [0, 1, 2]
for index, value in enumerate(alist):
    # 使用break避免无限循环：
    if index == 20:
        break
    alist.insert(index, 'a')
print(alist)
# 输出 (简略) :['a', 'a', ..., 'a', 'a', 0, 1, 2]

```

如果没有break条件，循环会一直插入'a'，直到计算机内存耗尽或程序被强制停止。在这种情况下，通常更倾向于创建一个新列表，并在遍历原列表时将项目添加到新列表中。

**使用for循环时，不能通过占位变量修改列表元素：**

```
alist = [1,2,3,4]
```

```

alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Out: [1]

```

Note: list.pop() is being used to remove elements from the list.

The second element was not deleted because the iteration goes through the indices in order. The above loop iterates twice, with the following results:

```

# Iteration #1
index = 0
alist = [0, 1, 2]
alist.pop(0) # removes '0'

# Iteration #2
index = 1
alist = [1, 2]
alist.pop(1) # removes '2'

# loop terminates, but alist is not empty:
alist = [1]

```

This problem arises because the indices are changing while iterating in the direction of increasing index. To avoid this problem, you can **iterate through the loop backwards**:

```

alist = [1,2,3,4,5,6,7]
for index, item in reversed(list(enumerate(alist))):
    # delete all even items
    if item % 2 == 0:
        alist.pop(index)
print(alist)
# Out: [1, 3, 5, 7]

```

By iterating through the loop starting at the end, as items are removed (or added), it does not affect the indices of items earlier in the list. So this example will properly remove all items that are even from alist.

A similar problem arises when **inserting or appending elements to a list that you are iterating over**, which can result in an infinite loop:

```

alist = [0, 1, 2]
for index, value in enumerate(alist):
    # break to avoid infinite loop:
    if index == 20:
        break
    alist.insert(index, 'a')
print(alist)
# Out (abbreviated): ['a', 'a', ..., 'a', 'a', 0, 1, 2]

```

Without the **break** condition the loop would insert 'a' as long as the computer does not run out of memory and the program is allowed to continue. In a situation like this, it is usually preferred to create a new list, and add items to the new list as you loop through the original list.

When using a **for** loop, **you cannot modify the list elements with the placeholder variable**:

```
alist = [1,2,3,4]
```

```
for item in alist:  
    if item % 2 == 0:  
        item = '偶数'  
print(alist)  
# 输出: [1,2,3,4]
```

在上面的例子中，改变item实际上并不会改变原始列表中的任何内容。你需要使用列表索引（alist[2]），enumerate()对此很有帮助：

```
alist = [1,2,3,4]  
for index, item in enumerate(alist):  
    if item % 2 == 0:  
        alist[index] = '偶数'  
print(alist)  
# 输出: [1, '偶数', 3, '偶数']
```

在某些情况下，while循环可能是更好的选择：

如果你打算删除列表中的所有项目：

```
zlist = [0, 1, 2]  
while zlist:  
    print(zlist[0])  
    zlist.pop(0)  
print('After: zlist =', zlist)  
  
# 输出: 0  
#     1  
#     2  
# After: zlist = []
```

虽然简单地重置zlist也能达到相同的效果；

```
zlist = []
```

上述示例也可以结合len()使用，以在某个点停止，或删除列表中除 x 个项目之外的所有项目：

```
zlist = [0, 1, 2]  
x = 1  
while len(zlist) > x:  
    print(zlist[0])  
    zlist.pop(0)  
print('After: zlist =', zlist)  
  
# 输出: 0  
#     1  
# After: zlist = [2]
```

或者循环遍历列表，同时删除满足某个条件的元素（在本例中删除所有偶数元素）：

```
zlist = [1,2,3,4,5]  
i = 0  
while i < len(zlist):  
    if zlist[i] % 2 == 0:  
        zlist.pop(i)  
    else:
```

```
for item in alist:  
    if item % 2 == 0:  
        item = 'even'  
print(alist)  
# Out: [1,2,3,4]
```

In the above example, changing item doesn't actually change anything in the original list. You need to use the list index (alist[2]), and enumerate() works well for this:

```
alist = [1,2,3,4]  
for index, item in enumerate(alist):  
    if item % 2 == 0:  
        alist[index] = 'even'  
print(alist)  
# Out: [1, 'even', 3, 'even']
```

A while loop might be a better choice in some cases:

If you are going to delete all the items in the list:

```
zlist = [0, 1, 2]  
while zlist:  
    print(zlist[0])  
    zlist.pop(0)  
print('After: zlist =', zlist)  
  
# Out: 0  
#     1  
#     2  
# After: zlist = []
```

Although simply resetting zlist will accomplish the same result;

```
zlist = []
```

The above example can also be combined with len() to stop after a certain point, or to delete all but x items in the list:

```
zlist = [0, 1, 2]  
x = 1  
while len(zlist) > x:  
    print(zlist[0])  
    zlist.pop(0)  
print('After: zlist =', zlist)  
  
# Out: 0  
#     1  
# After: zlist = [2]
```

Or to loop through a list while deleting elements that meet a certain condition (in this case deleting all even elements):

```
zlist = [1,2,3,4,5]  
i = 0  
while i < len(zlist):  
    if zlist[i] % 2 == 0:  
        zlist.pop(i)  
    else:
```

```
i += 1
print(zlist)
# Out: [1, 3, 5]
```

注意，删除元素后不要增加 `i`。通过删除位于 `zlist[i]` 的元素，下一项的索引已经减少了一个，因此在下一次迭代中用相同的 `i` 值检查 `zlist[i]` 时，您将正确地检查列表中的下一项。

一种去除列表中不需要项的相反思路是将需要的项添加到一个新列表中。以下示例是前面 `while` 循环示例的另一种替代方案：

```
zlist = [1, 2, 3, 4, 5]

z_temp = []
for item in zlist:
    if item % 2 != 0:
        z_temp.append(item)
zlist = z_temp
print(zlist)
# Out: [1, 3, 5]
```

这里我们将所需结果汇集到一个新列表中。然后我们可以选择将临时列表重新赋值给原始变量。

基于这种思路，你可以调用 Python 最优雅且强大的特性之一，**列表推导式**，它消除了临时列表，并且不同于之前讨论的原地列表/索引变异理念。

```
zlist = [1, 2, 3, 4, 5]
[item for item in zlist if item % 2 != 0]
# 输出: [1, 3, 5]
```

## 第200.4节：整数和字符串的身份

Python 使用内部缓存机制来缓存一系列整数，以减少重复创建它们时的必要开销。

实际上，这可能导致在比较整数身份时出现令人困惑的行为：

```
>>> -8 is (-7 - 1)
False
>>> -3 is (-2 - 1)
True
```

另一个例子：

```
>>> (255 + 1) is (255 + 1)
True
>>> (256 + 1) is (256 + 1)
False
```

等等，什么情况？

我们可以看到，身份操作符 `is` 对某些整数 (-3, 256) 返回 `True`，但对其他整数 (-8, 257) 则不然。

更具体地说，解释器启动时，范围在 [-5, 256] 内的整数会被内部缓存，并且只创建一次。因此，它们是 相同的，使用 `is` 比较它们的身份会返回 `True`；而超出此范围的整数

```
i += 1
print(zlist)
# Out: [1, 3, 5]
```

Notice that you don't increment `i` after deleting an element. By deleting the element at `zlist[i]`, the index of the next item has decreased by one, so by checking `zlist[i]` with the same value for `i` on the next iteration, you will be correctly checking the next item in the list.

A contrary way to think about removing unwanted items from a list, is to **add wanted items to a new list**. The following example is an alternative to the latter `while` loop example:

```
zlist = [1, 2, 3, 4, 5]

z_temp = []
for item in zlist:
    if item % 2 != 0:
        z_temp.append(item)
zlist = z_temp
print(zlist)
# Out: [1, 3, 5]
```

Here we are funneling desired results into a new list. We can then optionally reassign the temporary list to the original variable.

With this trend of thinking, you can invoke one of Python's most elegant and powerful features, **list comprehensions**, which eliminates temporary lists and diverges from the previously discussed in-place list/index mutation ideology.

```
zlist = [1, 2, 3, 4, 5]
[item for item in zlist if item % 2 != 0]
# Out: [1, 3, 5]
```

## Section 200.4: Integer and String identity

Python uses internal caching for a range of integers to reduce unnecessary overhead from their repeated creation.

In effect, this can lead to confusing behavior when comparing integer identities:

```
>>> -8 is (-7 - 1)
False
>>> -3 is (-2 - 1)
True
```

and, using another example:

```
>>> (255 + 1) is (255 + 1)
True
>>> (256 + 1) is (256 + 1)
False
```

Wait what?

We can see that the identity operation `is` yields `True` for some integers (-3, 256) but no for others (-8, 257).

To be more specific, integers in the range [-5, 256] are internally cached during interpreter startup and are only created once. As such, they are **identical** and comparing their identities with `is` yields `True`; integers outside this

range 通常是动态创建的，其身份比较结果为False。

这是一个常见的陷阱，因为这是测试中常用的范围，但代码在开发中运行完美后，往往会在后期部署阶段（甚至更糟——生产环境）无明显原因地失败。

解决方案是始终使用相等（==）运算符比较值，而不要使用身份（is）运算符。

Python 还会保留对常用字符串的引用，在比较字符串身份（即使用is）时也可能导致类似的混淆行为。

```
>>> 'python' is 'py' + 'thon'  
True
```

字符串'python'是常用的，因此 Python 只有一个对象，所有对字符串'python'的引用都指向该对象。

对于不常见的字符串，即使字符串相等，身份比较也会失败。

```
>>> 'this is not a common string' is 'this is not' + ' a common string'  
False  
>>> 'this is not a common string' == 'this is not' + ' a common string'  
True
```

因此，就像整数的规则一样，**始终使用相等（==）运算符比较字符串值，而不要使用身份（is）运算符**。

## 第200.5节：字典是无序的

你可能会期望Python字典像例如C++的std::map那样按键排序，但事实并非如此：

```
myDict = {'first': 1, 'second': 2, 'third': 3}  
print(myDict)  
# 输出: {'first': 1, 'second': 2, 'third': 3}  
  
print([k for k in myDict])  
# 输出: ['second', 'third', 'first']
```

Python没有任何内置类能自动按键对元素进行排序。

但是，如果排序不是必须的，你只是想让字典记住其键/值对的插入顺序，可以使用collections.OrderedDict：

```
from collections import OrderedDict  
  
oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])  
  
print([k for k in oDict])  
# 输出: ['first', 'second', 'third']
```

请记住，用标准字典初始化OrderedDict不会对字典进行任何排序。该结构所做的只是保留键的插入顺序。

Python 3.6 中字典的实现[发生了变化](#)，以改善其内存消耗。这种新实现的一个副作用是它还保留了传递给函数的关键字参数的顺序：

Python 3.x 版本 ≥ 3.6

range are (usually) created on-the-fly and their identities compare to False.

This is a common pitfall since this is a common range for testing, but often enough, the code fails in the later staging process (or worse - production) with no apparent reason after working perfectly in development.

The solution is to **always compare values using the equality** (==) operator and **not** the identity (is) operator.

Python also keeps references to commonly used strings and can result in similarly confusing behavior when comparing identities (i.e. using is) of strings.

```
>>> 'python' is 'py' + 'thon'  
True
```

The string 'python' is commonly used, so Python has one object that all references to the string 'python' use.

For uncommon strings, comparing identity fails even when the strings are equal.

```
>>> 'this is not a common string' is 'this is not' + ' a common string'  
False  
>>> 'this is not a common string' == 'this is not' + ' a common string'  
True
```

So, just like the rule for Integers, **always compare string values using the equality** (==) operator and **not** the identity (is) operator.

## Section 200.5: Dictionaries are unordered

You might expect a Python dictionary to be sorted by keys like, for example, a C++ std::map, but this is not the case:

```
myDict = {'first': 1, 'second': 2, 'third': 3}  
print(myDict)  
# Out: {'first': 1, 'second': 2, 'third': 3}  
  
print([k for k in myDict])  
# Out: ['second', 'third', 'first']
```

Python doesn't have any built-in class that automatically sorts its elements by key.

However, if sorting is not a must, and you just want your dictionary to remember the order of insertion of its key/value pairs, you can use `collections.OrderedDict`:

```
from collections import OrderedDict  
  
oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])  
  
print([k for k in oDict])  
# Out: ['first', 'second', 'third']
```

Keep in mind that initializing an OrderedDict with a standard dictionary won't sort in any way the dictionary for you. All that this structure does is to *preserve* the order of key insertion.

The implementation of dictionaries was [changed in Python 3.6](#) to improve their memory consumption. A side effect of this new implementation is that it also preserves the order of keyword arguments passed to a function:

Python 3.x Version ≥ 3.6

```
def func(**kw): print(kw.keys())
func(a=1, b=2, c=3, d=4, e=5)
dict_keys(['a', 'b', 'c', 'd', 'e']) # 预期顺序
```

**注意事项:** 请注意“[这种新实现的顺序保留特性被视为实现细节,不应依赖于此](#)”, 因为未来可能会发生变化。

## 第200.6节 : 列表推导式和for

[循环中的变量泄漏](#)

考虑以下列表推导式

Python 2.x 版本 ≤ 2.7

```
i = 0
a = [i for i in range(3)]
print(i) # 输出 2
```

这种情况仅发生在 Python 2 中, 因为列表推导式会“泄漏”循环控制变量到外部作用域 ([来源](#))。这种行为可能导致难以发现的错误, **并且在 Python 3 中已被修复**

Python 3.x 版本 ≥ 3.0

```
i = 0
a = [i for i in range(3)]
print(i) # 输出 0
```

同样, 对于循环, 其迭代变量没有私有作用域

```
i = 0
for i in range(3):
    通过
    print(i) # 输出 2
```

这种行为在 Python 2 和 Python 3 中都会发生。

为避免变量泄漏问题, 应在列表推导式和 for 循环中适当使用新的变量。

## 第 200.7 节 : 或运算符的链式调用

当测试多个相等比较中的任意一个时 :

```
if a == 3 or b == 3 or c == 3:
```

可能会想将其简写为

```
if a or b or c == 3: # 错误
```

这是错误的 ; or 运算符的优先级低于 ==, 因此表达式将被解析为 if (a) or (b) or (c == 3)。正确的做法是显式地检查所有条件 :

```
if a == 3 or b == 3 or c == 3: # 正确写法
```

或者, 可以使用内置的 any() 函数来替代链式的 or 运算符 :

```
def func(**kw): print(kw.keys())
```

```
func(a=1, b=2, c=3, d=4, e=5)
dict_keys(['a', 'b', 'c', 'd', 'e']) # expected order
```

**Caveat:** beware that “[the order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon](#)”, as it may change in the future.

## Section 200.6: Variable leaking in list comprehensions and for loops

Consider the following list comprehension

Python 2.x Version ≤ 2.7

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 2
```

This occurs only in Python 2 due to the fact that the list comprehension “leaks” the loop control variable into the surrounding scope ([source](#)). This behavior can lead to hard-to-find bugs and **it has been fixed in Python 3**.

Python 3.x Version ≥ 3.0

```
i = 0
a = [i for i in range(3)]
print(i) # Outputs 0
```

Similarly, for loops have no private scope for their iteration variable

```
i = 0
for i in range(3):
    通过
    pass
print(i) # Outputs 2
```

This type of behavior occurs both in Python 2 and Python 3.

To avoid issues with leaking variables, use new variables in list comprehensions and for loops as appropriate.

## Section 200.7: Chaining of or operator

When testing for any of several equality comparisons:

```
if a == 3 or b == 3 or c == 3:
```

it is tempting to abbreviate this to

```
if a or b or c == 3: # Wrong
```

This is wrong; the or operator has [lower precedence](#) than ==, so the expression will be evaluated as if (a) or (b) or (c == 3)。The correct way is explicitly checking all the conditions:

```
if a == 3 or b == 3 or c == 3: # Right Way
```

Alternately, the built-in any() function may be used in place of chained or operators:

```
if any([a == 3, b == 3, c == 3]): # 正确
```

或者，为了提高效率：

```
if any(x == 3 for x in (a, b, c)): # 正确
```

或者，为了更简洁：

```
if 3 in (a, b, c): # 正确
```

这里，我们使用 `in` 运算符来测试该值是否存在于包含我们想要比较的值的元组中。

类似地，写成下面这样是不正确的

```
if a == 1 or 2 or 3:
```

应该写成

```
if a in (1, 2, 3):
```

## 第200.8节：`sys.argv[0]` 是正在执行的文件名

`sys.argv[0]` 的第一个元素是正在执行的 Python 文件名。其余元素是脚本参数。

```
# script.py
import sys

print(sys.argv[0])
print(sys.argv)
```

```
$ python script.py
=> script.py
=> ['script.py']
```

```
$ python script.py fizz
=> script.py
=> ['script.py', 'fizz']
```

```
$ python script.py fizz buzz
=> script.py
=> ['script.py', 'fizz', 'buzz']
```

```
if any([a == 3, b == 3, c == 3]): # Right
```

Or, to make it more efficient:

```
if any(x == 3 for x in (a, b, c)): # Right
```

Or, to make it shorter:

```
if 3 in (a, b, c): # Right
```

Here, we use the `in` operator to test if the value is present in a tuple containing the values we want to compare against.

Similarly, it is incorrect to write

```
if a == 1 or 2 or 3:
```

which should be written as

```
if a in (1, 2, 3):
```

## Section 200.8: `sys.argv[0]` is the name of the file being executed

The first element of `sys.argv[0]` is the name of the python file being executed. The remaining elements are the script arguments.

```
# script.py
import sys

print(sys.argv[0])
print(sys.argv)
```

```
$ python script.py
=> script.py
=> ['script.py']
```

```
$ python script.py fizz
=> script.py
=> ['script.py', 'fizz']
```

```
$ python script.py fizz buzz
=> script.py
=> ['script.py', 'fizz', 'buzz']
```

## 第200.9节：访问整数字面量的属性

你可能听说过Python中的一切都是对象，甚至字面量也是。这意味着，例如，`7` 也是一个对象，这意味着它有属性。例如，其中一个属性是`bit_length`。它返回表示调用它的值所需的位数。

```
x = 7
x.bit_length()
# 输出: 3
```

```
x = 7
x.bit_length()
# Out: 3
```

## Section 200.9: Accessing int literals' attributes

You might have heard that everything in Python is an object, even literals. This means, for example, `7` is an object as well, which means it has attributes. For example, one of these attributes is the `bit_length`. It returns the amount of bits needed to represent the value it is called upon.

看到上述代码能运行，你可能直觉认为7.bit\_length()也能运行，结果却发现它抛出了SyntaxError。为什么？因为解释器需要区分属性访问和浮点数（例如7.2或7.bit\_length()）。它无法区分，因此抛出了异常。

有几种方法可以访问int字面量的属性：

```
# 括号  
(7).bit_length()  
# 一个空格  
7 .bit_length()
```

在这种情况下使用两个点（比如这样 7..bit\_length()）不起作用，因为这会创建一个float字面量，而浮点数没有bit\_length()方法。

当访问float字面量的属性时，这个问题不存在，因为解释器足够“智能”，知道一个float字面量不可能包含两个.，例如：

```
7.2.as_integer_ratio()  
# 输出: (8106479329266893, 1125899906842624)
```

## 第200.10节：全局解释器锁（GIL）和阻塞线程

关于Python的GIL已经有大量的讨论。它有时会在处理多线程（不要与多进程混淆）应用时引起困惑。

这里有一个例子：

```
import math  
from threading import Thread  
  
def calc_fact(num):  
    math.factorial(num)  
  
num = 600000  
t = Thread(target=calc_fact, daemon=True, args=[num])  
print("即将计算 : {}!".format(num))  
t.start()  
print("正在计算...")  
t.join()  
print("计算完成")
```

你会期望在启动线程后立即看到正在计算...被打印出来，毕竟我们希望计算是在新线程中进行的！但实际上，你会看到它在计算完成后才被打印出来。这是因为新线程依赖于一个C函数（math.factorial），该函数执行时会锁住全局解释器锁（GIL）。

有几种方法可以解决这个问题。第一种是用原生Python实现你的阶乘函数。这将允许主线程在你循环执行时获得控制权。缺点是这个解决方案会慢很多，因为我们不再使用C函数了。

```
def calc_fact(num):  
    """ 用原生Python实现的一个慢速阶乘版本 """  
    res = 1  
    while num >= 1:  
        res = res * num
```

Seeing the above code works, you might intuitively think that 7.bit\_length() would work as well, only to find out it raises a [SyntaxError](#). Why? because the interpreter needs to differentiate between an attribute access and a floating number (for example 7.2 or 7.bit\_length()). It can't, and that's why an exception is raised.

There are a few ways to access an [int](#) literals' attributes:

```
# parenthesis  
(7).bit_length()  
# a space  
7 .bit_length()
```

Using two dots (like this 7..bit\_length()) doesn't work in this case, because that creates a [float](#) literal and floats don't have the bit\_length() method.

This problem doesn't exist when accessing [float](#) literals' attributes since the interpreter is "smart" enough to know that a [float](#) literal can't contain two ., for example:

```
7.2.as_integer_ratio()  
# Out: (8106479329266893, 1125899906842624)
```

## Section 200.10: Global Interpreter Lock (GIL) and blocking threads

Plenty has been [written about Python's GIL](#). It can sometimes cause confusion when dealing with multi-threaded (not to be confused with multiprocessing) applications.

Here's an example:

```
import math  
from threading import Thread  
  
def calc_fact(num):  
    math.factorial(num)  
  
num = 600000  
t = Thread(target=calc_fact, daemon=True, args=[num])  
print("About to calculate: {}!".format(num))  
t.start()  
print("Calculating...")  
t.join()  
print("Calculated")
```

You would expect to see Calculating... printed out immediately after the thread is started, we wanted the calculation to happen in a new thread after all! But in actuality, you see it get printed after the calculation is complete. That is because the new thread relies on a C function ([math.factorial](#)) which will lock the GIL while it executes.

There are a couple ways around this. The first is to implement your factorial function in native Python. This will allow the main thread to grab control while you are inside your loop. The downside is that this solution will be **a lot** slower, since we're not using the C function anymore.

```
def calc_fact(num):  
    """ A slow version of factorial in native Python """  
    res = 1  
    while num >= 1:  
        res = res * num
```

```
num -= 1  
return res
```

你也可以在开始执行前sleep一段时间。注意：这实际上不会让你的程序中断C函数内部正在进行的计算，但它会让你的主线程在创建子线程后继续运行，这可能是你所期望的。

```
def calc_fact(num):  
    sleep(0.001)  
    math.factorial(num)
```

## 第200.11节：多重返回

函数xyz返回两个值a和b：

```
def xyz():  
    返回 a, b
```

代码调用 xyz 并将结果存储到一个变量中，假设 xyz 只返回一个值：

```
t = xyz()
```

变量 t 的值实际上是一个元组 (a, b)，因此如果对 t 进行操作时假设它不是元组，可能会在代码深处出现关于元组的意外错误。

```
| TypeError: type tuple doesn't define ... method
```

解决方法是：

```
a, b = xyz()
```

初学者仅凭元组错误信息很难找到该错误的原因！

## 第 200.12 节：Python 风格的 JSON 键

```
my_var = 'bla';  
api_key = 'key';  
...这里有大量的 代码 ...  
params = {"language": "en", my_var: api_key}
```

如果你习惯了JavaScript，Python字典中的变量求值不会像你预期的那样。JavaScript中的这条语句会生成如下的params对象：

```
{  
    "language": "en",  
    "my_var": "key"  
}
```

然而，在Python中，它会生成如下字典：

```
{  
    "language": "en",  
    "bla": "key"
```

```
num -= 1  
return res
```

You can also sleep for a period of time before starting your execution. Note: this won't actually allow your program to interrupt the computation happening inside the C function, but it will allow your main thread to continue after the spawn, which is what you may expect.

```
def calc_fact(num):  
    sleep(0.001)  
    math.factorial(num)
```

## Section 200.11: Multiple return

Function xyz returns two values a and b:

```
def xyz():  
    return a, b
```

Code calling xyz stores result into one variable assuming xyz returns only one value:

```
t = xyz()
```

Value of t is actually a tuple (a, b) so any action on t assuming it is not a tuple may fail **deep** in the code with a an unexpected **error** about tuples.

```
| TypeError: type tuple doesn't define ... method
```

The fix would be to do:

```
a, b = xyz()
```

Beginners will have trouble finding the reason of this message by only reading the tuple error message !

## Section 200.12: Pythonic JSON keys

```
my_var = 'bla';  
api_key = 'key';  
...lots of code here...  
params = {"language": "en", my_var: api_key}
```

If you are used to JavaScript, variable evaluation in Python dictionaries won't be what you expect it to be. This statement in JavaScript would result in the params object as follows:

```
{  
    "language": "en",  
    "my_var": "key"  
}
```

In Python, however, it would result in the following dictionary:

```
{  
    "language": "en",  
    "bla": "key"
```

}

my\_var 会被求值，其值被用作键。

}

my\_var is evaluated and its value is used as the key.

# 第201章：隐藏特性

## 第201.1节：运算符重载

Python中的一切都是对象。每个对象都有一些特殊的内部方法，用于与其他对象交互。通常，这些方法遵循`_action_`的命名规范。统称为Python数据模型。

您可以重载这些方法中的任何一个。这在Python的运算符重载中非常常见。下面是一个使用Python数据模型进行运算符重载的示例。Vector类创建了一个包含两个变量的简单向量。我们将通过运算符重载为两个向量的数学运算添加相应的支持。

```
class Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, v):
        # 与另一个向量相加。
        return Vector(self.x + v.x, self.y + v.y)

    def __sub__(self, v):
        # 与另一个向量相减。
        return Vector(self.x - v.x, self.y - v.y)

    def __mul__(self, s):
        # 与标量相乘。
        return Vector(self.x * s, self.y * s)

    def __div__(self, s):
        # 与标量的除法。
        float_s = float(s)
        return Vector(self.x / float_s, self.y / float_s)

    def __floordiv__(self, s):
        # 与标量的除法 (向下取整)。
        return Vector(self.x // s, self.y // s)

    def __repr__(self):
        # Vector类的友好打印表示。否则，它会显示为
        # <__main__.Vector instance at 0x01DDDC8> 这样的形式。
        return '<Vector (%f, %f)>' % (self.x, self.y, )

a = Vector(3, 5)
b = Vector(2, 7)

print a + b # 输出: <Vector (5.000000, 12.000000)>
print b - a # 输出: <Vector (-1.000000, 2.000000)>
print b * 1.3 # 输出: <Vector (2.600000, 9.100000)>
print a // 17 # 输出: <Vector (0.000000, 0.000000)>
print a / 17 # 输出: <Vector (0.176471, 0.294118)>
```

上述示例演示了基本数值运算符的重载。完整列表可在 [here](#) 找到。

# Chapter 201: Hidden Features

## Section 201.1: Operator Overloading

Everything in Python is an object. Each object has some special internal methods which it uses to interact with other objects. Generally, these methods follow the `__action__` naming convention. Collectively, this is termed as the [Python Data Model](#).

You can overload *any* of these methods. This is commonly used in operator overloading in Python. Below is an example of operator overloading using Python's data model. The Vector class creates a simple vector of two variables. We'll add appropriate support for mathematical operations of two vectors using operator overloading.

```
class Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, v):
        # Addition with another vector.
        return Vector(self.x + v.x, self.y + v.y)

    def __sub__(self, v):
        # Subtraction with another vector.
        return Vector(self.x - v.x, self.y - v.y)

    def __mul__(self, s):
        # Multiplication with a scalar.
        return Vector(self.x * s, self.y * s)

    def __div__(self, s):
        # Division with a scalar.
        float_s = float(s)
        return Vector(self.x / float_s, self.y / float_s)

    def __floordiv__(self, s):
        # Division with a scalar (value floored).
        return Vector(self.x // s, self.y // s)

    def __repr__(self):
        # Print friendly representation of Vector class. Else, it would
        # show up like, <__main__.Vector instance at 0x01DDDC8>.
        return '<Vector (%f, %f)>' % (self.x, self.y, )

a = Vector(3, 5)
b = Vector(2, 7)

print a + b # Output: <Vector (5.000000, 12.000000)>
print b - a # Output: <Vector (-1.000000, 2.000000)>
print b * 1.3 # Output: <Vector (2.600000, 9.100000)>
print a // 17 # Output: <Vector (0.000000, 0.000000)>
print a / 17 # Output: <Vector (0.176471, 0.294118)>
```

The above example demonstrates overloading of basic numeric operators. A comprehensive list can be found [here](#).

# 致谢

非常感谢所有来自Stack Overflow文档的人员帮助提供此内容，  
更多更改可发送至 [web@petercv.com](mailto:web@petercv.com) 以发布或更新新内容

<a href="#">Çağatay Uslu</a>	第20章
<a href="#">2的三次方</a>	第39、122和147章
<a href="#">4444</a>	第21章
<a href="#">A. 西克莱特</a>	第196章
<a href="#">A. 拉扎</a>	第1章
<a href="#">亚伦·克里斯蒂安森</a>	第40章和第109章
<a href="#">亚伦·克里奇利</a>	第1章
<a href="#">亚伦·霍尔</a>	第38章
<a href="#">阿比谢克·库马尔</a>	第149章
<a href="#">abukaj</a>	第200章
<a href="#">acdr</a>	第21章
<a href="#">亚当·布雷内茨基</a>	第85章
<a href="#">亚当·马坦</a>	第84章和第104章
<a href="#">Adam_92</a>	第40章
<a href="#">adeora</a>	第197章
<a href="#">阿迪亚</a>	第40章, 第53章, 第134章, 第195章和第200章
<a href="#">阿德里安·安图内斯</a>	第88章
<a href="#">阿德里亚诺</a>	第33章
<a href="#">afeique</a>	第1章
<a href="#">艾丹</a>	第75章
<a href="#">阿让</a>	第5章
<a href="#">阿克沙特·马哈詹</a>	第33、67、120和146章
<a href="#">aldanor</a>	第40章
<a href="#">阿尔多</a>	第103章
<a href="#">亚历克</a>	第200章
<a href="#">alecxo</a>	第5、40、87、92和93章
<a href="#">alejosocorro</a>	第1章和第75章
<a href="#">亚历克斯·盖纳</a>	第55章
<a href="#">亚历克斯-L</a>	第16章
<a href="#">亚历克斯·洛根</a>	第1章
<a href="#">AlexV</a>	第33章
<a href="#">Alfe</a>	第88章
<a href="#">alfonso.kim</a>	第16章
<a href="#">ALinuxLover</a>	第1章
<a href="#">阿里雷扎·萨万德</a>	第192章
<a href="#">阿列奥</a>	第21章和第149章
<a href="#">阿隆·亚历山大</a>	第116章
<a href="#">安布利纳</a>	第83章、第85章和第129章
<a href="#">阿米·塔沃里</a>	第192章
<a href="#">阿明</a>	第9章
<a href="#">阿米尔·拉楚姆</a>	第19章和第39章
<a href="#">阿米泰·斯特恩</a>	第41章和第91章
<a href="#">指代</a>	第159章
<a href="#">阿纳托利·泰克托尼克</a>	第160章
<a href="#">安德里亚</a>	第1章
<a href="#">安德鲁</a>	第131章
<a href="#">安德鲁·沙德</a>	第84章

# Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,  
more changes can be sent to [web@petercv.com](mailto:web@petercv.com) for new content to be published or updated

<a href="#">Çağatay Uslu</a>	Chapter 20
<a href="#">2Cubed</a>	Chapters 39, 122 and 147
<a href="#">4444</a>	Chapter 21
<a href="#">A. Ciclet</a>	Chapter 196
<a href="#">A. Raza</a>	Chapter 1
<a href="#">Aaron Christiansen</a>	Chapters 40 and 109
<a href="#">Aaron Critchley</a>	Chapter 1
<a href="#">Aaron Hall</a>	Chapter 38
<a href="#">Abhishek Kumar</a>	Chapter 149
<a href="#">abukaj</a>	Chapter 200
<a href="#">acdr</a>	Chapter 21
<a href="#">Adam Brenecki</a>	Chapter 85
<a href="#">Adam Matan</a>	Chapters 84 and 104
<a href="#">Adam_92</a>	Chapter 40
<a href="#">adeora</a>	Chapter 197
<a href="#">Aditya</a>	Chapters 40, 53, 134, 195 and 200
<a href="#">Adrian Antunez</a>	Chapter 88
<a href="#">Adriano</a>	Chapter 33
<a href="#">afeique</a>	Chapter 1
<a href="#">Aidan</a>	Chapter 75
<a href="#">Ajean</a>	Chapter 5
<a href="#">Akshat Mahajan</a>	Chapters 33, 67, 120 and 146
<a href="#">aldanor</a>	Chapter 40
<a href="#">Aldo</a>	Chapter 103
<a href="#">Alec</a>	Chapter 200
<a href="#">alecxo</a>	Chapters 5, 40, 87, 92 and 93
<a href="#">alejosocorro</a>	Chapters 1 and 75
<a href="#">Alex Gaynor</a>	Chapter 55
<a href="#">Alex L</a>	Chapter 16
<a href="#">Alex Logan</a>	Chapter 1
<a href="#">AlexV</a>	Chapter 33
<a href="#">Alfe</a>	Chapter 88
<a href="#">alfonso.kim</a>	Chapter 16
<a href="#">ALinuxLover</a>	Chapter 1
<a href="#">Alireza Savand</a>	Chapter 192
<a href="#">Alleo</a>	Chapters 21 and 149
<a href="#">Alon Alexander</a>	Chapter 116
<a href="#">amblina</a>	Chapters 83, 85 and 129
<a href="#">Ami Tavoray</a>	Chapter 192
<a href="#">amin</a>	Chapter 9
<a href="#">Amir Rachum</a>	Chapters 19 and 39
<a href="#">Amitay Stern</a>	Chapters 41 and 91
<a href="#">Anaphory</a>	Chapter 159
<a href="#">anatoly techtonik</a>	Chapter 160
<a href="#">Andrea</a>	Chapter 1
<a href="#">andrew</a>	Chapter 131
<a href="#">Andrew Schade</a>	Chapter 84

安德烈·阿布拉莫夫  
安杰伊·普罗诺比斯  
安迪  
安迪·海登  
angussidney  
安妮·梅农  
匿名  
安东尼·范  
安托万·博尔维  
安托万·皮纳尔  
安蒂·哈帕拉  
安特万  
一个人  
阿奎布·贾维德·汗  
阿瑞斯  
阿尔卡迪  
阿尔皮特·索兰基  
阿尔捷姆·科隆泰  
ArtOfCode  
阿伦  
阿里曼·阿罗拉  
ashes999  
asmeurer  
atayenel  
Athari  
Avantol13  
avb  
Axe  
B8vrede  
咩咩牛  
巴赫罗姆  
巴库里乌  
巴尔基  
巴里  
巴斯蒂安  
bbayles  
Beall619  
bee  
本尼迪克特·邦廷  
Bharel  
巴尔加夫  
巴尔加夫·拉奥  
大鼻子  
比利  
Biswa\_9937  
bitchaser  
bixel  
blubberdiblub  
blueberryfields  
blueenvelope  
Bluethon  
boboquack  
bogdanciobanu

第1章  
第8、38和55章  
第1、5、7、17、33、51、82和88章  
第8、33、67、73、75、77、98和149章  
第1章  
第1、4、40、149和154章  
第78、87和199章  
第13、15、16、19、20、30、33、43、45、55、60、70和179章  
第1和149章  
第39章  
第5章、第16章、第72章、第87章和第106章  
第149章  
第21章、第69章和第72章  
第1章  
第1章、第15章、第20章和第41章  
第33章  
第1章、第82章和第125章  
第173章  
第67章、第173章和第197章  
第65章和第108章  
第179章  
第75章  
第45章和第47章  
第124章  
第151章  
第38章  
第30章  
第21章和第149章  
第1章、第40章、第75章和第103章  
第1章和第200章  
第8章  
第149章  
第53章  
第20章  
第86章  
第128章  
第74章和第101章  
第161章和第164章  
第99章  
第30章和第149章  
第40章  
第41章、第149章和第200章  
第149章  
第200章  
第178章、第182章和第183章  
第149章  
第200章  
第177章  
第181章  
第9章和第20章  
第149章  
第10章、第11章和第18章  
第111章

Andrii Abramov  
Andrzej Pronobis  
Andy  
Andy Hayden  
angussidney  
Ani Menon  
Anonymous  
Anthony Pham  
Antoine Bolvy  
Antoine Pinsard  
Antti Haapala  
Antwan  
APerson  
Aquia Javed Khan  
Ares  
Arkady  
Arpit Solanki  
Artem Kolontay  
ArtOfCode  
Arun  
Aryaman Arora  
ashes999  
asmeurer  
atayenel  
Athari  
Avantol13  
avb  
Axe  
B8vrede  
Baaing Cow  
Bahrom  
Bakuriu  
baliki  
Barry  
Bastian  
bbayles  
Beall619  
bee  
Benedict Bunting  
Bharel  
Bhargav  
Bhargav Rao  
bignose  
Billy  
Biswa\_9937  
bitchaser  
bixel  
blubberdiblub  
blueberryfields  
blueenvelope  
Bluethon  
boboquack  
bogdanciobanu

Chapter 1  
Chapters 8, 38 and 55  
Chapters 1, 5, 7, 17, 33, 51, 82 and 88  
Chapters 8, 33, 67, 73, 75, 77, 98 and 149  
Chapter 1  
Chapters 1, 4, 40, 149 and 154  
Chapters 78, 87 and 199  
Chapters 13, 15, 16, 19, 20, 30, 33, 43, 45, 55, 60, 70 and 179  
Chapters 1 and 149  
Chapter 39  
Chapters 5, 16, 72, 87 and 106  
Chapter 149  
Chapters 21, 69 and 72  
Chapter 1  
Chapters 1, 15, 20 and 41  
Chapter 33  
Chapters 1, 82 and 125  
Chapter 173  
Chapters 67, 173 and 197  
Chapters 65 and 108  
Chapter 179  
Chapter 75  
Chapters 45 and 47  
Chapter 124  
Chapter 151  
Chapter 38  
Chapter 30  
Chapters 21 and 149  
Chapters 1, 40, 75 and 103  
Chapters 1 and 200  
Chapter 8  
Chapter 149  
Chapter 53  
Chapter 20  
Chapter 86  
Chapter 128  
Chapters 74 and 101  
Chapters 161 and 164  
Chapter 99  
Chapters 30 and 149  
Chapter 40  
Chapters 41, 149 and 200  
Chapter 149  
Chapter 149  
Chapter 200  
Chapters 178, 182 and 183  
Chapter 149  
Chapter 200  
Chapter 177  
Chapter 181  
Chapters 9 and 20  
Chapter 149  
Chapters 10, 11 and 18  
Chapter 111

博尼法西奥2  
博普雷H  
博索内安多  
博žo 斯托伊科维ć  
bpachev  
布伦丹·阿贝尔  
布伦南  
布雷特·坎农  
布赖恩·C  
布里恩  
布莱恩·P  
BSL  
布尔汗·哈立德  
BusyAnt  
Buzz  
Cache Staheli  
CamelBackNotation  
卡梅隆·加农  
Camsbury  
caped114  
carrdelling  
Cbeb24404  
ceruleus  
cfi  
钱丹·普罗希特  
ChaoticTwist  
查尔斯  
查鲁尔  
钦迈·赫格德  
唐崇  
克里斯·亨特  
克里斯·拉尔森  
克里斯·米德格利  
克里斯·穆勒  
克里斯蒂安·特努斯  
克里斯托弗·奥尔松  
克里斯托夫·鲁西  
铬  
西利安  
钦巴利  
cizixs  
cjs  
克利奥娜  
克劳迪乌  
克莱顿·沃尔斯特罗姆  
cledoux  
代号Lambda  
科迪·皮尔索尔  
科林·杨  
同志 SparklePony  
康拉德·迪恩  
crhodes  
C\_L\_S

第64章  
第19章  
第46章  
第1、14、21、33和149章  
第53章  
第84章  
第173章  
第11章和第43章  
第1章  
第41章  
第1章  
第1章和第197章  
第19章  
第1、20、43、60、78和88章  
第18章  
第41章  
第33章  
第149章  
第9和33章  
第41章  
第77章  
第1章  
第1章  
第21章、第26章和第69章  
第20章和第33章  
第21章、第33章、第41章和第107章  
第10章、第21章、第30章、第41章、第149章和第200章  
第167章  
第50、94、132、164和171章  
第21章  
第16章  
第33章  
第1章  
第19章  
第1章、第16章、第43章、第51章和第78章  
第45章  
第200章  
第134章  
第170章  
第8章  
第19、20和128章  
第110章和第134章  
第1章  
第1章、第31章、第67章、第75章、第80章、第83章、第88章、第111章、第118章、第153章、第192章和第193章  
第149章  
第108章  
第1章和第67章  
第8章  
第149章  
第180章和第181章  
第1章、第5章、第21章、第38章和第43章  
第30章  
第1章、第149章、第161章和第191章

Bonifacio2  
BoppreH  
Bosoneando  
Božo Stojković  
bpachev  
Brendan Abel  
brennan  
Brett Cannon  
Brian C  
Brien  
Bryan P  
BSL  
Burhan Khalid  
BusyAnt  
Buzz  
Cache Staheli  
CamelBackNotation  
Cameron Gagnon  
Camsbury  
caped114  
carrdelling  
Cbeb24404  
ceruleus  
cfi  
Chandan Purohit  
ChaoticTwist  
Charles  
Charul  
Chinmay Hegde  
Chong Tang  
Chris Hunt  
Chris Larson  
Chris Midgley  
Chris Mueller  
Christian Ternus  
Christofer Ohlsson  
Christophe Roussy  
Chromium  
Cilyan  
Cimbali  
cizixs  
cjs  
Clíodhna  
Claudiu  
Clayton Wahlstrom  
cledoux  
CodenameLambda  
Cody Piersall  
Colin Yang  
Comrade SparklePony  
Conrad.Dean  
crhodes  
C\_L\_S

Chapter 64  
Chapter 19  
Chapter 46  
Chapters 1, 14, 21, 33 and 149  
Chapter 53  
Chapter 84  
Chapter 173  
Chapters 11 and 43  
Chapter 1  
Chapter 41  
Chapter 1  
Chapters 1 and 197  
Chapter 19  
Chapters 1, 20, 43, 60, 78 and 88  
Chapter 18  
Chapter 41  
Chapter 33  
Chapter 149  
Chapters 9 and 33  
Chapter 41  
Chapter 77  
Chapter 1  
Chapter 1  
Chapters 21, 26 and 69  
Chapters 20 and 33  
Chapters 21, 33, 41 and 107  
Chapters 10, 21, 30, 41, 149 and 200  
Chapter 167  
Chapters 50, 94, 132, 164 and 171  
Chapter 21  
Chapter 16  
Chapter 33  
Chapter 1  
Chapter 19  
Chapters 1, 16, 43, 51 and 78  
Chapter 45  
Chapter 200  
Chapter 134  
Chapter 170  
Chapter 8  
Chapters 19, 20 and 128  
Chapters 110 and 134  
Chapter 1  
Chapters 1, 31, 67, 75, 80, 83, 88, 111, 118, 153, 192 and 193  
Chapter 149  
Chapter 108  
Chapters 1 and 67  
Chapter 8  
Chapter 149  
Chapters 180 and 181  
Chapters 1, 5, 21, 38 and 43  
Chapter 30  
Chapters 1, 149, 161 and 191

达伊尔  
达克什·古普塔  
达尼娅  
danidee  
丹尼尔  
达尼尔·雷日科夫  
达卡德  
达斯·科蒂克  
达斯·影子  
达特茅斯  
戴夫J  
大卫  
大卫·卡伦  
大卫·海曼  
davidism  
黎明圣骑士  
迪  
deenes  
deepakkt  
深空  
德尔甘  
登瓦尔  
德珀姆  
德夫D  
德维什·赛尼  
迪亚TN  
迪曼塔  
业余爱好者  
迪玛·蒂斯内克  
贾什楚罗夫斯基  
文档  
多德尔  
哆啦A梦  
道格·亨德森  
道格拉斯·斯塔恩斯  
多夫  
dreftymac  
driax  
呃  
杜纳托塔托斯  
dwanderson  
eandersson  
edwinksl  
eenblam  
以拉扎尔  
Eleftheria  
legant  
埃利斯  
埃洛丁  
艾玛  
埃纳穆尔·哈桑  
engineerencoding  
恩里科·玛丽亚·德·安吉利斯

第11章、第120章和第173章  
第1章和第38章  
第1章  
第33章  
第43章  
第173章  
第173章  
第16章  
第1、40、75、97、149和173章  
第1、62、149和150章  
第40和149章  
第9、33、124和161章  
第30和121章  
第41章和第149章  
第13章  
第33章  
第186章  
第1章、第20章和第67章  
第14章  
第9章、第16章、第77章、第100章、第149章和第200章  
第1章、第16章、第20章和第40章  
第167章  
第1章、第3章、第38章、第41章和第99章  
第1章  
第115章  
第16章  
第184章和第185章  
第200章  
第21章  
第167章  
第143章  
第1章  
第29章  
第41章  
第1章  
第30章  
第40章  
第39章、第75章和第88章  
第149章  
第171章  
第149章  
第127章  
第173章  
第21章  
第1、7、8、13、15、16、20、21、28、38、41、67、87、111、144和149章  
第113章  
第33章  
第20章和第45章  
第33章和第195章  
第20章和第21章  
第16章  
第192章  
第1章

Dair  
Daksh Gupta  
Dania  
danidee  
Daniel  
Daniil Ryzhkov  
Darkade  
Darth Kotik  
Darth Shadow  
Dartmouth  
Dave J  
David  
David Cullen  
David Heyman  
davidism  
DawnPaladin  
Dee  
deenes  
deepakkt  
DeepSpace  
Delgan  
denvaar  
depperm  
DevD  
Devesh Saini  
DhiaTN  
dhimanta  
Dilettant  
Dima Tisnek  
djaszczurowski  
Doc  
dodell  
Doraemon  
Doug Henderson  
Douglas Starnes  
Dov  
dreftymac  
driax  
Duh  
Dunatotatos  
dwanderson  
eandersson  
edwinksl  
eenblam  
Elazar  
Eleftheria  
legant  
Ellis  
Elodin  
Emma  
Enamul Hassan  
engineerencoding  
Enrico Maria De Angelis

Chapters 11, 120 and 173  
Chapters 1 and 38  
Chapter 1  
Chapter 33  
Chapter 43  
Chapter 173  
Chapter 173  
Chapter 16  
Chapters 1, 40, 75, 97, 149 and 173  
Chapters 1, 62, 149 and 150  
Chapters 40 and 149  
Chapters 9, 33, 124 and 161  
Chapters 30 and 121  
Chapters 41 and 149  
Chapter 13  
Chapter 33  
Chapter 186  
Chapters 1, 20 and 67  
Chapter 14  
Chapters 9, 16, 77, 100, 149 and 200  
Chapters 1, 16, 20 and 40  
Chapter 167  
Chapters 1, 3, 38, 41 and 99  
Chapter 1  
Chapter 115  
Chapter 16  
Chapters 184 and 185  
Chapter 200  
Chapter 21  
Chapter 167  
Chapter 143  
Chapter 1  
Chapter 29  
Chapter 41  
Chapter 1  
Chapter 30  
Chapter 40  
Chapters 39, 75 and 88  
Chapter 149  
Chapter 171  
Chapter 149  
Chapter 127  
Chapter 173  
Chapter 21  
Chapters 1, 7, 8, 13, 15, 16, 20, 21, 28, 38, 41, 67, 87, 111, 144 and 149  
Chapter 113  
Chapter 33  
Chapters 20 and 45  
Chapters 33 and 195  
Chapters 20 and 21  
Chapter 16  
Chapter 192  
Chapter 1

<a href="#">enrico.bacis</a>	第21章、第56章和第149章	<a href="#">enrico.bacis</a>	Chapters 21, 56 and 149
<a href="#">erewok</a>	第149章	<a href="#">erewok</a>	Chapter 149
<a href="#">埃里克</a>	第107章	<a href="#">Eric</a>	Chapter 107
<a href="#">埃里克·芬恩</a>	第16章	<a href="#">Eric Finn</a>	Chapter 16
<a href="#">埃里克·张</a>	第110章	<a href="#">Eric Zhang</a>	Chapter 110
<a href="#">埃丽卡</a>	第1章	<a href="#">Erica</a>	Chapter 1
<a href="#">ericdwang</a>	第149章	<a href="#">ericdwang</a>	Chapter 149
<a href="#">ericmarkmartin</a>	第67、98、110和149章	<a href="#">ericmarkmartin</a>	Chapters 67, 98, 110 and 149
<a href="#">埃里克·戈达德</a>	第1章	<a href="#">Erik Godard</a>	Chapter 1
<a href="#">EsmaeelE</a>	第1章	<a href="#">EsmaeelE</a>	Chapter 1
<a href="#">Esteis</a>	第13章和第21章	<a href="#">Esteis</a>	Chapters 13 and 21
<a href="#">ettanany</a>	第27章和第149章	<a href="#">ettanany</a>	Chapters 27 and 149
<a href="#">Everyone_Else</a>	第149章	<a href="#">Everyone_Else</a>	Chapter 149
<a href="#">evuez</a>	第7、8、14、20、40、113和149章	<a href="#">evuez</a>	Chapters 7, 8, 14, 20, 40, 113 and 149
<a href="#">ex huma</a>	第20章	<a href="#">ex huma</a>	Chapter 20
<a href="#">法比奥·佩雷斯</a>	第31章	<a href="#">Fábio Perez</a>	Chapter 31
<a href="#">法伊兹·哈尔德</a>	第21章、第114章和第119章	<a href="#">Faiz Halde</a>	Chapters 21, 114 and 119
<a href="#">法泽尔</a>	第111章和第148章	<a href="#">FazeL</a>	Chapters 111 and 148
<a href="#">费利克斯·D.</a>	第16章	<a href="#">Felix D.</a>	Chapter 16
<a href="#">费尔克</a>	第21章	<a href="#">Felk</a>	Chapter 21
<a href="#">费米悖论</a>	第21章	<a href="#">Fermi paradox</a>	Chapter 21
<a href="#">费尔南多</a>	第173章	<a href="#">Fernando</a>	Chapter 173
<a href="#">Ffisegydd</a>	第14章、第38章、第98章、第108章和第193章	<a href="#">Ffisegydd</a>	Chapters 14, 38, 98, 108 and 193
<a href="#">菲利普·哈格伦</a>	第1章	<a href="#">Filip Haglund</a>	Chapter 1
<a href="#">Firix</a>	第1章和第150章	<a href="#">Firix</a>	Chapters 1 and 150
<a href="#">弗拉门戈</a>	第56章	<a href="#">flamenco</a>	Chapter 56
<a href="#">闪烁之光</a>	第20章	<a href="#">Flickerlight</a>	Chapter 20
<a href="#">弗洛里安·本德</a>	第21章	<a href="#">Florian Bender</a>	Chapter 21
<a href="#">FMc</a>	第43章	<a href="#">FMc</a>	Chapter 43
<a href="#">弗朗西斯科·吉马良斯</a>	第94章	<a href="#">Francisco Guimaraes</a>	Chapter 94
<a href="#">弗朗克·德尔农库尔</a>	第1章	<a href="#">Franck Dernoncourt</a>	Chapter 1
<a href="#">FrankBr</a>	第36章	<a href="#">FrankBr</a>	Chapter 36
<a href="#">frankyjuang</a>	第181章	<a href="#">frankyjuang</a>	Chapter 181
<a href="#">弗雷德·巴克莱</a>	第1章和第157章	<a href="#">Fred Barclay</a>	Chapters 1 and 157
<a href="#">弗雷迪</a>	第1章	<a href="#">Freddy</a>	Chapter 1
<a href="#">fredley</a>	第45章	<a href="#">fredley</a>	Chapter 45
<a href="#">freidrichen</a>	第21章	<a href="#">freidrichen</a>	Chapter 21
<a href="#">沮丧的</a>	第74章	<a href="#">Frustrated</a>	Chapter 74
<a href="#">加尔·德雷曼</a>	第16、20、21、33、40、136和137章	<a href="#">Gal Dreiman</a>	Chapters 16, 20, 21, 33, 40, 136 and 137
<a href="#">加内什·加迪拉</a>	第20和41章	<a href="#">ganesh gadila</a>	Chapters 20 and 41
<a href="#">加内什·K</a>	第148章	<a href="#">Ganesh K</a>	Chapter 148
<a href="#">加雷斯·拉蒂</a>	第19章	<a href="#">Gareth Latty</a>	Chapter 19
<a href="#">garg10may</a>	第21章和第149章	<a href="#">garg10may</a>	Chapters 21 and 149
<a href="#">加文</a>	第149章	<a href="#">Gavin</a>	Chapter 149
<a href="#">Geekhem</a>	第14章、第110章和第124章	<a href="#">Geekhem</a>	Chapters 14, 110 and 124
<a href="#">通用蛇</a>	第16章	<a href="#">Generic Snake</a>	Chapter 16
<a href="#">geoffspear</a>	第149章	<a href="#">geoffspear</a>	Chapter 149
<a href="#">杰拉德·罗什</a>	第1章	<a href="#">Gerard Roche</a>	Chapter 1
<a href="#">gerrit</a>	第40章	<a href="#">gerrit</a>	Chapter 40
<a href="#">ghostarbeiter</a>	第16章、第21章、第33章、第41章、第45章、第88章、第132章和第149章	<a href="#">ghostarbeiter</a>	Chapters 16, 21, 33, 41, 45, 88, 132 and 149
<a href="#">扬尼斯·斯皮利奥普洛斯</a>	第40章	<a href="#">Giannis Spiliopoulos</a>	Chapter 40
<a href="#">GiantsLoveDeathMetal</a>	第40章和第164章	<a href="#">GiantsLoveDeathMetal</a>	Chapters 40 and 164
<a href="#">girish946</a>	第31章和第154章	<a href="#">girish946</a>	Chapters 31 and 154

<a href="#">giucal</a>	第55章	<a href="#">giucal</a>	Chapter 55
<a href="#">GoatsWearHats</a>	第1章、第16章、第29章、第41章和第72章	<a href="#">GoatsWearHats</a>	Chapters 1, 16, 29, 41 and 72
<a href="#">goodmami</a>	第75章	<a href="#">goodmami</a>	Chapter 75
<a href="#">格雷格</a>	第22章	<a href="#">Greg</a>	Chapter 22
<a href="#">greut</a>	第37章	<a href="#">greut</a>	Chapter 37
<a href="#">盖伊</a>	第16章、第19章和第156章	<a href="#">Guy</a>	Chapters 16, 19 and 156
<a href="#">H. 保维伦</a>	第1章	<a href="#">H. Pauwelyn</a>	Chapter 1
<a href="#">hackvan</a>	第164章	<a href="#">hackvan</a>	Chapter 164
<a href="#">汉内莱</a>	第21章	<a href="#">Hannele</a>	Chapter 21
<a href="#">汉内斯·卡尔皮拉</a>	第14章和第134章	<a href="#">Hannes Karppila</a>	Chapters 14 and 134
<a href="#">哈里森</a>	第40章	<a href="#">Harrison</a>	Chapter 40
<a href="#">hashcode55</a>	第76章	<a href="#">hashcode55</a>	Chapter 76
<a href="#">ha_1694</a>	第173章	<a href="#">ha_1694</a>	Chapter 173
<a href="#">heyhey2k</a>	第94章	<a href="#">heyhey2k</a>	Chapter 94
<a href="#">广主角</a>	第54章和第200章	<a href="#">hiro protagonist</a>	Chapters 54 and 200
<a href="#">悬停地狱</a>	第12章	<a href="#">HoverHell</a>	Chapter 12
<a href="#">赫里迪·德伊</a>	第105章	<a href="#">Hriddhi Dey</a>	Chapter 105
<a href="#">赫尔基尔</a>	第21章和第33章	<a href="#">Hurkyl</a>	Chapters 21 and 33
<a href="#">hxysayhi</a>	第66章和第168章	<a href="#">hxysayhi</a>	Chapters 66 and 168
<a href="#">伊恩</a>	第1章	<a href="#">ian</a>	Chapter 1
<a href="#">伊恩·奥尔德</a>	第1章和第21章	<a href="#">ianAuld</a>	Chapters 1 and 21
<a href="#">iankit</a>	第21章和第37章	<a href="#">iankit</a>	Chapters 21 and 37
<a href="#">我相信</a>	第19章	<a href="#">iBelieve</a>	Chapter 19
<a href="#">idjaw</a>	第41章	<a href="#">idjaw</a>	Chapter 41
<a href="#">ifma</a>	第108章	<a href="#">ifma</a>	Chapter 108
<a href="#">伊戈尔·劳什</a>	第1、19、20、38、39、45、67和98章	<a href="#">Igor Raush</a>	Chapters 1, 19, 20, 38, 39, 45, 67 and 98
<a href="#">伊利亚·巴拉霍夫斯基</a>	第32、41和88章	<a href="#">Ilia Barahovski</a>	Chapters 32, 41 and 88
<a href="#">ilse2005</a>	第30章	<a href="#">ilse2005</a>	Chapter 30
<a href="#">因巴尔·罗斯</a>	第16章	<a href="#">Inbar Rose</a>	Chapter 16
<a href="#">因德拉达努什·古普塔</a>	第49章	<a href="#">Indradhanush Gupta</a>	Chapter 49
<a href="#">无限</a>	第19和115章	<a href="#">Infinity</a>	Chapters 19 and 115
<a href="#">InitializeSahib</a>	第38、39和82章	<a href="#">InitializeSahib</a>	Chapters 38, 39 and 82
<a href="#">intboolstring</a>	第10、21和45章	<a href="#">intboolstring</a>	Chapters 10, 21 and 45
<a href="#">iScrE4m</a>	第149章	<a href="#">iScrE4m</a>	Chapter 149
<a href="#">JF</a>	第1、9、15、20、29、33、38、88、97、111、119、125和149章	<a href="#">JF</a>	Chapters 1, 9, 15, 20, 29, 33, 38, 88, 97, 111, 119, 125 and 149
<a href="#">约恩·赫斯</a>	第137章	<a href="#">Jörn Hees</a>	Chapter 137
<a href="#">JOHN</a>	第21和67章	<a href="#">JOHN</a>	Chapters 21 and 67
<a href="#">j3485</a>	第20章	<a href="#">j3485</a>	Chapter 20
<a href="#">jackskis</a>	第53章和第67章	<a href="#">jackskis</a>	Chapters 53 and 67
<a href="#">雅克·德·霍格</a>	第97章和第151章	<a href="#">Jacques de Hooge</a>	Chapters 97 and 151
<a href="#">JakeD</a>	第22章	<a href="#">JakeD</a>	Chapter 22
<a href="#">詹姆斯</a>	第8章、第14章、第19章、第20章、第33章和第100章	<a href="#">James</a>	Chapters 8, 14, 19, 20, 33 and 100
<a href="#">詹姆斯·埃尔德菲尔德</a>	第20章、第40章、第45章和第149章	<a href="#">James Elderfield</a>	Chapters 20, 40, 45 and 149
<a href="#">詹姆斯·泰勒</a>	第1章	<a href="#">James Taylor</a>	Chapter 1
<a href="#">JamesS</a>	第21章	<a href="#">JamesS</a>	Chapter 21
<a href="#">Jan</a>	第75章	<a href="#">Jan</a>	Chapter 75
<a href="#">jani</a>	第20章	<a href="#">jani</a>	Chapter 20
<a href="#">japborst</a>	第86章	<a href="#">japborst</a>	Chapter 86
<a href="#">Jean</a>	第1章和第40章	<a href="#">Jean</a>	Chapters 1 and 40
<a href="#">jedwards</a>	第1章、第102章和第149章	<a href="#">jedwards</a>	Chapters 1, 102 and 149
<a href="#">杰夫·哈钦斯</a>	第110章	<a href="#">Jeff Hutchins</a>	Chapter 110
<a href="#">杰弗里·林</a>	第1章、第16章、第75章、第132章和第200章	<a href="#">Jeffrey Lin</a>	Chapters 1, 16, 75, 132 and 200
<a href="#">耶尔默·S</a>	第102章	<a href="#">JelmerS</a>	Chapter 102

[J·格林韦尔](#)

[JHS](#)

[吉姆·法萨拉基斯·希利亚德](#)

[吉姆·奥普莱杜尔文](#)

[吉米·宋](#)

[jimsug](#)

[jkdev](#)

[jkitchen](#)

[JL·佩雷特](#)

[jlarsch](#)

[jmunsch](#)

[JNat](#)

[joel3000](#)

[约翰·伦德伯格](#)

[约翰·斯莱格斯](#)

[约翰·兹温克](#)

[乔纳坦](#)

[jonrharpe](#)

[约瑟夫·特鲁](#)

[乔仕](#)

[乔西·卡尔德隆](#)

[jrast](#)

[JRodDynamite](#)

[jtbandes](#)

[胡安·T](#)

[胡安·巴勃罗](#)

[朱利安·斯普朗克](#)

[尤利·叶戈罗夫](#)

[贾斯汀](#)

[贾斯汀·查德韦尔](#)

[贾斯汀·M·乌卡尔](#)

[J](#)

[卡比](#)

[卡尔兹](#)

[卡姆兰·麦基](#)

[卡尔·克内赫特尔](#)

[卡蒂克·卡纳普尔](#)

[kdopen](#)

[keiv.fly](#)

[凯文·布朗](#)

[KeyWeeUsr](#)

[KIDJourney](#)

[Kinifwyne](#)

[基兰·韦穆里](#)

[kisanme](#)

[knight](#)

[kollery](#)

[kon psych](#)

[krato](#)

[库纳尔·马尔瓦哈](#)

[Kwarrtz](#)

[L3viathan](#)

[拉菲克斯洛斯](#)

第3、9、33和37章

第21章

第1、16、33、41、55、87、110、149和200章

第1章

第149章

第1章和第20章

第20章和第38章

第33章

第40章和第51章

第38章

第1章、第47章、第92章、第125章、第181章和第192章

第10章、第20章和第29章

第21章、第103章和第192章

第1章

第1和149章

第11章

第40章、第64章和第87章

第1章、第75章、第78章和第98章

第1章

第149章

第86章

第16章

第1章、第21章和第40章

第70章

第1章、第67章、第90章和第149章

第110章

第53章和第75章

第188章

第30章、第33章、第40章、第74章和第149章

第125章

第149章

第41章

第149章

第38章

第1章

第67、69和149章

第20章和第38章

第19、21和110章

第194章

第1、5、14、30、53、75、146、149和192章

第80和153章

第21章

第43和193章

第1章

第1章

第40章

第156章

第47章

第83章

第149章

第21章

第33章和第98章

第1章、第9章和第20章

[JGreenwell](#)

[JHS](#)

[Jim Fasarakis Hilliard](#)

[jim opleydulven](#)

[Jimmy Song](#)

[jimsug](#)

[jkdev](#)

[jkitchen](#)

[JL Peyret](#)

[jlarsch](#)

[jmunsch](#)

[JNat](#)

[joel3000](#)

[Johan Lundberg](#)

[John Slegers](#)

[John Zwinck](#)

[Jonatan](#)

[jonrharpe](#)

[Joseph True](#)

[Josh](#)

[Jossie Calderon](#)

[jrast](#)

[JRodDynamite](#)

[jtbandes](#)

[Juan T](#)

[JuanPablo](#)

[Julien Spronck](#)

[Julij Jegorov](#)

[Justin](#)

[Justin Chadwell](#)

[Justin M. Ucar](#)

[J](#)

[Kabie](#)

[Kallz](#)

[Kamran Mackey](#)

[Karl Knechtel](#)

[KartikKannapur](#)

[kdopen](#)

[keiv.fly](#)

[Kevin Brown](#)

[KeyWeeUsr](#)

[KIDJourney](#)

[Kinifwyne](#)

[Kiran Vemuri](#)

[kisanme](#)

[knight](#)

[kollery](#)

[kon psych](#)

[krato](#)

[Kunal Marwaha](#)

[Kwarrtz](#)

[L3viathan](#)

[Lafexlos](#)

Chapters 3, 9, 33 and 37

Chapter 21

Chapters 1, 16, 33, 41, 55, 87, 110, 149 and 200

Chapter 1

Chapter 149

Chapters 1 and 20

Chapters 20 and 38

Chapter 33

Chapters 40 and 51

Chapter 38

Chapters 1, 47, 92, 125, 181 and 192

Chapters 10, 20 and 29

Chapters 21, 103 and 192

Chapter 1

Chapters 1 and 149

Chapter 11

Chapters 40, 64 and 87

Chapters 1, 75, 78 and 98

Chapter 1

Chapter 149

Chapter 86

Chapter 16

Chapters 1, 21 and 40

Chapter 70

Chapters 1, 67, 90 and 149

Chapter 110

Chapters 53 and 75

Chapter 188

Chapters 30, 33, 40, 74 and 149

[LDP](#)  
[李·内瑟顿](#)  
[利奥](#)  
[利奥·图玛](#)  
[莱昂](#)  
[莱昂·Z.](#)  
[Liteye](#)  
[加载中...](#)  
[Locane](#)  
[lorenzofeliz](#)  
[LostAvatar](#)  
[卢卡·范·奥特](#)  
[卢克·泰勒](#)  
[lukewrites](#)  
[林赛·西蒙](#)  
[机器渴望](#)  
[magu](#)  
[马赫迪](#)  
[马哈茂德·哈希米](#)  
[马吉德](#)  
[麦芽](#)  
[manu](#)  
[MANU](#)  
[马尔科·帕什科夫](#)  
[马里奥·科尔切罗](#)  
[马克](#)  
[马克·米勒](#)  
[马克·奥莫](#)  
[马库斯·梅斯卡嫩](#)  
[马克·派森](#)  
[马龙·阿贝库恩](#)  
[马丁·皮特斯](#)  
[马丁·瓦尔古尔](#)  
[数学](#)  
[马蒂亚斯711](#)  
[马茨乔伊斯](#)  
[马特·多奇](#)  
[马特·吉尔塔吉](#)  
[马特·罗兰](#)  
[马特科尔](#)  
[马修·惠特](#)  
[mattgathu](#)  
[马修](#)  
[最大值](#)  
[冯马克斯](#)  
[mbrig](#)  
[mbsingh](#)  
[穆罕默德·西法图尔·伊斯兰](#)  
[mdegis](#)  
[机械师](#)  
[mertyildiran](#)  
[MervS](#)  
[metmirr](#)

第20章  
第21章、第33章和第114章  
第49章和第97章  
第20章  
第1章  
第74章  
第21章和第38章  
第83章和第114章  
第21章  
第2章  
第1章和第161章  
第165章  
第70章  
第20章  
第21章  
第19、53和67章  
第77章  
第21、82和120章  
第199章  
第19章、第28章、第77章、第82章、第98章、第112章和第173章  
第200章  
第1章  
第1章  
第29章、第39章、第40章、第83章和第173章  
第192章  
第125章  
第130章  
第168章  
第21章  
第41章  
第79章  
第1、67、77和97章  
第104章  
第16章  
第1章  
第1章和第9章  
第149章和第200章  
第33章、第40章、第173章和第193章  
第149章  
第4章和第121章  
第21章、第37章、第39章、第110章、第146章、第173章和第192章  
第19章、第117章、第124章和第190章  
第77章  
第67章  
第14章  
第21章  
第161章  
第28章和第75章  
第1章  
第1章、第9章、第19章和第28章  
第1章  
第125章  
第162章

[LDP](#)  
[Lee Netherton](#)  
[Leo](#)  
[Leo Thumma](#)  
[Leon](#)  
[Leon Z.](#)  
[Liteye](#)  
[loading...](#)  
[Locane](#)  
[lorenzofeliz](#)  
[LostAvatar](#)  
[Luca Van Oort](#)  
[Luke Taylor](#)  
[lukewrites](#)  
[Lyndsy Simon](#)  
[machine yearning](#)  
[magu](#)  
[Mahdi](#)  
[Mahmoud Hashemi](#)  
[Majid](#)  
[Malt](#)  
[manu](#)  
[MANU](#)  
[Marco Pashkov](#)  
[Mario Corchero](#)  
[Mark](#)  
[Mark Miller](#)  
[Mark Omo](#)  
[Markus Meskanen](#)  
[MarkyPython](#)  
[Marlon Abeykoon](#)  
[Martijn Pieters](#)  
[Martin Valgur](#)  
[Math](#)  
[Mathias711](#)  
[matsjoyce](#)  
[Matt Dodge](#)  
[Matt Giltaji](#)  
[Matt Rowland](#)  
[MattCorr](#)  
[Mattev Whitt](#)  
[mattgathu](#)  
[Matthew](#)  
[max](#)  
[Max Feng](#)  
[mbrig](#)  
[mbsingh](#)  
[Md Sifatul Islam](#)  
[mdegis](#)  
[Mechanic](#)  
[mertyildiran](#)  
[MervS](#)  
[metmirr](#)

Chapter 20  
Chapters 21, 33 and 114  
Chapters 49 and 97  
Chapter 20  
Chapter 1  
Chapter 74  
Chapters 21 and 38  
Chapters 83 and 114  
Chapter 21  
Chapter 2  
Chapters 1 and 161  
Chapter 165  
Chapter 70  
Chapter 20  
Chapter 21  
Chapters 19, 53 and 67  
Chapter 77  
Chapters 21, 82 and 120  
Chapter 199  
Chapters 19, 28, 77, 82, 98, 112 and 173  
Chapter 200  
Chapter 1  
Chapter 1  
Chapters 29, 39, 40, 83 and 173  
Chapter 192  
Chapter 125  
Chapter 130  
Chapter 168  
Chapter 21  
Chapter 41  
Chapter 79  
Chapters 1, 67, 77 and 97  
Chapter 104  
Chapter 16  
Chapter 1  
Chapters 1 and 9  
Chapters 149 and 200  
Chapters 33, 40, 173 and 193  
Chapter 149  
Chapters 4 and 121  
Chapters 21, 37, 39, 110, 146, 173 and 192  
Chapters 19, 117, 124 and 190  
Chapter 77  
Chapter 67  
Chapter 14  
Chapter 21  
Chapter 161  
Chapters 28 and 75  
Chapter 1  
Chapters 1, 9, 19 and 28  
Chapter 1  
Chapter 125  
Chapter 162

<a href="#">mezzode</a>	第28章	<a href="#">mezzode</a>	Chapter 28
<a href="#">mgilson</a>	第192章	<a href="#">mgilson</a>	Chapter 192
<a href="#">迈克尔·雷卡奇纳斯</a>	第149章	<a href="#">Michael Recachinas</a>	Chapter 149
<a href="#">米歇尔·托乌</a>	第161章	<a href="#">Michel Touw</a>	Chapter 161
<a href="#">迈克·德里斯科尔</a>	第1章和第13章	<a href="#">Mike Driscoll</a>	Chapters 1 and 13
<a href="#">米尔廷·米基奇</a>	第1章	<a href="#">Miljen Mikic</a>	Chapter 1
<a href="#">米穆尼</a>	第1章	<a href="#">Mimouni</a>	Chapter 1
<a href="#">米雷克·米斯库夫</a>	第21章	<a href="#">Mirec Miskuf</a>	Chapter 21
<a href="#">姆诺罗尼亚</a>	第1和149章	<a href="#">mnorohna</a>	Chapters 1 and 149
<a href="#">穆罕默德·朱尔菲卡尔</a>	第123章	<a href="#">Mohammad Julfikar</a>	Chapter 123
<a href="#">莫谢梅雷莱斯</a>	第1章	<a href="#">moshemeirelles</a>	Chapter 1
<a href="#">MrP01</a>	第20章	<a href="#">MrP01</a>	Chapter 20
<a href="#">姆图奥维宁</a>	第92章	<a href="#">mrtuovinen</a>	Chapter 92
<a href="#">MSD</a>	第1章	<a href="#">MSD</a>	Chapter 1
<a href="#">MSeifert</a>	第9、16、19、21、26、33、37、41、43、45、48、55、64、67、68、69、70、71、72、73和200章	<a href="#">MSeifert</a>	Chapters 9, 16, 19, 21, 26, 33, 37, 41, 43, 45, 48, 55, 64, 67, 68, 69, 70, 71, 72, 73 and 200
<a href="#">泥鱼</a>	第1、20、21、33、37、57、111和149章	<a href="#">muddyfish</a>	Chapters 1, 20, 21, 33, 37, 57, 111 and 149
<a href="#">穆昆达·莫德尔</a>	第37章	<a href="#">Mukunda Modell</a>	Chapter 37
<a href="#">蒙塔西尔·阿拉姆</a>	第1章	<a href="#">Muntasir Alam</a>	Chapter 1
<a href="#">墨菲4</a>	第33章	<a href="#">Murphy4</a>	Chapter 33
<a href="#">MYGz</a>	第40章	<a href="#">MYGz</a>	Chapter 40
<a href="#">纳加2拉贾</a>	第150章	<a href="#">Naga2Raja</a>	Chapter 150
<a href="#">南德尔·斯皮尔斯特拉</a>	第40、75和116章	<a href="#">Nander Speerstra</a>	Chapters 40, 75 and 116
<a href="#">纳伦</a>	第42章和第89章	<a href="#">naren</a>	Chapters 42 and 89
<a href="#">内森·奥斯曼</a>	第190章	<a href="#">Nathan Osman</a>	Chapter 190
<a href="#">内森尼尔·福特</a>	第1章、第29章和第41章	<a href="#">Nathaniel Ford</a>	Chapters 1, 29 and 41
<a href="#">ncmathsadist</a>	第200章	<a href="#">ncmathsadist</a>	Chapter 200
<a href="#">nd.</a>	第33章	<a href="#">nd.</a>	Chapter 33
<a href="#">尼希米亚</a>	第173章	<a href="#">nehemiah</a>	Chapter 173
<a href="#">nemesisfixx</a>	第10章	<a href="#">nemesisfixx</a>	Chapter 10
<a href="#">镍</a>	第1章和第92章	<a href="#">Ni.</a>	Chapters 1 and 92
<a href="#">尼克·胡姆里奇</a>	第139章	<a href="#">Nick Humrich</a>	Chapter 139
<a href="#">尼古拉斯</a>	第29章	<a href="#">Nicolás</a>	Chapter 29
<a href="#">妮可·怀特</a>	第5章	<a href="#">Nicole White</a>	Chapter 5
<a href="#">niemmi</a>	第149章	<a href="#">niemmi</a>	Chapter 149
<a href="#">niyasc</a>	第1章、第43章、第45章和第149章	<a href="#">niyasc</a>	Chapters 1, 43, 45 and 149
<a href="#">nlsdfnbch</a>	第5章、第19章、第28章、第30章、第53章、第67章、第117章、第120章和第121章	<a href="#">nlsdfnbch</a>	Chapters 5, 19, 28, 30, 53, 67, 117, 120 and 121
<a href="#">努尔·查维奇</a>	第40章	<a href="#">Nour Chawich</a>	Chapter 40
<a href="#">nou</a>	第1、19、21、28、88和149章	<a href="#">nou</a>	Chapters 1, 19, 21, 28, 88 and 149
<a href="#">努希尔·梅赫迪</a>	第173章	<a href="#">Nuhil Mehdy</a>	Chapter 173
<a href="#">numbermaniac</a>	第1章和第9章	<a href="#">numbermaniac</a>	Chapters 1 and 9
<a href="#">obust</a>	第54章	<a href="#">obust</a>	Chapter 54
<a href="#">奥哈德·埃坦</a>	第5章	<a href="#">Ohad Eytan</a>	Chapter 5
<a href="#">ojas mohril</a>	第38章	<a href="#">ojas mohril</a>	Chapter 38
<a href="#">omgimanerd</a>	第200章	<a href="#">omgimanerd</a>	Chapter 200
<a href="#">东方</a>	第8章、第21章、第75章、第112章和第189章	<a href="#">Or East</a>	Chapters 8, 21, 75, 112 and 189
<a href="#">OrangeTux</a>	第149章	<a href="#">OrangeTux</a>	Chapter 149
<a href="#">奥尔托马拉·洛克尼</a>	第173章	<a href="#">Ortomala Lokni</a>	Chapter 173
<a href="#">orvi</a>	第1章、第25章、第41章、第125章、第133章和第134章	<a href="#">orvi</a>	Chapters 1, 25, 41, 125, 133 and 134
<a href="#">奥兹酒吧</a>	第20章	<a href="#">Oz Bar</a>	Chapter 20
<a href="#">奥扎尔·卡弗雷</a>	第30章	<a href="#">Ozair Kafray</a>	Chapter 30
<a href="#">熊猫</a>	第21章	<a href="#">Panda</a>	Chapter 21
<a href="#">帕罗西亚</a>	第23章和第69章	<a href="#">Parousia</a>	Chapters 23 and 69

[帕沙](#)  
[帕特里克·霍](#)  
[保罗](#)  
[保罗·韦弗](#)  
[paulmorriss](#)  
[保罗·弗雷塔斯](#)  
[保罗·斯卡丁](#)  
[帕万·纳特](#)  
[pcurry](#)  
[彼得·布里坦](#)  
[彼得·莫尔高·帕勒森](#)  
[彼得·辛纳斯](#)  
[petrs](#)  
[philngo](#)  
[Pigman168](#)  
[开心果](#)  
[pktangyue](#)  
[poke](#)  
[PolyGeo](#)  
[墨粟](#)  
[pperry](#)  
[普雷姆·纳莱恩](#)  
[普雷斯顿](#)  
[proprefenetre](#)  
[proprius](#)  
[PSN](#)  
[pylang](#)  
[PYPL](#)  
[Pythonista](#)  
[pzp](#)  
[Quill](#)  
[qwertyuiop9](#)  
[R·科尔梅纳雷斯](#)  
[R·纳尔](#)  
[安德拉什·拉普利](#)  
[雷吉斯·B.](#)  
[拉加夫](#)  
[拉胡尔·奈尔](#)  
[RahulHP](#)  
[rajah9](#)  
[拉姆·格兰迪](#)  
[RandomHash](#)  
[rassar](#)  
[ravigadila](#)  
[Razik](#)  
[Rednivrug](#)  
[regnarg](#)  
[Reut Sharabani](#)  
[rfkortekaas](#)  
[里卡多](#)  
[里卡多·佩特拉利亚](#)  
[理查德·菲茨休](#)  
[rlee827](#)

第3章、第20章、第21章、第33章、第37章、第38章、第67章、第70章、第77章、第83章和第149章  
第1章和第200章  
第5章  
第88章  
第5章  
第149章  
第39章  
第1章、第20章和第179章  
第29章、第110章和第149章  
第77章  
第73章  
第109章  
第77章  
第16章  
第96章  
第38章  
第149章  
第20章  
第145章  
第149章  
第55章  
第59章  
第138章和第173章  
第147章  
第5章  
第1章  
第1、8、21、33、38、43、53、54、73、80、128、147、173、192和200章  
第79章  
第32章和第149章  
第1、29、30、33、41、49、64和67章  
第1章  
第161、173和181章  
第10章  
第21章  
第82章  
第173章  
第125章  
第1章、第21章、第43章、第88章和第143章  
第5章、第8章、第45章、第53章、第64章和第112章  
第14章、第16章和第45章  
第1章  
第95章  
第172章  
第20、60、85、91和104章  
第158章  
第2、35和63章  
第75章  
第64和200章  
第1章和第115章  
第157章  
第21章、第84章和第117章  
第38章  
第62章

[Pasha](#)  
[Patrick Haugh](#)  
[Paul](#)  
[Paul Weaver](#)  
[paulmorriss](#)  
[Paulo Freitas](#)  
[Paulo Scardine](#)  
[Pavan Nath](#)  
[pcurry](#)  
[Peter Brittain](#)  
[Peter Mølgaard Pallesen](#)  
[Peter Shinners](#)  
[petrs](#)  
[philngo](#)  
[Pigman168](#)  
[pistache](#)  
[pktangyue](#)  
[poke](#)  
[PolyGeo](#)  
[poppie](#)  
[pperry](#)  
[Prem Narain](#)  
[Preston](#)  
[proprefenetre](#)  
[proprius](#)  
[PSN](#)  
[pylang](#)  
[PYPL](#)  
[Pythonista](#)  
[pzp](#)  
[Quill](#)  
[qwertyuiop9](#)  
[R Colmenares](#)  
[R Nar](#)  
[Rápli András](#)  
[Régis B.](#)  
[Raghav](#)  
[Rahul Nair](#)  
[RahulHP](#)  
[rajah9](#)  
[Ram Grandhi](#)  
[RandomHash](#)  
[rassar](#)  
[ravigadila](#)  
[Razik](#)  
[Rednivrug](#)  
[regnarg](#)  
[Reut Sharabani](#)  
[rfkortekaas](#)  
[Ricardo](#)  
[Riccardo Petraglia](#)  
[Richard Fitzhugh](#)  
[rlee827](#)

Chapters 3, 20, 21, 33, 37, 38, 67, 70, 77, 83 and 149  
Chapters 1 and 200  
Chapter 5  
Chapter 88  
Chapter 5  
Chapter 149  
Chapter 39  
Chapters 1, 20 and 179  
Chapters 29, 110 and 149  
Chapter 77  
Chapter 73  
Chapter 109  
Chapter 77  
Chapter 16  
Chapter 96  
Chapter 38  
Chapter 149  
Chapter 20  
Chapter 145  
Chapter 149  
Chapter 55  
Chapter 59  
Chapters 138 and 173  
Chapter 147  
Chapter 5  
Chapter 1  
Chapters 1, 8, 21, 33, 38, 43, 53, 54, 73, 80, 128, 147, 173, 192 and 200  
Chapter 79  
Chapters 32 and 149  
Chapters 1, 29, 30, 33, 41, 49, 64 and 67  
Chapter 1  
Chapters 161, 173 and 181  
Chapter 10  
Chapter 21  
Chapter 82  
Chapter 173  
Chapter 125  
Chapters 1, 21, 43, 88 and 143  
Chapters 5, 8, 45, 53, 64 and 112  
Chapters 14, 16 and 45  
Chapter 1  
Chapter 95  
Chapter 172  
Chapters 20, 60, 85, 91 and 104  
Chapter 158  
Chapters 2, 35 and 63  
Chapter 75  
Chapters 64 and 200  
Chapters 1 and 115  
Chapter 157  
Chapters 21, 84 and 117  
Chapter 38  
Chapter 62

[rll](#)  
[罗布·H](#)  
[罗布·默里](#)  
[罗嫩·内斯](#)  
[ronrest](#)  
[罗伊·雅各布](#)  
[rrao](#)  
[rrawat](#)  
[瑞安·史密斯](#)  
[ryanyuyu](#)  
[Ry](#)  
[萨钦·卡尔库尔](#)  
[sagism](#)  
[赛福尔·阿扎德](#)  
[萨姆·克里格谢尔德](#)  
[萨姆·怀特德](#)  
[桑吉斯·苏迪尔](#)  
[萨基布·沙姆西](#)  
[Sardathrion](#)  
[萨尔多尔贝克·伊莫马利耶夫](#)  
[sarvajeetsuman](#)  
[SashaZd](#)  
[satsumas](#)  
[sayan](#)  
[斯科特·默梅尔斯坦](#)  
[塞巴斯蒂安·施拉德](#)  
[塞尔丘克](#)  
[森普](#)  
[宁静](#)  
[SerialDev](#)  
[serv](#)  
[塞思·M·拉尔森](#)  
[七](#)  
[sevenforce](#)  
[ShadowRanger](#)  
[尚塔努·阿尔希](#)  
[肖恩·米汉](#)  
[希哈布·沙里亚尔](#)  
[Shijo](#)  
[鞋子](#)  
[施雷·古普塔](#)  
[施雷亚什·S·萨尔纳亚克](#)  
[朔](#)  
[SiggyF](#)  
[西蒙·弗雷泽](#)  
[西蒙·希布斯](#)  
[Simplans](#)  
[西拉朱斯·萨拉伊欣](#)  
[sisanared](#)  
[skrrgwasme](#)  
[SN·拉维昌德兰 KR](#)  
[solarc](#)

第21章  
第121章  
第94章  
第30章  
第19章和第41章  
第19章  
第1章  
第161章  
第21章和第120章  
第21章  
第149章  
第125章  
第5章  
第43章  
第1章  
第111章  
第1章  
第92章  
第103章  
第103章  
第9章和第16章  
第12章、第14章、第29章、第64章、第86章、第134章、第143章、第144章、第146章和第194章  
第67章  
第1章和第96章  
第110章、第135章和第149章  
第173章  
第1章、第28章、第149章和第201章  
第38章  
第20章、第30章、第40章、第43章、第149章和第173章  
第82章  
第40章  
第54章和第87章  
第1章、第11章、第20章和第33章  
第16章和第67章  
第149章  
第173章  
第10章、第15章、第19章、第20章和第47章  
第200章  
第198章  
第21章  
第41章、第56章和第173章  
第12章  
第77章  
第16章和第111章  
第173章  
第169章  
第1、9、10、14、16、19、21、28、33、37、38、41、43、44、45、50、69、75、77、82、88、99、147、149和173章  
第5、175和176章  
第39章  
第16和99章  
第75章  
第20和149章

[rll](#)  
[Rob H](#)  
[Rob Murray](#)  
[Ronen Ness](#)  
[ronrest](#)  
[Roy Iacob](#)  
[rrao](#)  
[rrawat](#)  
[Ryan Smith](#)  
[ryanyuyu](#)  
[Ry](#)  
[Sachin Kalkur](#)  
[sagism](#)  
[Saiful Azad](#)  
[Sam Krygsheld](#)  
[Sam Whited](#)  
[Sangeeth Sudheer](#)  
[Saqib Shamsi](#)  
[Sardathrion](#)  
[Sardorbek Imomaliev](#)  
[sarvajeetsuman](#)  
[SashaZd](#)  
[satsumas](#)  
[sayan](#)  
[Scott Mermelstein](#)  
[Sebastian Schrader](#)  
[Selcuk](#)  
[Sempoo](#)  
[Serenity](#)  
[SerialDev](#)  
[serv](#)  
[Seth M. Larson](#)  
[Seven](#)  
[sevenforce](#)  
[ShadowRanger](#)  
[Shantanu Alshi](#)  
[Shawn Mehan](#)  
[Shihab Shahriar](#)  
[Shijo](#)  
[Shoe](#)  
[Shrey Gupta](#)  
[Shreyash S Sarnayak](#)  
[Shuo](#)  
[SiggyF](#)  
[Simon Fraser](#)  
[Simon Hibbs](#)  
[Simplans](#)  
[Sirajus Salayhin](#)  
[sisanared](#)  
[skrrgwasme](#)  
[SN Ravichandran KR](#)  
[solarc](#)

Chapter 21  
Chapter 121  
Chapter 94  
Chapter 30  
Chapters 19 and 41  
Chapter 19  
Chapter 1  
Chapter 161  
Chapters 21 and 120  
Chapter 21  
Chapter 149  
Chapter 125  
Chapter 5  
Chapter 43  
Chapter 1  
Chapter 111  
Chapter 1  
Chapter 92  
Chapter 103  
Chapter 103  
Chapters 9 and 16  
Chapters 12, 14, 29, 64, 86, 134, 143, 144, 146 and 194  
Chapter 67  
Chapters 1 and 96  
Chapters 110, 135 and 149  
Chapter 173  
Chapters 1, 28, 149 and 201  
Chapter 38  
Chapters 20, 30, 40, 43, 149 and 173  
Chapter 82  
Chapter 40  
Chapters 54 and 87  
Chapters 1, 11, 20 and 33  
Chapters 16 and 67  
Chapter 149  
Chapter 173  
Chapters 10, 15, 19, 20 and 47  
Chapter 200  
Chapter 198  
Chapter 21  
Chapters 41, 56 and 173  
Chapter 12  
Chapter 77  
Chapters 16 and 111  
Chapter 173  
Chapter 169  
Chapters 1, 9, 10, 14, 16, 19, 21, 28, 33, 37, 38, 41, 43, 44, 45, 50, 69, 75, 77, 82, 88, 99, 147, 149 and 173  
Chapters 5, 175 and 176  
Chapter 39  
Chapters 16 and 99  
Chapter 75  
Chapters 20 and 149

[苏门德拉·库马尔·萨胡](#)第14章和第38章

[苏维克·马吉](#) 第1章

[sricharan](#) 第149章

[StardustGogeta](#) 第43章

[stark](#) 第1章

[斯蒂芬·尼亞姆韦亚](#) 第161章

[史蒂夫·巴恩斯](#) 第33章和第82章

[史蒂文·莫德](#) 第11章、第33章、第47章和第92章

[sth](#) 第91章、第92章和第141章

[strpeter](#) 第85章和第192章

[StuxCrystal](#) 第21章、第37章、第43章、第56章、第76章、第82章、第121章和第142章

[苏迪普·班达里](#) 第88章

[孙庆尧](#) 第101章

[桑尼·帕特尔](#) 第21章

[SuperBiasedMan](#)

[supersam654](#)

[surfthecity](#)

[Symmitchry](#) 第47章和第53章

[sytech](#) 第92章

[Shadowfa](#) 第1章和第134章

[塔德格·麦克唐纳](#) 第149章

[talhasch](#) 第92章、第93章和第164章

[塔斯迪克·拉赫曼](#) 第30章

[泰勒·斯威夫特](#) 第1章

[techydesigner](#) 第1章、第38章、第43章和第190章

[Teepeemm](#) 第152章

[特贾斯·贾达夫](#) 第201章

[特朱斯·普拉萨德](#) 第1章

[TemporalWolf](#)

[textshell](#)

[TheGenie OfTruth](#)

[扫帚头](#)

[猫女士](#)

[咖喱人](#)

[托马斯·阿勒](#)

[托马斯·克劳利](#) 第60章和第134章

[托马斯·格罗特](#) 第105章

[托马斯·莫罗](#) 第30章、第43章、第58章、第61章、第126章和第181章

[图图](#) 第119章

[蒂姆](#) 第80章

[蒂姆D](#) 第149章

[tjohnson](#) 第200章

[tlo](#) 第44章

[tobias\\_k](#) 第82章

[汤姆](#) 第28章和第40章

[汤姆·巴伦](#) 第16章

[汤姆·德·格乌斯](#) 第1章和第21章

[托尼·迈耶](#) 第1章

[托尼·萨福克66](#) 第43章

[tox123](#) 第9章、第10章、第38章和第40章

[TuringTux](#) 第38章

[泰勒·克朗普顿](#) 第4章

[第86章](#)

[Soumendra Kumar Sahoo](#) Chapters 14 and 38

[SouvikMaji](#) Chapter 1

[sricharan](#) Chapter 149

[StardustGogeta](#) Chapter 43

[stark](#) Chapter 1

[Stephen Nyamweya](#) Chapter 161

[Steve Barnes](#) Chapters 33 and 82

[Steven Maude](#) Chapters 11, 33, 47 and 92

[sth](#) Chapters 91, 92 and 141

[strpeter](#) Chapters 85 and 192

[StuxCrystal](#) Chapters 21, 37, 43, 56, 76, 82, 121 and 142

[Sudip Bhandari](#) Chapter 88

[Sun Qingyao](#) Chapter 101

[Sunny Patel](#) Chapter 21

[SuperBiasedMan](#) Chapters 1, 4, 15, 16, 20, 21, 26, 29, 30, 33, 41, 43, 64, 69, 77, 80, 88, 132, 149 and 200

[supersam654](#) Chapter 70

[surfthecity](#) Chapter 6

[Symmitchry](#) Chapters 47 and 53

[sytech](#) Chapter 92

[Shadowfa](#) Chapters 1 and 134

[Tadhg McDonald](#) Chapter 149

[talhasch](#) Chapters 92, 93 and 164

[Tasdk Rahman](#) Chapter 30

[taylor swift](#) Chapter 1

[techydesigner](#) Chapters 1, 38, 43 and 190

[Teepeemm](#) Chapter 152

[Tejas Jadhav](#) Chapter 201

[Tejas Prasad](#) Chapter 1

[TemporalWolf](#) Chapter 90

[textshell](#) Chapters 16, 28, 33 and 121

[TheGenie OfTruth](#) Chapter 1

[theheadofabroom](#) Chapters 41, 49 and 64

[the\\_cat\\_lady](#) Chapter 43

[The\\_Curry\\_Man](#) Chapter 16

[Thomas Ahle](#) Chapters 60 and 134

[Thomas Crowley](#) Chapter 105

[Thomas Gerot](#) Chapters 30, 43, 58, 61, 126 and 181

[Thomas Moreau](#) Chapter 119

[Thtru](#) Chapter 80

[Tim](#) Chapter 149

[Tim D](#) Chapter 200

[tjohnson](#) Chapter 44

[tlo](#) Chapter 82

[tobias\\_k](#) Chapters 28 and 40

[Tom](#) Chapter 16

[Tom Barron](#) Chapters 1 and 21

[Tom de Geus](#) Chapter 1

[Tony Meyer](#) Chapter 43

[Tony Suffolk 66](#) Chapters 9, 10, 38 and 40

[tox123](#) Chapter 38

[TuringTux](#) Chapter 4

[Tyler Crompton](#) Chapter 86

<a href="#">泰勒·古巴拉</a>	第119章和第122章	<a href="#">Tyler Gubala</a>	Chapters 119 and 122
<a href="#">乌迪</a>	第54章	<a href="#">Udi</a>	Chapter 54
<a href="#">超级鲍勃</a>	第38章	<a href="#">UltraBob</a>	Chapter 38
<a href="#">海坊主</a>	第30章	<a href="#">Umibozu</a>	Chapter 30
<a href="#">安德里克斯</a>	第49章	<a href="#">Underyx</a>	Chapter 49
<a href="#">撤销</a>	第9章	<a href="#">Undo</a>	Chapter 9
<a href="#">unutbu</a>	第116章	<a href="#">unutbu</a>	Chapter 116
<a href="#">user2027202827</a>	第163章	<a href="#">user2027202827</a>	Chapter 163
<a href="#">user2314737</a>	第8、9、13、16、20、28、30、33、38、40、41、57、60、64、69、75、88、110、134、142、149、154、161、196和200章	<a href="#">user2314737</a>	Chapters 8, 9, 13, 16, 20, 28, 30, 33, 38, 40, 41, 57, 60, 64, 69, 75, 88, 110, 134, 142, 149, 154, 161, 196 and 200
<a href="#">user2683246</a>	第43和187章	<a href="#">user2683246</a>	Chapters 43 and 187
<a href="#">user2728397</a>	第166章	<a href="#">user2728397</a>	Chapter 166
<a href="#">user2853437</a>	第1章	<a href="#">user2853437</a>	Chapter 1
<a href="#">user312016</a>	第1章、第40章和第77章	<a href="#">user312016</a>	Chapters 1, 40 and 77
<a href="#">user3333708</a>	第33章	<a href="#">user3333708</a>	Chapter 33
<a href="#">user405</a>	第33章	<a href="#">user405</a>	Chapter 33
<a href="#">user6457549</a>	第20章	<a href="#">user6457549</a>	Chapter 20
<a href="#">乌萨夫·T</a>	第20章	<a href="#">Utsav T</a>	Chapter 20
<a href="#">vaichidrewar</a>	第1章	<a href="#">vaichidrewar</a>	Chapter 1
<a href="#">valeas</a>	第161章	<a href="#">valeas</a>	Chapter 161
<a href="#">瓦伦丁·洛伦茨</a>	第20章、第21章、第43章、第110章和第149章	<a href="#">Valentin Lorentz</a>	Chapters 20, 21, 43, 110 and 149
<a href="#">瓦洛尔·纳拉姆</a>	第43章	<a href="#">Valor Naram</a>	Chapter 43
<a href="#">vaultah</a>	第20、33、43和77章	<a href="#">vaultah</a>	Chapters 20, 33, 43 and 77
<a href="#">Veedrac</a>	第21、33、41和110章	<a href="#">Veedrac</a>	Chapters 21, 33, 41 and 110
<a href="#">维卡什·库马尔·贾因</a>	第174章	<a href="#">Vikash Kumar Jain</a>	Chapter 174
<a href="#">Vin</a>	第1和17章	<a href="#">Vin</a>	Chapters 1 and 17
<a href="#">维纳亚克</a>	第149章	<a href="#">Vinayak</a>	Chapter 149
<a href="#">vinzee</a>	第99、116和120章	<a href="#">vinzee</a>	Chapters 99, 116 and 120
<a href="#">viveksyঞ্চ</a>	第10章和第19章	<a href="#">viveksyঞ্চ</a>	Chapters 10 and 19
<a href="#">VJ 马加尔</a>	第31章	<a href="#">VJ Magar</a>	Chapter 31
<a href="#">弗拉德·贝兹登</a>	第103章	<a href="#">Vlad Bezden</a>	Chapter 103
<a href="#">weewooquestionaire</a>	第1章	<a href="#">weewooquestionaire</a>	Chapter 1
<a href="#">魏中图</a>	第41章和第149章	<a href="#">WeizhongTu</a>	Chapters 41 and 149
<a href="#">威克拉马兰加</a>	第53章	<a href="#">Wickramaranga</a>	Chapter 53
<a href="#">威尔</a>	第5、15、16、21、30、33、91、116、117、121和164章	<a href="#">Will</a>	Chapters 5, 15, 16, 21, 30, 33, 91, 116, 117, 121 and 164
<a href="#">温斯顿·沙伦</a>	第155章	<a href="#">Winston Sharon</a>	Chapter 155
<a href="#">弗拉基米尔·帕兰特</a>	第21章、第37章和第197章	<a href="#">Vladimir Palant</a>	Chapters 21, 37 and 197
<a href="#">wnnmaw</a>	第41章、第43章和第53章	<a href="#">wnnmaw</a>	Chapters 41, 43 and 53
<a href="#">沃尔夫</a>	第149章	<a href="#">Wolf</a>	Chapter 149
<a href="#">WombatPM</a>	第30章	<a href="#">WombatPM</a>	Chapter 30
<a href="#">Wombatz</a>	第200章	<a href="#">Wombatz</a>	Chapter 200
<a href="#">wrwrwr</a>	第173章	<a href="#">wrwrwr</a>	Chapter 173
<a href="#">wwii</a>	第14章	<a href="#">wwii</a>	Chapter 14
<a href="#">wythagoras</a>	第9章	<a href="#">wythagoras</a>	Chapter 9
<a href="#">泽维尔·康贝尔</a>	第15章、第19章、第111章和第120章	<a href="#">Xavier Combelle</a>	Chapters 15, 19, 111 and 120
<a href="#">XCoder Real</a>	第47章	<a href="#">XCoder Real</a>	Chapter 47
<a href="#">xgord</a>	第14章和第30章	<a href="#">xgord</a>	Chapters 14 and 30
<a href="#">XonAether</a>	第52章	<a href="#">XonAether</a>	Chapter 52
<a href="#">xtreak</a>	第67章和第149章	<a href="#">xtreak</a>	Chapters 67 and 149
<a href="#">Y0da</a>	第85章	<a href="#">Y0da</a>	Chapter 85
<a href="#">ygram</a>	第190章	<a href="#">ygram</a>	Chapter 190
<a href="#">约根德拉·夏尔马</a>	第1章和第117章	<a href="#">Yogendra Sharma</a>	Chapters 1 and 117
<a href="#">yurib</a>	第64章	<a href="#">yurib</a>	Chapter 64

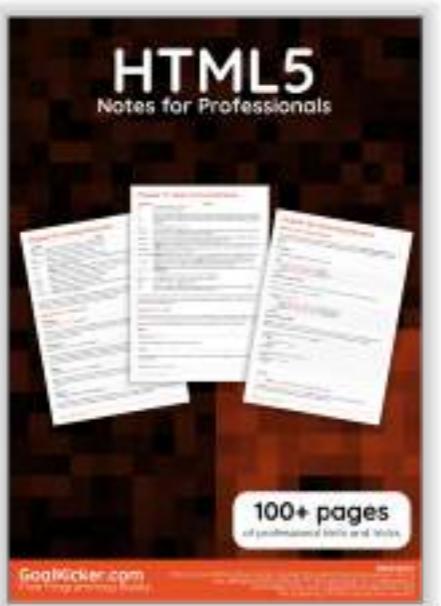
[扎克·贾尼基](#)  
[Zags](#)  
[扎伊德·阿贾杰](#)  
[zarak](#)  
[Zaz](#)  
[zenlc2000](#)  
[石占平](#)  
[zmo](#)  
[zondo](#)  
[zopieux](#)  
[zvone](#)  
[zxxz](#)  
[Zydnar](#)  
[KİSTÖF](#)  
[λuser](#)  
[某人](#)

第1章  
第1章  
第67章  
第149章  
第21章  
第34章  
第145章  
第83章、第140章和第141章  
第75章和第83章  
第173章  
第13章、第37章、第39章和第112章  
第33章  
第81章  
第117章  
第67章和第77章  
第24章、第37章和第128章

[Zach Janicki](#)  
[Zags](#)  
[Zaid Ajaj](#)  
[zarak](#)  
[Zaz](#)  
[zenlc2000](#)  
[Zhanping Shi](#)  
[zmo](#)  
[zondo](#)  
[zopieux](#)  
[zvone](#)  
[zxxz](#)  
[Zydnar](#)  
[KİSTÖF](#)  
[λuser](#)  
[Некто](#)

Chapter 1  
Chapter 1  
Chapter 67  
Chapter 149  
Chapter 21  
Chapter 34  
Chapter 145  
Chapters 83, 140 and 141  
Chapters 75 and 83  
Chapter 173  
Chapters 13, 37, 39 and 112  
Chapter 33  
Chapter 81  
Chapter 117  
Chapters 67 and 77  
Chapters 24, 37 and 128

## 你可能也喜欢



## You may also like

