## 一、基本的python使用概念

### ****1.1 Python脚本的调用方式****

在linux中，python、shell这些程序都是普通的文本格式，都需要一种程序去解释执行它。要么调用的时候指定，要么在文件头指定。

1. 直接使用python xxxx.py执行。其中python可以写成python的绝对路径。使用which python进行查询。

2、在文件的头部（第一行）写上**#!/usr/bin/python，**这个地方使用python的绝对路径，就是上面用which python查询来的结果。然后在外面就可以使用./xxx.py执行了。

### ****1.2 源程序编码方式****

默认情况下, Python 源码文件以UTF-8 编码. 在这种编码下, 世界上大多数语言的字符都可以用于, 字符串常量, 标识符, 以及注释——尽管标准库遵循一个所有可移植代码都应遵守的约定: 仅使用ASCII 字符作为标识符. 要正确地显示所有这些字符, 你的编辑器一定要有能力辨认出是UTF-8 编码, 还要使用一个支持所有文件中字符的字体.

也可以为源码文件指定不同的编码. 为此, 要在#! 行后面指定一个特

殊的注释行, 以定义源码文件的编码:

# -\*- coding: encoding -\*-

有了这样的声明，源文件中的所有字符都会被以encoding 的编码来解读，

而非是UTF-8.

## 二、数据结构

### 2.1 字符串 ' '

（1）定义

字符串是用单引号或双引号包裹起来的。

字符串有几种方法来跨越多行：

第一种，利用单引号或双引号，在一行最后加上一个反斜杠以表明下一行是这行的逻辑延续:

另一种方法, 字符串可以使用一对匹配的三引号对包围: """ 或'''. 当使用三引号时, 回车不需要被舍弃, 他们会包含在字符串里.

（2）操作

|  |
| --- |
| word = 'Help ' + 'A'  'str' 'ing' #该操作只支持常量  '<' + word\*5 + '>'  word[-1] # 最后一个字符  word[-2] # 倒数第二个字符  word[-2:] # 最后两个字符  word[:-2] #除了最后两个字符以外的字符  len(word) #表示字符串长度 |
|  |

### 2.2 列表相关 [ ]

#### 2.2.1 简介

Python 有一些**复合数据类型**, 用来把其它值分组. 最全能的就是list,它可以写为在方括号中的通过逗号分隔的一列值(项). 列表的项并不需要是同一类型。

就像字符串索引, 列表的索引从0 开始, 列表也可以切片, 连接等等。

所有的切片操作返回一个包含请求元素的新列表。

|  |
| --- |
| a = ['spam ', 'eggs', 100 , 1234]  a[1:-1] #第二个到倒数第二个  3\*a[:3] + ['Boo!']  a[:] #返回a的浅复制  a[2] = a[2] + 23 #列表中的单个元素是可变的，包括列表的长度、内容、值等都是可以变的  a[0:2] = [1, 12] #替代一些项  a[0:2] = [] #移除一些项  a[1:1] = ['bletch ', 'xyzzy '] #插入一些项  a[:] = [] #清空列表  len(a) # len也有效  p = [1, [1,2], 4] #可以嵌套列表  p[1].append('xtra') #元素尾部加入一些内容 |

#### 2.2.2 常见用法

|  |
| --- |
| list.**append(x)**  在列表的尾部添加一个项; 等价于a[len(a):]=[x]  list.**extend(L)**  用给入的列表将当前列表接长; 等价于a[len(a):]=L  list.**insert(i, x)**  在给定的位置上插入项. 第一个参数就是准备在它之前插入的元素的索引, 因此a.insert(0,x) 会在列表的头部插入, 而a.insert(len(a),x) 则等价于a.append(x)  list.**remove(x)**  移除列表中第一个值为x 的项. 没有符合要求的项时, 会产生一个错误.  list.**pop([i])**  删除列表给定位置的项, 并返回它. 如果没有指定索引, a.pop 移除并返回列表的最后一项. (函式原型的i 在中方括号中意味着它是一个可选参数, 而不是你应当在那里键入一个方括号. 你将会在Python 库参考中经常见到这种表示法.)  list.**index(x)**  返回列表中第一个值为x 的项索引值. 如果没有匹配的项, 则产生一个错误  list.**count(x)**  返回列表中x 出现的次数  list.**sort()**  就地完成列表排序  list.**reverse()**  就地完成列表项的翻转 |

**（1）把列表当作堆栈用**

堆栈的特性是最后添加的元素就是第一个取出的元素(即“后入先出”). 要在栈顶添加一个项, 就使用**append()**. 要从栈顶取回一个项, 就使用不带显式索引的**pop()**

**（2）把列表或者collection.deque当作队列用**

从列表的尾部添加和弹出是很快的, 而在列表的开头插入或弹出是慢的(因为所有元素都得移动一个位置).

要实现一个队列, 使用collection.deque, 它被设计成在两端添加和弹出都很快

|  |
| --- |
| **from** collections **import** deque  >>> queue = deque(["Eric", "John", "Michael"])  >>> queue.append("Terry") # Terry 进入  >>> queue.append("Graham") # Graham 进入  >>> queue.popleft() # 第一个进入的现在离开  'Eric'  >>> queue.popleft() # 第二个进入的现在离开  'John' |

**（3）列表推导式**

列表推导式的结构是, 在一个方括号里, 首先是一个表达式, 随后是一个for 子句, 然后是零个或更多的for 或if 子句. 结果将是通过计算for和if 子句来获得的一个列表.

如果要使表达式推导式出元组, 就必须用圆括号.

|  |
| --- |
| >>> vec = [2, 4, 6]  >>> [[x, x\*\*2] **for** x **in** vec]  [[2, 4], [4, 16], [6, 36]]  >>> [3\*x **for** x **in** vec **if** x > 3]  [12 , 18]  >>> [(x, x\*\*2) **for** x **in** vec]  [(2, 4), (4, 16), (6, 36)]  >>> vec1 = [2, 4, 6]  >>> vec2 = [4, 3, -9]  >>> [x\*y **for** x **in** vec1 **for** y **in** vec2]  [8, 6, -18 , 16 , 12 , -36 , 24 , 18 , -54]  >>> [x+y **for** x **in** vec1 **for** y **in** vec2]  [6, 5, -7, 8, 7, -5, 10 , 9, -3]  >>> [vec1[i]\*vec2[i] **for** i **in** range(len(vec1))]  [8, 12 , -54]  >>> [str(round(355/113 , i)) **for** i **in** range(1, 6)]  ['3.1', '3.14', '3.142', '3.1416', '3.14159 '] |

**（4）del语句**

通过给定索引而不是值, 来删除列表中项的方法: 用**del** 语句. 它与返回一个值的**pop()** 方法不同. **del** 语句也可以移除列表中的切片, 或者清除整个列表(之前我们通过给切片赋值为空列表来完成这点)。

|  |
| --- |
| >>> **del** a[0]  >>> **del** a[2:4]  >>> **del** a[:]  >>> **del** a #删除变量实体，后面引用a就会出错 |

### 2.3元组（ ）和序列

另一种标准序列数据类型: 元组。元组由若干逗号分隔的值组成, 一般需要用圆括号包含，例如:

|  |
| --- |
| >>> t = 12345 , 54321 , 'hello!'  >>> t[0]  12345  >>> t  (12345 , 54321 , 'hello!')  >>> u = t, (1, 2, 3, 4, 5)  >>> u  ((12345 , 54321 , 'hello!'), (1, 2, 3, 4, 5)) |

元组有许多用途. 例如: (x, y) 坐标对, 数据库里的员工记录等. 元组同字串都是不可变的: 无法对元组指定项进行赋值(尽管可通过切片和连接来模拟这个操作). 元组中可以包含可变的对象, 如列表。

构造包含0 或1 个项的元组是个特殊问题: 语法上为了适应这一情况,有些额外的规则.

**空元组由一对空的圆括号构造;**

**一个项的元组由一个值后面跟着一个逗号构造**(把一个值放入一对圆括号里并不足以构造一个元组)。

**（1）元组打包和序列解包**

|  |
| --- |
| >>> t = 12345,54321,'hello!' #元组打包  >>> x, y, z = t #序列解包 |

### 2.4 集合 set{ a }

集合是种无序不重复的元素集.基本用途包括成员关系测试和重复条目消除. 集合对象也支持合(union),交(intersection), 差(difference), 和对称差(sysmmetric difference) 等数学操作。

花括号或函式set() 可用于创建集合. 注意: 创建一个空集合只能使用set(), 而不能使用{}; 后者是创建一个空字典。

|  |
| --- |
| >>> basket = {'apple ', 'orange ', 'apple ', 'pear', 'orange ', 'banana '}  >>> **print**(basket) # 重复的被移除了  {'orange ', 'banana ', 'pear', 'apple '}  >>> 'orange ' **in** basket # 快速成员关系测试  True  >>> 'crabgrass ' **in** basket  False  >>> a = set('abracadabra ')  >>> b = set('alacazam ')  >>> a # a 中的不重复字母  {'a', 'r', 'b', 'c', 'd'}  >>> a - b # a 中有而b中没有的字母  {'r', 'd', 'b'}  >>> a | b # 在a 中或在b中的字母  {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}  >>> a & b # a 和b 中都有的字母  {'a', 'c'}  >>> a ^ b # a 或b中只有一个有的字母  {'r', 'd', 'b', 'm', 'z', 'l'}  >>> a = {x **for** x **in** 'abracadabra ' **if** x **not in** 'abc'}  >>> a  {'r', 'd'} |

### 2.5 字典 { a:b }

在其它语言中字典一般被叫做“关联存储”或“关联数组”. 与使用某个范围作为索引的序列不一样, **字典通过键来索引, 而键可以是任意不可变类型;**

通常用**字符串和数字作为键.** 如果**元组只包含字符串和数字, 元组也可以作为键**; 但是, 当元组直接或间接地包含可变对象时, 就不能用作一个键. **不能使用列表作为键,** 因为列表可以通过索引, 切片原地赋值而被改变。

**字典**看成是一个**没有顺序的键*:* 值对集合, 键必须是唯一的**(在一个字典里). 一对花括号创建一个空字典: {}. 在括号中间放置的以逗号分隔的键:值对列表就是字典的初始键:值对. 这也是字典输出时的格式.

**（1）存储、取值与删除**

字典最主要的操作是通过某键存储一个值, 以及从给定的键里提取它的值.

如果你使用一个已被使用的键进行存储操作, 该键的旧值就没有了.

使用一个不存在的键提取值会产生一个错误.

使用**del** 可以删除一个键: 值对.

**（2）查询字典中的键值**

在一个字典上执行list(d.keys()) 返回该字典中所使用键的列表, 该列表的顺序不确定(如果需要有序, 只要使用sorted(d.keys()))。

**（3）检查键是否存在**

要检查某一个键是否在字典里, 使用**in** 关键字.

|  |
| --- |
| >>> tel = {'jack ': 4098 , 'sape': 4139}  >>> tel['guido '] = 4127  >>> **del** tel['sape']  >>> tel['irv'] = 4127  >>> list(tel.keys())  ['irv', 'guido ', 'jack']  >>> sorted(tel.keys())  ['guido ', 'irv', 'jack']  >>> 'guido ' **in** tel  True |

**（4）构造器dict()**

|  |
| --- |
| >>> dict([('sape ', 4139), ('guido ', 4127), ('jack', 4098)])  {'sape': 4139 , 'jack': 4098 , 'guido ': 4127}  >>> dict([(x, x\*\*2) **for** x **in** (2, 4, 6)]) # 使用列表推导式  {2: 4, 4: 16 , 6: 36}  若键为字符串, 有时用关键字参数指定键值对更为简单:  >>> dict(sape=4139 , guido=4127 , jack=4098)  {'sape': 4139 , 'jack': 4098 , 'guido ': 4127} |

### 2.6 遍历技巧

#### 2.6.1 enumerate()

对序列遍历时, 可以使用**enumerate()** 函式来同时取回位置索引和相应的值.

|  |
| --- |
| >>> **for** i, v **in** enumerate(['tic', 'tac', 'toe']):  ... **print**(i, v)  ...  0 tic  1 tac  2 toe |

#### 2.6.2 zip()

同时对两个或更多的序列进行遍历时, 可用**zip()** 进行组合.

|  |
| --- |
| >>> questions = ['name', 'quest ', 'favorite color ']  >>> answers = ['lancelot ', 'the holy grail ', 'blue']  >>> **for** q, a **in** zip(questions , answers):  ... **print**('What is your {0}? It is {1}.'.format(q, a))  ...  What **is** your name? It **is** lancelot.  What **is** your quest? It **is** the holy grail.  What **is** your favorite color? It **is** blue. |

#### 2.6.3 reversed()、sorted()

反向遍历序列和有序遍历序列

|  |
| --- |
| >>> **for** i **in** reversed(range(1, 10 , 2)):  ... **print**(i)  >>> basket = ['apple ', 'orange ', 'apple ', 'pear', 'orange ', 'banana ']  >>> **for** f **in** sorted(set(basket)):  ... **print**(f) |

#### 2.6.4 items()

当对字典遍历时, 可用**items()** 方法同时取回键和对应的值。

|  |
| --- |
| >>> knights = {'gallahad ': 'the pure', 'robin ': 'the brave '}  >>> **for** k, v **in** knights.items():  ... **print**(k, v)  gallahad the pure  robin the brave |

### 2.7 条件控制

在**while** 和**if** 语句中使用的条件可以包含任何操作符。

#### 2.7.1 比较操作符

• 比较操作符**in** 和**not in** 检查一个值是否在序列中.

• 操作符**is** 和**is not** 比较两个对象是否为同一对象; 这只对诸如列表的可变对象有用.

所有比较操作符具有相同的优先级, 低于所有的数值操作.比较操作符可以连起来使用. 例如, a < b == c 测试a 小于b 且b 与c相等.

#### 2.7.2 逻辑操作符

• 逻辑操作符的优先级又低于比较操作符;

• 这其中, **not** 优先级最高, 而**or** 的优先级最低, 因此A **and not** B **or** C 等价于(A **and** (**not** B)) **or** C . 同样, 可以使用圆括号来表达想要的结果.

• 逻辑操作符**and** 和**or** 被称为短路操作符: 它从左至右计算参数, 并且当结果确定时计算就立即停止.

比较操作(或其它任何布尔表达式) 都能用逻辑操作符**and** 和**or** 连接,结果值可以用**not** 取反。

|  |
| --- |
| >>> string1 , string2 , string3 = '', 'Trondheim ', 'Hammer Dance '  >>> non\_null = string1 **or** string2 **or** string3  >>> non\_null  'Trondheim ' |

## 三、python函数

### 3.1 流程控制

（1）If控制

if …:

elif …:

else :

（2）For控制

for x in a[:]: # 制造整个列表的切片复本

if len(x) > 6: a.insert(0, x)

（3）break 和continue 语句, 以及循环中的else 子句

break 语句工作得如同C 语言一样, 跳出最小的for 或while 循环.

循环语句可以有一个else 子句; 该子句会在以下情况被执行: 循环因迭代到列表末尾而终止(for 语句), 或者, 当循环条件为假(while 语句), 同时它不会在循环因break 语句终止的情况下被执行. 下面搜索素数的示例说明了这一特性:

|  |
| --- |
| >>> for n in range(2, 10):  ... for x in range(2, n):  ... if n % x == 0:  ... print(n, 'equals ', x, '\*', n//x)  ... break  ... else:  ... # loop fell through without finding a factor  ... print(n, 'is a prime number ')  ...  2 is a prime number  3 is a prime number  4 equals 2 \* 2  5 is a prime number  6 equals 2 \* 3  7 is a prime number  8 equals 2 \* 4  9 equals 3 \* 3  (是的, 这是正确的代码. 仔细看: else 子句属于for 循环, 而非是if 语句) |

与循环搭配使用时, else子句的行为和它与try语句的搭配时相对于它与if语句的搭配时有更多共性: try语句的else子句在没有异常发生时被执行, 循环的else子句在没有break语句是被执行.

continue语句同样是从C语言借用的, 它终止当前迭代而进行循环的下一次迭代.

(4) range函数和len函数

|  |
| --- |
| range(5, 10)  range(0, 10 , 3)  range(-10 , -100 , -30)  a = ['Mary ', 'had', 'a', 'little ', 'lamb']  for i in range(len(  print(i, a[i]) |

（5）pass语句

pass 语句什么都不做. 当语法上需要一个语句, 但程序不要动作时, 就可以使用它. 例如:

while True:

pass # 忙等待键盘中断(Ctrl+C)

### 3.2 函数定义

#### 3.2.1 def

关键字def 引入了一个函式定义. 后面必须跟上函式名和在圆括号里的参数序列. 函式体从一行开始, 并且一定要缩进。

函式体的第一个语句可以是字串; 这个字串就是函式的文档字符串, 或称为docstring. (可以在文档字串一节找到更多信息) 有很多能将文档字串自动转换为在线或可打印文档的工具, 或让用户在代码中交互地浏览它的工具; 在代码里加上文档字符串是一个好的实践, 因此, 请养成这个习惯。

执行函式，会引入新的符号表(symbol table) 用于该函式的局部变量.更精确地说, 所有在函式中被赋值的变量和值都将存储在局部符号表中。

在函式中的尽管全局变量可以引用, 但是不可直接赋值(除非用global 语句进行声明).

函式的实参在它被调用时被引入到这个函式的局部变量表；因此，参数是按值传递的(值总是对象的一个引用，而不是对象本身的值)。

当一个函式调用另一个时, 对应这次调用, 一个新的局部符号表就会被创建.

#### 3.2.2 默认参数

例如：

def ask\_ok(prompt , retries=4, complaint='Yes or no ,please!'):

**重要警告: 默认参数的值只会被求一次值. 这使得当默认参数的值是可变对象时会有所不同，如列表, 字典, 或大多类的对象时。**

例如, 下面的函式在随后的调用中会累积参数值:

|  |
| --- |
| def f(a, L=[]):  L.append(a)  return L  print(f(1))  print(f(2))  print(f(3))  如上方式调用，L的内容会不断增加。 |
| 若要默认参数不会受影响，解决如下：  def f(a, L=None):  if L is None:  L = []  L.append(a)  return L |

#### 3.2.3 关键字参数

函式也可以通过keyword= value 形式的关键字参数来调用。

|  |
| --- |
| def parrot(voltage , state='a stiff ', action='voom', type='Norwegian Blue'):  parrot(1000)  parrot(action = 'VOOOOOM ', voltage = 1000000)  parrot('a thousand ', state = 'pushing up the daisies ') #此时voltage=’a thousand’  parrot('a million ', 'bereft of life', 'jump') #此时，type采用默认值  在函式调用时, 关键字参数必须跟在位置参数之后. 所有的关键字参数都必须与函式接受的形式参数匹配 |

\*name和\*\*name参数形式：

当最后一个形式参数的形式为\*\*name 时, 则除去其他的形参的值，它将以字典(参阅映射类型——字典) 的形式包含所有剩余关键字参数。 这种调用可以与具有\*name 形式的形式参数联合使用, 这种形式参数接受所有超出函式接受范围的位置参数.(\*name 必须在\*\*name之前使用)。

例如, 如果我们像这样定义一个函式:

|  |
| --- |
| **def** cheeseshop(kind , \*arguments , \*\*keywords):  **print**(kind )  **for** arg **in** arguments:  **print**(arg)  keys = sorted(keywords.keys())  **for** kw **in** keys:  **print**(kw , ":", keywords[kw])  cheeseshop(10, 1,2,3,shopkeeper="Michael Palin",client="John Cleese",sketch="Cheese Shop Sketch")  如上调用，kind=10，\*arguments代表1，2，3，\*\*keywords代表后面三个关键字参数  注意, 关键字参数名的列表是通过之前对字典keys() 进行排序操作而创建的; 如果不这样做, 参数打印的顺序是不确定的. |

#### 3.2.4 任意参数表

指定函式能够在调用时接受任意数量的参数. 这些参数会被包装进一个元组(参看元组和序列). 在变长参数之前, 可以使用任意多个正常参数:

**def** write\_multiple\_items(file , separator , \*args):

一般地, 这种variadic 参数必须在形参列表的末尾, 因为它们将接收传递给函式的所有剩余输入参数. 任何出现在\*arg 之后的形式参数只能是关键字参数, 这意味着它们只能使用关键字参数的方式接收传值, 而不能使用位置参数.

**def** concat(\*args , sep="/"):

#### 3.2.5 释放参数列表

当**参数存在于一个既存的列表、元组甚至字典之中**, 但却需要解包以若干位置参数的形式被函数调用。

**可以利用\*释放列表或元组中的元素；**

**利用\*\*释放字典中的参数**

|  |
| --- |
| args = [3, 6]  list(range(\*args)) # 通过解包列表参数调用  [3, 4, 5]  **def** parrot(voltage , state='a stiff ', action='voom'):  print(voltage)  print(state+action)  d = {"voltage": "four million", "state": "bleed in demised", "action": "VOOM"}  parrot(\*\*d) |

#### 3.2.6 文档字符串

这里介绍一些文档字串有关内容和格式的约定.

第一行总应当是对该对象的目的进行简述. 这行应当以一个大写字母开始, 并以句号结束。

如果这个文档字符串不只一行, 那么第二行应当为空, 以能从视觉上分隔概述和其它部分. 接下来的行应当为一个或更多段来描述该对象的调用条件, 它的边界效应等等.

|  |
| --- |
| **def** my\_function():  """Do nothing , but document it.  No , really , it doesn 't do anything.  """  **pass**  **print**(my\_function.\_\_doc\_\_) |

## 四、模块

用户把定义存放在文件里, 同时又能在脚本或交互式环境下方便的使用它们. 这样的文件称为模块模块就是包含python定义和语句的文件。

一个模块中的定义可以导入(import) 到另一个模块或主模块(主模块是执行脚本的最上层或计算模式下的一组可访问变量的集合).

在一个模块中，模块的名字（一个字符串）可以通过全局变量 \_ \_name\_ \_得到。

|  |
| --- |
| # fibo.py  def fib(n): # 打印小于n 的Fibonacci 数列  a, b = 0, 1  while b < n:  print(b, end=' ')  a, b = b, a+b  print() |
| >>> fibo.fib(1000)  1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987  >>> fibo.\_\_name\_\_  'fibo' |

### 4.1 深入模块

每个模块有其私有的符号表, 由模块内部定义的函式当成全局符号表来使用. 因此, 模块的作者可以在模块中放胆使用全局变量而无需担心与用户的全局变量发生冲突。

另一方面, 当你确实明白你在做什么的时候, 你可以通过modname.itemname 形式来访问模块的全局变量。

#### 4.1.1 import用法

|  |
| --- |
| **import numpy as np**  **#通过np.的方式就可以调用numpy中的代码**  **from fibo import fib**  **#可以直接调用fibo中的fib函数**  **from fibo import \***  **#一次性导入模块中所有名字定义，这样可以导入除下划线开头（\_）的所有名字，一般不使用这个窍门，因为会导入一些未知的名字到解释器，可能意外重载一些已经定义的东西**  **#这种方式一般在交互式会话中，会节省键入**  注意, 当使用**from package import item** 时, 这个项即可以是这个包的一个子模块(或子包), 也可以是其它的定义在这个包里的名字, 如函式, 类或变量. import 语句首先测试这个项是否在包里定义; 如果没有, 就假设它是一个模块并试图载入它. 如果寻找它失败, 就会抛出一个ImportError. |

#### 4.1.2 if \_\_name\_\_ == ‘\_\_main\_\_’ （即可以提供接口，也可以当做单独测试）

一句话总结：

**当按照.py文件直接运行时，if \_\_name\_\_ == ‘\_\_main\_\_’之下代码块将被运行；**

**当以模块形式被导入时，if \_\_name\_\_ == ‘\_\_main\_\_’之下代码块将不被运行；**

**（1）程序入口（为什么会有此定义的原因）**

对于很多编程语言来说，程序都必须要有一个入口。而Python则不同，它属于脚本语言，不像编译型语言那样先将程序编译成二进制再运行，而是动态的逐行解释运行。也就是从脚本第一行开始运行，没有统一的入口。

一个Python源码文件（.py）除了可以被直接运行外，还可以作为模块（也就是库），被其他.py文件导入。**不管是直接运行还是被导入，.py文件的最顶层代码都会被运行（Python用缩进来区分代码层次），而当一个.py文件作为模块被导入时，我们可能不希望一部分代码被运行。**

**（2）\_\_name\_\_**

\_\_name\_\_是内置变量，可用于反映一个包的结构。假设我们有一个包a，包的结构如下：

a

├── b

│ ├── c.py

│ └── \_\_init\_\_.py

└── \_\_init\_\_.py

\_\_name\_\_是内置变量，可用于表示当前模块的名字。我们直接运行一个.py文件（模块）

|  |
| --- |
| #a.py脚本内容  Print(\_\_name\_\_)  运行脚本  Python a.py  输出为  \_\_main\_\_ |

由此我们可知：如果一个.py文件（模块）被直接运行时，则其没有包结构，其\_\_name\_\_值为\_\_main\_\_，即模块名为\_\_main\_\_

1. **\_\_main\_\_.py文件与python -m**

python xxx.py，直接运行xxx.py文件

python -m xxx.py，把xxx.py当做模块运行

* 当加上-m参数时，Python会把当前工作目录添加到sys.path中；而不加-m时，Python则会把脚本所在目录添加到sys.path中。
* 当加上-m参数时，Python会先将模块或者包导入，然后再执行。
* \_\_main\_\_.py文件是一个包或者目录的入口程序。不管是用python package还是用python -m package运行，\_\_main\_\_.py文件总是被执行。

#### 4.1.3 模块搜索路径

对于导入的模块，解释器会先在当前目录下查找该名称的文件；若未找到，则解释器会在变量sys.path给定的目录列表中寻找该模块。

Sys.path的路径从如下位置初始化：

* + 包含输入脚本命令的目录，或者当前目录
  + 环境变量PYTHONPATH的目录名列表
  + 安装时的默认目录

Python程序可以修改sys.path，被运行的脚本所在的目录，排在标准库路径之前被搜索，这意味着如果脚本所在目录中有与标准库相同名称的模块，则会加载这个模块而不是标准库中的模块。

变量sys.path 是一个字符串列表, 它为解释器指定了模块的搜索路径. 它通过环境变量PATHONPATH 初始化为一个默认路径, 当没有设置PYTHONPATH 时, 就使用内建默认值来初始化. 你可以通过标准列表操作来修订之：

import sys

sys.path.append('path ')

**（1）已编译的python文件**

为了减少使用了大量标准模块的小程序的启动时间, 如果.py 所在目录下中有名为.pyc 的文件, 解释器就会优先导入spam 模块的这一“已编译字节” 版本文件. 用来创建.pyc 的.py 的版本修改时间被记录在.pyc 中, 如果不匹配的话, .pyc 文件就会被忽略.

给专家的小技巧:

• 当使用-O 参数来调用Python 解释器时, python 会对代码进行优化,并存入在.pyo 文件里.

• 程序从.pyc 或.pyo 文件里读取时, 并不会比它从.py 文件中读取会有更快的执行速度; 唯一提高的是载入速度.

• 对于同一个模块, 可以只包含spam.pyc (或者spam.pyo 当使用-O时) 文件而无需spam.py 文件. 使用这种形式可用以发布Python 代码库, 并使得反编译工程有一定的难度.

### 4.2 标准模块

#### 4.2.1 dir()函式

内建函式dir() 用于找出一个模块里定义了那些名字. 它返回一个有序字串列表:

|  |
| --- |
| >>> import fibo , sys  >>> dir(fibo)  ['\_\_name\_\_ ', 'fib', 'fib2'] |

不给参数时，dir()就罗列出当前定义的所有名字。注意, 它列举出了所有类型的名字: 变量, 模块, 函式, 等等.

#### 4.2.2 包

**包是一种Python 模块命名空间的组织方法,** 通过使用“带点号的模块名”. 例如, 模块名A.B 指定了一个名为A 的包里的一个名为B 的子模块. 就像模块的使用使不同模块的作者避免担心其它全局变量的名字, 而带点号的模块使得多模块包, 例如NumPy 或Python 图像库, 的作者避免担心其它模块名.

1. **例子：**

|  |
| --- |
| 假设你想设计一个模块集(一个“包”), 用于统一声音文件和声音数据的处理. 有许多不同的声音格式(通常通过它们的后缀来辨认, 例如: .wave,.aiff, .au), 因此你可能需要创建和维护一个不断增长的模块集, 用以各种各样的文件格式间的转换. 还有许多你想对声音数据执行的不同操作(例如混频, 增加回音, 应用一个均衡器功能, 创建人造的立体声效果), 因此, 你将额外的**写一个永无止尽的模块流**来执行这些操作. 这是你的包的一个可能的结构:  sound/ 顶级包  \_\_init\_\_.py 初始化这个声音包  formats/ 文件格式转换子包  \_\_init\_\_.py  wavread.py  wavwrite.py  aiffread.py  aiffwrite.py  auread.py  auwrite.py  ...  effects/ 音效子包  \_\_init\_\_.py  echo.py  surround.py  reverse.py  ...  filters/ 过滤器子包  \_\_init\_\_.py  equalizer.py  vocoder.py  karaoke.py  当导入这个包时, Python 搜索sys.path 上的目录以寻找这个包的子目录。  需要\_\_init\_\_.py 文件来使得Python 知道这个目录包含了包; 这用来预防名字为一个通用名字。  包的用户可以包里的单独的模块, 例如:  import sound.effects.echo  导入子模块的一个替代方法是:  from sound.effects import echo  另一个变种是直接导入想要的函式或变量:  from sound.effects.echo import echofilter |

**（2）内部包参考和多目录包**

当包被构造到子包时(如例子中的sound 包), 你可以独立地导入来获取兄弟包的子模块的引用. 例如, 如果模块sound.filters.vocoder 需要使用sound.effects 包下的echo 模块, 就可以使用from sound.effects import echo.

包支持额外一个特殊的属性, \_\_path\_\_ . 它在文件中的代码执行之前,被初始化为一个列表, 它包含保存在这个包的\_\_init\_\_.py 文件中目录名.

虽然这个特性不经常需要, 但它可以用于扩展在一个包里发现的模块的集合.

## 五、输入和输出

常见的输出值的格式有：

1. 表达式语句；
2. print()函数；
3. 文件对象的write()方法
4. 标准输出文件可以用sys.stdout引用；

小技巧：

print()函数：

关键词end可以用来避免输出后的回车, 或者以一个不同的字符串结束输出:

Print(‘hello ‘, end=’!’)

### 5.1 控制输出格式

Python 有多种方式将任何值转为字符串: 将它传给**repr() 或str() 函数.**

str() 函数意味着返回一个用户易读的表达形式, 而repr() 则意味着产生一个解释器易读的表达形式(或者如果没有这样的语法会给出SyntaxError ).

对于那些没有特殊表达的对象, str() 将会与repr() 返回相同的值. 很多的值, 如数字或一些如列表和字典那样的结构, 使用这两个函数的结果完全一致. 字符串与浮点型则有两种不同的表达.

#### 5.1.1 string操作

一种是使用字符串切片和连接操作, 来实现你所想象的外观. 标准模块string 包含了一些有用的操作, 用以填充字符串至某一给定的宽度;比如：

rjust() 将字符串靠右, 并在左边填充空格. 还有类似的方法

ljust()

center()

zfill() 它会在数字的左边填充0. 它知道正负号:

|  |
| --- |
| >>> for x in range(1, 4):  ... print(repr(x).rjust(2), repr(x\*x).rjust(3), end=' ')  ... # Note use of 'end' on previous line 注意前一行'end'  的使用  ... print(repr(x\*x\*x).rjust(4))  ...  1 1 1  2 4 8  3 9 27  4 16 64  >>> '12'.zfill(5)  '00012 '  >>> '-3.14'.zfill(7)  '-003.14' |

#### 5.1.2 str.format()方法

第二种方式是使用str.format() 方法.

|  |
| --- |
| >>> print('We.are.the.{}.who.say."{}!"'.format('knights ', 'Ni'))  We are the knights who say "Ni!"  **1）括号及其里面的字符(称作format field) 将会被format() 中的参数替换. 在括号中的数字用于指向传入对象在format() 中的位置.**  >>> print('{0} and {1}'.format('spam', 'eggs'))  spam and eggs  >>> print('{1} and {0}'.format('spam', 'eggs'))  eggs and spam  **2）如果在format() 中使用了关键字参数, 那么它们的值会指向使用该名字的参数.**  print('This {food} is {adjective}.'.format(  ... food='spam', adjective='absolutely horrible '))  This spam is absolutely horrible.  >>> print('The story of {0}, {1}, and {other}.'.format('Bill'  , 'Manfred ', other='Georg '))  The story of Bill , Manfred , and Georg.  **3）'!a' (使用ascii()), '!s' (使用str()) 和'!r' (使用repr()) 可以用于在格式化某个值之前对其进行转化:**   1. **可选项':' 和格式标识符可以跟着field name. 这就允许对值进行更好的格式化. 下面的例子将Pi 保留到小数点后三位.**   >>> import math  >>> print('The value of PI is approximately {0:.3f}.'.format(  math.pi))  The value of PI is approximately 3.142.  >>> for x in range(1, 4):  ... print('{0:2d} {1:3d} {2:4d}'.format(x, x\*x, x\*x\*x))  1 1 1  2 4 8  3 9 27  4 16 64  **5）在':' 后传入一个整数, 可以保证该域至少有这么多的宽度. 用于美化表格时很有用.**  >>> table = {'Sjoerd ': 4127 , 'Jack': 4098 , 'Dcab': 7678}  >>> for name , phone in table.items():  ... print('{0:10} ==> {1:10d}'.format(name , phone))  ...  Jack ==> 4098  Dcab ==> 7678  Sjoerd ==> 4127  **6）传入一个字典, 然后使用方括号'[]' 来访问键值:**  >>> table = {'Sjoerd ': 4127 , 'Jack': 4098 , 'Dcab': 8637678}  >>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd ]:d}; ' 'Dcab: {0[Dcab]:d}'.format(table))  Jack: 4098; Sjoerd: 4127; Dcab: 8637678  **7）也可以通过在table 变量前使用‘\*\*’ 来实现相同的功能**.  >>> table = {'Sjoerd ': 4127 , 'Jack': 4098 , 'Dcab': 8637678}  >>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(\*\*table))  Jack: 4098; Sjoerd: 4127; Dcab: 8637678 |

#### 5.1.3 旧式字符串格式化--- %操作符

% 操作符也可以实现字符串格式化. 它将左边的参数作为类似sprintf() 式的格式化字符串, 而将右边的代入, 然后返回格式化后的字符串。

因为str.format() 很新, 大多数的Python 代码仍然使用% 操作符. 但是因为这种旧式的格式化最终会从该语言中移除, 应该更多的使用str.format().

|  |
| --- |
| >>> import math  >>> print('The.value.of.PI.is.approximately.%5.3f.' % math.pi) |

### 5.2 读和写文件

open() 将会返回一个文件对象, 并且一般使用两个参数进行使用:

**f=open(filename, mode).**

第一个参数filename是包含文件名的字符串.

第二个参数mode是另一个字符串, 包含描述文件如何使用的字符. 是可选参数

mode 可以是：

'r' 只文件只读 是默认值.

'w' 只用于写(如果存在同名文件则将被删除)

'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾

'r+' 同时用于读写.

#### 5.2.1 文本格式和二进制格式

文件以text mode 打开, 这就意味着, 从文件中读写的字符串, 是以一种特定的编码进行编码(**默认的是UTF-8).**

**追加到mode 后的'b' , 将意味着以binary mode 打开文件:** 现在的数据是以字节对象的形式进行读写. 这个模式应该用于那些不包含文本的文件.

**在文本模式下(text mode), 默认是将特定平台的行末标识符( Unix 下为\n, Windows 下为\r\n ) 在读时转为\n 而写时将\n 转为特定平台的标识符.**

这种隐藏的行为对于文本文件是没有问题的, 但是**对于二进制数据像JPEG 或EXE 是会出问题的. 在使用这些文件时请小心使用二进制模式.**

#### 5.2.2 文件的读写

**（1）f.read()**

为了读取一个文件的内容, 调用f.read(size), 这将读取一定数目的数据, 然后作为字符串或字节对象返回. size 是一个可选的数字类型的参数. 当size 被忽略了或者为负, 那么该文件的所有内容都将被读取并且返回; 如果文件比你的内存大两倍, 那么就会成为你的问题了. 否则, 最多size字节将被读取并返回. 如果到达了文件的末尾, f.read() 将会返回一个空字符串('').

|  |
| --- |
| >>> f.read()  'This is the entire file.\n'  >>> f.read()  '' |

**（2）f.readline()**

f.readline() 会从文件中读取单独的一行; 在每个字符串的末尾都会留下换行符(\n), 除非是该文件的最后一行并且没有以换行符结束, 这个字符才会被忽略. 这就使结果很明确; f.readline() 如果返回一个空字符串,那么文件已到底了, 而如果是以'\n' 表示, 那么就是只包行一个新行.

|  |
| --- |
| >>> f.readline()  'This is the first line of the file.\n'  >>> f.readline()  'Second line of the file\n'  >>> f.readline()  '' |

**（3）f.readlines()**

f.readlines() 将返回该文件中包含的所有行. 如果给定一个可选参数sizehint, 它就读取这么多字节, 并且将这些字节按行分割. 这经常用于允许按行读取一个大文件, 但是不需要载入全部的文件时非常有用. 只会返回完整的行.

|  |
| --- |
| >>> f.readlines()  ['This.is.the.first.line.of.the.file.\n', 'Second.line.of.the .file\n'] |

**（4）f.write(string)**

f.write(string) 将string 写入到文件中, 然后返回写入的字符数.

|  |
| --- |
| >>> f.write('This is a test\n')  15  如果要写入一些不是字符串的东西, 那么将需要先进行转换:  >>> value = ('the answer ', 42)  >>> s = str(value)  >>> f.write(s)  18 |

**（5）f.tell() f.seek()**

f.tell() 返回文件对象当前所处的位置, 它是从文件开头开始算起的字节数.

要改变文件当前的位置, 使用f.seek(offset, from\_what). 这个位置是通过将当前位置加上offset 所得. from\_what 的值, 如果是0 表示开头, 如果是1 表示当前位置, 2 表示文件的结尾. from\_what 的默认为0, 即从开头开始.

**（6）f.close()**

当你处理完一个文件后, 调用f.close() 会关闭它, 并释放系统的资源. 在调用完f.close() 之后, 尝试使用那个文件对象是会失败的.

**（7）结合with 处理文件**

当处理一个文件对象时, 使用with 关键字是非常好的方式. 在结束后, 它会帮你正确的关闭文件, 即使发生了异常. 而且写起来也比try -finally 语句块要简短:

#### 5.3 pickle模块

在文件中, 字符串可以很方便的读取写入. 数字可能稍微麻烦一些, 因为read() 方法只返回字符串, 我们还需要将其传给int() 这样的函数, 使其将如'123' 的字符串转为数字123. 但是, 如果要保存更复杂的数据类型, 如列表, 字典, 或者类的实例, 那么就会更复杂了.

**这个模块可以将几乎任何的Python 对象(甚至是Python 的代码), 转换为字符串表示; 这个过程称为pickling. 而要从里面重新构造回原来的对象, 则称为unpickling. 在pickling 和unpickling 之间, 表示这些对象的字符串表示, 可以存于一个文件, 也可以通过网络在远程机器间传输.**

如果你有一个对象x, 和一个已经打开并用于写的文件对象f, pickle这个对象最简单的方式就是使用:

**pickle.dump(x, f)**

有了pickle 这个对象, 就能对f 以读取的形式打开:

**x = pickle.load(f)**

pickle 是Python 中保存及重用对象的标准方式; 标准的属于称为persistent 对象(即持久化对象). 因为pickle 被广泛使用, 很多写Python 扩展的作者都会确保, 如矩阵这样的数据类型能被合理的pickle 和unpickle.

## 六、错误和异常

### 6.1 处理异常

写程序来处理异常是可能的. 看看下面的例子, 它请求用户输入一个合法的整数, 但是也允许用户来中断程序(使用Control-C 或任何操作系统支持的); 注意, 用户生成的中断是通过产生异常KeyboardInterrupt:

|  |
| --- |
| >>> while True:  ... try:  ... x = int(input("Please enter a number: "))  ... break  ... except ValueError:  ... print("Oops! That was no valid number. Try again ...")  ... |

#### 6.1.1 try语句

try 语句像下面这样使用.

• 首先, try clause (在try 和except 之间的语句) 将被执行.

• 如果没有异常发生, except clause 将被跳过, try 语句就算执行完了.

• 如果在try 语句执行时, 出现了一个异常, 该语句的剩下部分将被跳过. 然后如果它的类型匹配到了except 后面的异常名, 那么该异常的语句将被执行, 而执行完后会运行try 后面的问题.

• 如果一个异常发生时并没有匹配到except 语句中的异常名, 那么它就被传到try 语句外面; 如果没有处理, 那么它就是unhandled exception并且将会像前面那样给出一个消息然后执行.

一个try 语句可以有多于一条的except 语句, 用以指定不同的异常.但至多只有一个会被执行. Handler 仅仅处理在相应try 语句中的异常, 而不是在同一try 语句中的其他Handler. 一个异常的语句可以同时包括多个异常名, 但需要用括号括起来, 比如:

... except (RuntimeError , TypeError , NameError):

... pass

最后的异常段可以忽略异常的名字, 用以处理其他的情况. 使用这个时需要特别注意, 因为它很容易屏蔽了程序中的错误!

|  |
| --- |
| import sys  try:  f = open('myfile.txt')  s = f.readline()  i = int(s.strip())  except IOError as err:  print("I/O error: {0}".format(err))  except ValueError:  print("Could not convert data to an integer.")  except:  print("Unexpected error:", sys.exc\_info()[0])  raise |

try ... except 语句可以有一个可选的else 语句, 在这里, 必须要放在所有except 语句后面. 它常用于没有产生异常时必须执行的语句. 例如:

|  |
| --- |
| for arg in sys.argv[1:]:  try:  f = open(arg , 'r')  except IOError:  print('cannot open', arg)  else:  print(arg , 'has', len(f.readlines()), 'lines ')  f.close() |

使用else 比额外的添加代码到try 中要好, 因为这样可以避免偶然的捕获一个异常, 但却不是由于我们保护的代码所抛出的.

### 6.2 抛出异常

raise 语句允许程序员强制一个特定的异常的发生.

给raise 的唯一参数表示产生的异常. 这必须是一个异常实例或类(派生自Exception 的类).

如果你需要决定产生一个异常, 但是不准备处理它, 那么一个简单的方式就是, 重新抛出异常:

|  |
| --- |
| >>> try:  ... raise NameError('HiThere ')  ... except NameError:  ... print('An exception flew by!')  ... raise |

### 6.3 自定义异常与定义清理动作

## 七、类

同别的编程语言相比, Python 的类机制中增加了少量新的语法和语义.它是C++ 的类机制和Modula-3 的类机制的混合体.

**Python 类提供了面向对象编程的所有基本特征: 允许多继承的类继承机制, 派生类可以重写它父类的任何方法, 一个方法可以调用父类中重名的方法. 对象可以包含任意数量和类型的数据成员.**

**作为模块, 类也拥有Python 的动态特征: 他们可以被动态创建, 并且可以在创建之后被修改.**

从C++ 术语上讲, Python 类的成员(包括数据成员) 通常都是public的(例外见下私有变量(page 95)), 并且所有的成员函数都是virtual 的.

和Modula-3 中一样, Python 中没有关联对象成员和方法的隐式表达: 所有方法函数在声明时显式地将第一个参数表示为对象, 这个参数的值在方法被调用时隐式赋值.

同Smalltalk 类似, Python 类本身就是对象. 这就提供了导入和重命名的语义.

与C++ 和Modula-3 不同的是, Python 的内置类型可以被当做基类来让使用者扩展. 另外, 像C++ 一样, 大多数有特殊语法的内置操作符(算数运算符, 下标操作符等等) 在类的实例中都可以重定义.

### 7.1 名称和对象的讨论

对象都具有个别性, 多个名称(在多个作用域中) 可以被绑定到同一个对象上. 这就是其他语言中所谓的别名.

通常第一次接触Python 可能不会意识到这一点, 而且在处理不变的基本类型(数值, 字符串, 元组) 时这一点可能会被安全的忽略. 但是, 在涉及到可变对象如lists, dictionaries, 以及大多数其他类型时, 别名可能会在Python 代码的语义上起到惊人的效果.

别名通常对编程有益处, 因为别名在某些方面表现得像指针. 比如, 由于在实现的时候传递的是指针, 所以传递一个对象的开销很小; 又比如将对象作为参数传递给一个函数来对它进行修改, 调用者将会看到对象的变化。

### 7.2 python的作用域和命名空间

#### 7.2.1 命名空间

**命名空间是从名称到对象的映射.** 大多数命名空间现在的实现就如同Python 的字典, 但通常这一点并不明显(除了在性能上), 而且它有可能在将来发生改变.

我用了属性这个词来称呼任何点后面跟的名称——比如,在表达式z.real 中, real 就是对象z 的属性. 更直接的说, **对模块中名称的引用就是属性引用:** 在表达式modname.funcname 中, modname 是模块对象而funcname 是它的一个属性. 在这种情况下**模块的属性和它里面所定义的全局名称之间就刚好有一个直接的映射关系: 他们共享同一个命名空间!**

**属性可以是只读的或可写的**. 在后一种情况下, 给属性赋值才是可能的. 模块属性是可写的: 你可以写modname.the\_answer = 42. 可以利用:keyword:del 语句来删除可写属性. 例如, del modname.the\_answer 将从名为modname 的模块中移除属性the\_answer.

**命名空间们是在不同时刻创建的, 并且有着不同的生命期.**

包含内置名称的命名空间是在Python 解释器启动时创建的, 而且它永远不被删除. 一个模块的全局命名空间在模块的定义被读取的时候创建; 通常情况下, 模块的命名空间一直持续到解释器退出时. 被最高级别的解释器调用的语句, 不论是从脚本还是从交互读取的, 都被认为是一个名叫\_\_main\_\_ 的模块的一部分, 所以它们有自己的全局命名空间. (内置名称实际上也存在于一个模块中; 这个模块叫builtins.)

函数的局部命名空间在函数调用时被创建, 在函数返回时或者发生异常而终止时被删除。当然, 递归调用会有它们自己的局部命名空间.

**尽管作用域是静态的决定的, 它们使用时却是动态的. 在执行时的任何时刻, 至少有三个嵌套的作用域其命名空间可以直接访问:**

• 最内层的作用域, 首先被搜索, 包含局部变量名

• 任意函数的作用域, 它从最接近的作用域开始搜索, 包括非局部的, 但也是非全局的名字

• 紧邻最后的作用域包含了当前模块的全局变量

• 最外层的作用域(最后搜索) 是包含内置名字的命名空间

**如果一个名字在全局声明, 那么所有的引用和赋值都直接到这个模块的全局名中. 为了在最内部作用域中重新绑定变量, nonlocal 语句就可以使用了; 如果没有声明nonlocal , 那些变量只是只读(尝试给这样的变量赋值, 只是会简单的创建一个新的局部变量, 而外部的并没有什么改变) 重新绑定.**

一般来说, 局部作用域引用当前函数的局部变量名. 在函数外部, 局部变量 引用 和全局作用域相同的命名空间。

Python 的一个怪事就是– 如果global 语句没有起效果——赋值总是会使用最里层作用域的值. **赋值并没有拷贝数据——它们仅仅是绑定名字到对象上. 删除也是如此: del x 移除了x 从局部作用域的绑定.** 事实上,所有操作引入新的名字都使用局部作用域: **特别的, import 语句, 和函数定义都将模块或函数绑定到了当前作用域.**

**global 语句可以用于指示, 在全局作用域中的变量可以在这里重新绑定;**

**nonlocal 则表示在一个闭合的作用域中的变量可以在此处绑定.**

**（1）域和命名空间的例子**

|  |
| --- |
| def scope\_test():  def do\_local():  spam = "local spam" #局部的赋值（默认），没有改变scope\_test中绑定的spam  def do\_nonlocal():  nonlocal spam  spam = "nonlocal spam" #nonlocal的赋值，改变了scope\_test中的spam  def do\_global():  global spam  spam = "global spam" #global的赋值，则改变了模块级别的绑定    spam = "test spam"  do\_local() #局部赋值，不改变scope\_test中的绑定，只对do\_local有影响  print("After local assignment:", spam)  do\_nonlocal() #改变了scope\_test中的绑定，但未影响模块级别的绑定  print("After nonlocal assignment:", spam)  do\_global() #改变了模块级别的绑定，但并未改变scope\_tes中的spam  print("After global assignment:", spam)  scope\_test()  print("In global scope:", spam)  输出结果为  After local assignment: test spam  After nonlocal assignment: nonlocal spam  After global assignment: nonlocal spam  In global scope: global spam |

### 7.3 类的概要

类引入了一些新的语法, 三种新的对象类型, 和一些新的语义。

#### 7.3.1 类定义的语法

|  |
| --- |
| class ClassName:  <statement-1>  ...  <statement-N> |

类的定义, 和函数定义(def 语句) 一样必须在使用它们前执行. (你可以将一个类定义放置于if 语句的分支中, 或一个函数中.)

事实上, 类定义内部的语句一般是函数的定义, 但其他的语句也是允许的, 而且还很有用。

当进入一个类定义, 新的命名空间就被创建了, 这一般作为局部的作用域——因此, 所有的局部变量都在这个新的作用域中. 特别是, 函数定义会绑定。

当离开一个类定义后, 一个class object 就被创建. 通过类的定义, 就将这个命名空间包装了起来;

#### 7.3.2 类对象

类对象支持两种操作: 属性引用和实例化。

**（1）属性引用**

属性引用使用的语法和Python 中所有的属性引用一样. 合法的属性名是那些在类的命名空间中定义的名字。

|  |
| --- |
| class MyClass:  """A simple example class """  i = 12345  def f(self):  return 'hello world '  那么, **MyClass.i 和MyClass.f**就是合法的属性引用, 分别返回一个整数和一个函数对象.  **类属性也可以被指定**, 所以你可以给MyClass.i 赋值以改变其数值  **. \_\_doc\_\_ 也是一个合法的属性**, 返回属于这个类的docstring: "A simple example class". |

**（2）实例化**

类的实例化使用函数的形式. 只要当作一个无参的函数然后返回一个类的实例就可以了。比如：

x = MyClass() #创建了一个新的实例, 并且将其指定给局部变量x.

实例化的操作(“调用” 一个类对象) 创建了空的对象. 在创建实例时,**很多类可能都需要有特定的初始状态. 所以一个类可以定义一个特殊的方法, 称为\_\_init\_\_(),** 像这样:

|  |
| --- |
| def \_\_init\_\_(self):  self.data = []  >>> class Complex:  ... def \_ \_init\_ \_(self , realpart , imagpart):  ... self.r = realpart  ... self.i = imagpart  X=Complex(3.0,-4.5) |

**当一个类定义了\_\_init\_\_() 方法, 类在实例化时会自动调用\_\_init\_\_()方法, 用于创建新的类实例。**

#### 7.3.3 实例对象

实例对象唯一能理解的操作就是属性引用. 有两种合法的属性, **数据属性和方法.**

**数据属性**，相当于c++中的数据成员。但数据属性不需要声明，像局部变量，当第一次指定时，就会被引入。

|  |
| --- |
| x=Myclass()  x.counter = 1 #第一次指定时被引入  while x.counter < 10:  x.counter = x.counter \* 2  print(x.counter)  del x.counter #删除该数据属性 |

**方法**，**也可以理解为一个对象的函数。**合法的方法名依赖于实例的类。**x.f 和MyClass.f 并不一样——它是一个method object, 而不是function object.**

**通常, 方法的第一个参数称为self。**

**1）函数定义可以任意位置**

**作为类属性的任何函数对象, 定义了一个方法用于那个类的实例. 函数是否在一个类体中其实并不重要: 指定一个函数对象给类中的局部变量也是可以的.** 例如:

|  |
| --- |
| # Function defined outside the class  def f1(self , x, y):  return min(x, x+y)  class C:  f = f1  def g(self):  return 'hello world '  h = g |

**2）方法可以通过使用self 参数调用其他的方法**

|  |
| --- |
| class Bag:  def \_\_init\_\_(self):  self.data = []  def add(self , x):  self.data.append(x)  def addtwice(self , x):  self.add(x)  self.add(x) |

方法可以引用全局变量, 就像普通函数中那样. 与这个方法相关的全局作用域, 是包含那个类定义的模块. (类本身永远不会作为全局作用域使用.)

如果的确需要在方法中使用全局数据, 那么需要合法的使用: 首先一件事, 被导入全局作用域的函数和模块可以被方法使用, 就如定义在里面的函数和类一样. 通常来说, 定义在全局作用域中, 包含方法的类是它自己本身,并且在后面我们会知道为何方法应该引用自己的类.

每个值都是一个对象, 所以对于class (或称为它的type) 也是这样. 它存于object.\_\_class\_\_.

**（1）method object方法对象和function object 函数对象**

关于方法, 特殊的东西就是, 对象作为参数传递给了函数的第一个参数. 在我们的例子中，**x.f() 是严格等价于MyClass.f(x)。（注意一个传了参数，一个没有传参数）**

在多数情况下, 调用一个方法(有个n 个参数), 和调用相应的函数(也有那n 个参数, 但是再额外加入一个使用该方法的对象), 是等价的.

**（2）数据属性和方法属性的命名注意**

如果数据属性和方法出现同名，很可能导致难调的bug。所以一般命名有如下约定：

方法名称使用要大写，或者方法名称使用动词。

数据属性请增加特殊的前缀，比如下划线\_，或者数据成员使用名词。

**数据属性可以被该类的方法或者普通的用户(“客户”) 引用. 换句话说,类是不能实现完全的抽象数据类型。**

**客户需要小心地使用数据属性——客户会弄乱被方法控制的不变量，通过使用它们自己的方法属性. 注意用户可以增加它们自己的数据到实例对象上, 而没有检查有没有影响方法的有效性, 只要避免名字冲突。**

### 7.4 继承

#### 7.4.1 派生类定义和使用

派生类的定义如下:

|  |
| --- |
| class DerivedClassName(BaseClassName):  <statement-1>  ...  <statement-N>  比如基类在另一个模块中：  class DerivedClassName(modname.BaseClassName):  ... |

派生类就可以像基类一样使用. 当一个类被构建, 那么它就会记下基类. 这是用于解决属性引用的问题: 当一个属性在这个类中没有被找到, 那么就会去基类中寻找. 然后搜索就会递归, 因为如果基类本身也是从其他的派生.

**（1）实例化派生类**

实例化一个派生类没有什么特别: DerivedClassName() 会创建这个类的新实例.

**（2）方法的引用**

相应的类的属性会被搜寻, 如果需要回去搜寻基类, 如果返回一个函数对象, 那么这个引用就是合法的.

派生类会覆写基类的方法. 因为当调用同样的对象的其他方法时方法并没有什么特别的, 基类的方法会因为先调用派生类的方法而被覆写. (对于C++ 程序员: 所有的方法在Python 中都是vitual 的.)

一个在派生类中覆写的方法可能需要基类的方法. 最简单的方式就是直接调用基类的方法: 调用BaseClassName.methodname(self,arguments). 这对于可续来说也是很方便的. (这仅在BaseClassName 可访问时才有效.)

#### 7.4.2 多重继承

|  |
| --- |
| class DerivedClassName(Base1 , Base2 , Base3):  <statement-1>  ...  <statement-N> |

对于大多数目的, 在最简单的情况下, 你可以将属性搜寻的方式是, 从下至上, 从左到右, 在继承体系中, 同样的类只会被搜寻一次. 如果一个属性在DerivedClassName 中没有被找到, 它就会搜寻Base1, 然后(递归地)搜寻Base1 的基类, 然后如果还是没有找到, 那么就会搜索Base2, 等等。

事实上, 这更加的复杂; 方法的搜寻顺序会根据调用super() 而变化。

动态的顺序是很有必要的, 因为在那些处于菱形继承体系中(这里至少有个父类被多次派生). 比如, 所有的类都从object 派生, 所以到达object的路径不止一条. 为了防止基类被多次访问, 动态的算法线性化了搜寻的路径, 先从左至右搜索指定的类, 然后这样就可以让每个父类只搜寻一次, 并且单一(这就意味一个类可以被派生, 但是不会影响其父类的搜寻路径.

### 7.5 私有变量

在Python 之中, 并不存在那种无法访问的“私有” 变量. 但是, 在多数的Python 代码中有个约定: 以一个下划线带头的名字(如\_spam) 应该作为非公共的API (不管是函数, 方法或者数据成员). 这应该作为具体的实现, 而且变化它也无须提醒.

因为有一个合法的情况用于使用私有的成员(名义上是说在派生类中避免名字的冲突), 因此就有这样的一种机制称为name mangling.任何如\_\_spam 形式的标识符, (在开头至少有两个下划线) 将被替换为\_classname\_\_spam, 此处的classname 就是当前的类. 这样的处理无须关注标识符的句法上的位置, 尽管它是在一个类的定义中。

### 7.6 空的类定义（类似struct的作用）

有些时候, 有类似于C 的“struct” 这样的数据类型非常有用, 绑定一些命名的数据. 一个空的类定义就将很好:

|  |
| --- |
| class Employee:  pass  john = Employee() # Create an empty employee record  # Fill the fields of the record  john.name = 'John␣Doe'  john.dept = 'computer␣lab'  john.salary = 1000 |

一段Python 代码中如果希望一个抽象的数据类型, 那么可以通过传递一个类给那个方法, 就好像有了那个数据类型一样。

例如, 如果你有一个函数用于格式化某些从文件对象中读取的数据, 你可以定义一个类, 然后有方法read() 和readline() 用于读取数据, 然后将这个类作为一个参数传递给那个函数。

### 7.7 异常也是类

### 7.8 迭代器

#### 7.8.1 内置迭代器

大多数的容器对象都可以使用for 来迭代:

|  |
| --- |
| for element in [1, 2, 3]:  print(element)  for element in (1, 2, 3):  print(element)  for key in {'one':1, 'two':2}:  print(key)  for char in "123":  print(char) |

在这个外表之下, for 语句对容器对象调用了iter(). 这个函数返回一个迭代器对象, 它定义了\_\_next\_\_() 方法, 用以在每次访问时得到一个元素.当没有任何元素时, \_\_next\_\_() 将产生StopIteration 异常, 它告诉for停止迭代.

你可以使用内置函数next() 来调用\_\_next\_\_() 方法; 这个例子展示了它如何工作:

|  |
| --- |
| >>> s = 'abc'  >>> it = iter(s)  >>> it  <iterator object at 0x00A1DB50>  >>> next(it)  'a'  >>> next(it)  'b'  >>> next(it)  'c'  >>> next(it)  Traceback (most recent call last):  File "<stdin >", line 1, in ?  next(it)  StopIteration |

#### 7.8.2 定义类的迭代器

定义一个\_\_iter\_\_() 方法用以返回一个具有\_\_next\_\_() 的对象. 如果这个类定义了\_\_next\_\_() , 那么\_\_iter\_\_() 仅需要返回self:

|  |
| --- |
| class Reverse:  "Iterator␣for␣looping␣over␣a␣sequence␣backwards"  def \_\_init\_\_(self , data):  self.data = data  self.index = len(data)  def \_\_iter\_\_(self):  return self  def \_\_next\_\_(self):  if self.index == 0:  raise StopIteration  self.index = self.index - 1  return self.data[self.index]  >>> rev = Reverse('spam')  >>> iter(rev)  <\_\_main\_\_.Reverse object at 0x00A1DB50>  >>> for char in rev:  ... print(char)  m  a  p  s |

## 八、标准库的简明介绍

### 8.1 与操作系统接口 os模块 shutil模块 glob模块

os 模块提供了许多与操作系统交互的接口:

|  |
| --- |
| >>> import os  >>> os.getcwd() **# 返回当前工作目录(current working directory)**  >>> os.chdir('/server/accesslogs ') **# 改变当前工作目录**  >>> os.system('mkdir today ') **# 在系统的shell 中运行mkdir 命令0**  >>> dir(os)  **<returns a list of all module functions>**  >>> help(os)  **<returns an extensive manual page created from the module 's docstrings >** |

对于日常文件和目录的管理, shutil 模块提供了更便捷、更高层次的接口:

|  |
| --- |
| >>> import shutil  >>> shutil.copyfile('data.db', 'archive.db')  >>> shutil.move('/build/executables ', 'installdir ') |

glob 模块提供了这样一个函数, 这个函数使我们能以通配符的方式搜索某个目录下的特定文件, 并列出它们:

|  |
| --- |
| >>> import glob  >>> glob.glob('\*.py')  ['primes.py', 'random.py', 'quote.py'] |

### 8.2 sys模块 argparse模块

一些实用的脚本通常需要处理命令行参数. 这些参数被sys 模块的argv 属性以列表的方式存储起来. 下例中, 命令行中运行python demo.py one two three , 其结果便能说明这一点:

|  |
| --- |
| >>> import sys  >>> print(sys.argv)  ['demo.py', 'one', 'two', 'three '] |

argparse 提供了强大且更灵活的命令行参数处理方法。

sys 模块还包括了stdin, stdout, stderr 属性. 而最后一个属性stderr可以有效地使警告和出错信息以可见的方式传输出来, 即使是stdout 被重定向了:

最直接地结束整个程序的方法是调用sys.exit().

### 8.3 math模块 random模块

math 模块使我们可以访问底层的C 语言库里关于浮点数的一些函数；

random 模块提供了产生随机数的工具；Scipy项目里有许多关于数值计算的模块.

|  |
| --- |
| >>> import math  >>> math.cos(math.pi / 4)  **#math.cos()**  0.70710678118654757  >>> math.log(1024 , 2) **#math.log()**  10.0  import random  >>> random.choice(['apple ', 'pear', 'banana '])  'apple '  >>> random.sample(range(100), 10) #生成无需更换的随机抽样样本  [30 , 83 , 16 , 4, 8, 81 , 41 , 50 , 18 , 33]  >>> random.random() # 生成随机浮点数  0.17970987693706186  >>> random.randrange(6) # 以range(6)里的数为基准生成随机整数  4 |

### 8.4 datetime模块 timeit模块 time模块

datetime 模块提供了操作日期和时间的类, 包括了简单和复杂两种方式.

当我们知道了时间和日期的算法后, 工作的重心便放在了如何有效地格式化输出和操作之上了. 该模块也提供了区分时区的对象.

|  |
| --- |
| >>> from datetime import date  >>> now = date.today()  >>> now  datetime.date(2003 , 12 , 2)  >>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")  '12 -02 -03. 02 Dec 2003 is a Tuesday on the 02 day of December.'  # dates support calendar arithmetic  >>> birthday = date(1964 , 7, 31)  >>> age = now - birthday  >>> age.days  14368 |

**timeit模块：准确测量小段代码的执行时间**

（1）timeit 模块中的三个函数

timeit.timeit(stmt='pass', setup='pass', timer=<default timer>, number=1000000)：创建一个Timer实例，参数分别是stmt（需要测量的语句或函数），setup（初始化代码或构建环境的导入语句），timer（计时函数），number（每一次测量中语句被执行的次数）

timeit.repeat(stmt='pass', setup='pass', timer=<default timer>, repeat=3, number=1000000)：创建一个Timer实例，指定整个试验的重复次数，返回一个包含了每次试验的执行时间的列表，利用这一函数可以很方便得实现多次试验取平均的方法。

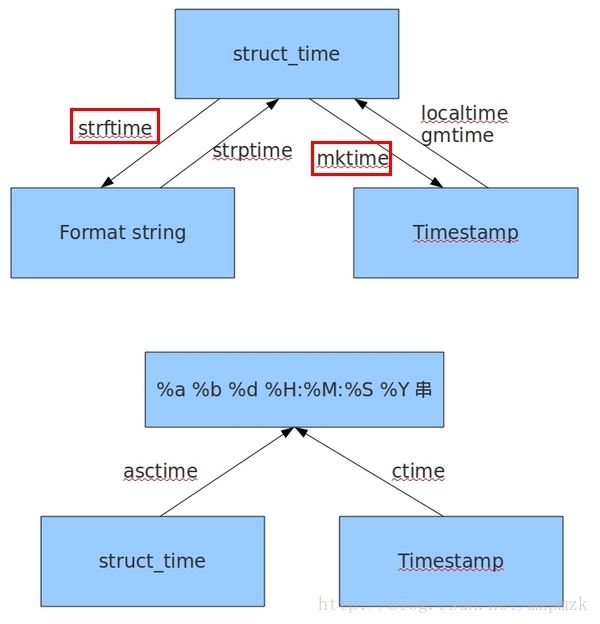
timeit.default\_timer()：默认的计时器，一般是time.perf\_counter()，time.perf\_counter()方法能够在任一平台提供最高精度的计时器（它也只是记录了自然时间，记录自然时间会被很多其他因素影响，例如计算机的负载）

|  |
| --- |
| import timeit  def test():  L = []  for i in range(100):  L.append(i)  if \_\_name\_\_ == '\_\_main\_\_':  print timeit.timeit("test()", setup="from \_\_main\_\_ import test")  x = list(range(2000000))  t0 = timeit.timeit("x.pop(0)", "from \_\_main\_\_ import x", number=1000)  print("cost {} seconds ".format(t0)) |

**time模块：**

1. **时间表现形式**

* timestamp：时间戳，时间戳表示的是从1970年1月1日00:00:00开始按秒计算的偏移量
* struct\_time： 时间元组，共有九个元素组
* format time： 格式化时间，已格式化的结构使时间更具可读性。包括自定义格式和固定格式



1. **常用方法**

|  |
| --- |
| # 1、获取当前时间  time.time() # 获取当前时间戳  >1500252381.100721  time.localtime() # 当前时间的 struct\_time 形式  >time.struct\_time(tm\_year=2017, tm\_mon=7, tm\_mday=17, tm\_hour=9, tm\_min=6, tm\_sec=29, tm\_wday=0, tm\_yday=198, tm\_isdst=0)  time.ctime() == time.asctime() # 当前时间的字符串形式  >'Mon Jul 17 09:08:20 2017'  # 2、把当前时间戳转化为字符格式  time.strftime('%Y-%m-%d-%H-%M-%S', time.localtime())  >'2017-07-17-09-10-14'  # 3、用来衡量不同程序的耗时的 time.clock()  >在UNIX系统上，它返回的是“进程时间”，它是用秒表示的浮点数（时间戳）。而在WINDOWS中，第一次调用，返回的是进程运行的实际时间。而第二次之后的调用是自第一次调用以后到现在的运行时间。  start = time.clock()  test()  finish = time.clock() |

### 8.5 数据压缩 zlib、gzip、ziffile、tarfile模块

有些模块可以支持常规的数据压缩和解压, 这些模块块包括: zlib, gzip, zipfile 和tarfile.

|  |
| --- |
| >>> import zlib  >>> s = b'witch which has which witches wrist watch '  >>> len(s)  41  >>> t = zlib.compress(s)  >>> len(t)  37  >>> zlib.decompress(t)  b'witch which has which witches wrist watch '  >>> zlib.crc32(s)  226805979 |

### 8.6 访问互联网 smtplib模块和urllib.request模块

python 里包含了许多访问互联网和处理互联网协议的模块. 其中最简单的两个分别是, 从网址中检索数据的urllib.request 模块, 和发送邮件的smtplib 模块。

|  |
| --- |
| >>> from urllib.request import urlopen  >>> for line in urlopen('http :// tycho.usno.navy.mil/cgi -bin/timer.pl'):  ... line = line.decode('utf -8') #将二进制文件解码成普通字符  ... if 'EST' in line or 'EDT' in line: #查找西方国家的时间  ... print(line)  <BR>Nov. 25 , 09:43:32 PM EST  >>> import smtplib  >>> server = smtplib.SMTP('localhost ')  >>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',  ... """To: jcaesar@example.org  ... From: soothsayer@example.org  ...  ... Beware the Ides of March.  ... """)  >>> server.quit()  (注意：第二个例子需要本地有一个邮件服务器. ) |