Python基础

一、什么是程序

程序是一组定义如何进行计算的指令的集合。这种计算可能是数学计算，比如解方程组或者找多项式的根，也可以是符号运算，如搜索和替换文档中的文本，或者图形相关的操作，如处理图像或播放视频。

在不同的编程语言中，程序的细节有所不同，但几乎所有编程语言中都会出现以下几类基本指令：

- **输入**：从键盘、文件或者其他设备中获取数据。
- **输出**：将数据显示到屏幕，保存到文件中，或者发送到网络上等。
- **数学**：进行基本数学操作，如加减乘除。
- **条件执行**：检查某种条件的状态，并执行相应的代码。
- **重复**：重复执行某种动作，往往在重复中有一些变化。

这差不多就是全部了。我们所遇到过的所有程序，无论多么复杂，都是由类似上面的这些指令组成的。所以我们可以把编程看作一个将大而复杂的任务分解为更小的子任务的过程，不断分解，直到任务简单到足以由上面的这些基本指令组合完成。

1.1 程序结构

在Python环境或者IDE中，其中Python.py是文件文件名，Python是模块名，.py是后缀名。

在Jupyter Notebook，我们的文件文件名为Python.ipynb，当然 Jupyter也提供转换功能：

> File ----> Download as ----> Python(.py)

1.2 形式语言和自然语言

自然语言是指人们所说的语言，如中文、英语、西班牙语、法语等等。它们不是由人设计而来的（虽然人们会尝试加以语法限制），而是自然演化而来的。

形式语言则是人们为了特殊用途设计的语言。例如，数学上使用的符号体系是一种特别擅于表示数字和符号之间关系的形式语言；化学家则使用另一种形式语言来表示分子的化学结构。

而这里最重要的是：**编程语言是人们为了表达计算过程而设计出来的形式语言。**

形式语言的密度远远大于自然语言，所以阅读起来需要花费更多的时间。当然，结构也非常重要，直接自上而下的阅读顺序有时候并一定是最好的。相反，我们要试着学会先在头脑中解析程序，辨别并解释出结构。最后，细节很重要。在自然语言中人们常常可以忽略一些小错误，如拼写错误或者标点符号错误，但在形式语言中则会造成很大的差别和错误。

1.3 调试

程序是很容易出错的。因为某种古怪的原因，程序错误被称为bug，而查补bug的过程称为**调试**（debugging）。

一个程序中可能出现3中类型的错误：语法错误、运行时错误和语义错误。对它们加以区分，可以更快更高效地找到错误。

当计算机表现好的时候，我们认为它就是 ”神助攻“ ，而当它们很固执的时候，又会感觉很无助。其实对这些反应做好准备有助于我们对付它们。我们需要充分利用优点、摒弃弱点，学习调试可能很令人泄气，但是它对于许多编程之外的活动也是一个非常有价值的技能。

1.4 术语表

问题求解（problem solving）：将问题形式化、寻找并表达解决方案的过程。

高级语言（high-level language）：像Python这样被设计成人类容易阅读和编写的编程语言。

低级语言（low-level language）：被设计成计算机容易运行的编程语言，也被称为 ”机器语言“（machine language）或 ”汇编语言”（assembly language）。

可移植性（portability）：程序能够在多种计算机上运行的特性。

解释器（interpreter）：读取另一个程序并执行该程序的程序。

提示符（prompt）：解释器所显示的字符，表明以准备好接收用户的输入。

程序（program）：一组定义了计算内容的指令。

打印语句（print statement）：使Python解释器在屏幕上显示某个值的指令。

运算符（operator）：代表类似加法、乘法或者字符串连接（string concatenation）等简单计算的特殊符号。

值（value）：程序所处理数据的基本元素之一，例如数字或字符串。

类型（type）：值的类别。我们目前接触的类型有整型数（int）、浮点数（float）和字符串（str）。

自然语言（natural language）：任何的人们日常使用的、由自然演变而来的语言。

形式语言（formal language）：任何由人类为了某种目的而设计的语言，例如用来表示数学概念或者电脑程序；所有的编程语言都是形式语言。

记号（token）：程序语法结构中的基本元素之一，与自然语言中的单词类似。

语法（syntax）：规定了程序结构的规则。

解析（parse）：阅读程序，并分析其语法结构的过程。

故障（bug）：程序中的错误。

调试（debugging）：寻找并解决错误的过程。

二、基础语法

2.1 标识符和保留字

标识符：标志不同的词法单位，通俗讲就是名字。由一串字符构成，字母，数字，下划线，中文，并且不能以数字开头。大小写敏感，名字不能和关键字/保留字相同。

关键字/保留字：已经被系统用或者留。

执行下方命令可以查看关键字和保留字：

```
import keyword
keyword.kwlist
```

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

2.3 变量和常量

变量是指运行过程中可以被修改的值。

常量是指初始化后就保持不变的值。

2.3.1 变量

变量定义是通过对变量的第一次进行赋值来实现的。下面的=号叫赋值。

```
x          # x是没有定义的变量，报错
x = 1      # 对x的第一次赋值，也就是对x的定义，此后x存在了
del x      # 从内存中删除
x          # 不能再被访问了
```

2.3.2 常量

Python中没有专门定义常量的关键字。

2.4 输入和输出

2.4.1 输入

标准输入设备默认是键盘。

```
x = input(<提示字符串>)  # 等待键盘输入
```

2.4.2 输出

标准输出设备默认是显示器。

```
print(<输出值1>[,<输出值2>,<输出值3>……,<输出值n>,sep=',',end='\n'])
```

sep是输出值之间的分隔符，如果不设置，默认为空格

end默认为换行

```
print('abc',123)    # abc 123
print('abc',123,sep=',')  # abc,123
```

三、标准数据类型

Python中有很多内置的数据类型，以下是我们使用的较多的数据类型。

- 布尔值（Booleans）True或False
- 数字（Number）整数、浮点数、分数、复数
- 字符串（Strings）Unicode字符序列
- 列表（Lists）有序的值的可变序列
- 元组（Tuples）有序的值的不可变序列
- 集合（Sets）无序且不重复的值的集合
- 字典（Dictionaries）无序的键-值对的集合

3.1 类型判断

type(obj): 返回obj的类型

```
type(1)      # int
type(1.0)    # float
type('1')    # str
type(True)   # bool
```

isinstance(obj,class): 测试对象obj是class的实例

```
isinstance(1,int)     # True
isinstance('1',str)   # True
```

3.2 布尔值 Booleans

和布尔代数的表示完全一致，一个布尔值只有True、False两种值，要么是True，要么是False。

```
type(True)   # bool
```

3.2.1 布尔运算（and，or，not）

```
True and False   # False
```

```
0 or []   # []
```

```
True or False   # True
```

```
not True   # False
```

Note：1==1.0?

3.3 数字

数字有四种类型：整数（int），浮点数（float），分数（fractions），复数（complex）

3.3.1 整数 int

Python可以处理任意大小的整数，当然包括负整数，在程序中的表示方法和数学上的写法一模一样，例如：1，100，-8080，0，等等。

在解释器中输入一个整数也会得到一个同样整数的输出：

```
2  # 2
2-5  # -3
2/5  # 0.4
```

3.3.2 浮点数（float）

浮点数也就是小数，浮点数可以用数学写法，如1.23，3.14，-9.01.

但是对于很大或很小的浮点数，就必须用科学记数法表示。

```
1.23e9   # 1230000000.0
```

```
1.2345678901234567890  # 1.1234567890123457
```

> **Note**
>
> 整数和浮点数在计算机内部存储的方式是不同的，整数运算结果永远是精确的，而浮点数运算则可能会有精度上的误差。
> 浮点数存在上界和下界，超过结果导致溢出。
> 由于存储有限，计算机不能精确显示无限小数，会产生误差。另外计算机内部采用二进制，不是所有的十进制实数都可以用二进制数精确表示。

查看float的最大值和最小值：

```
import sys
sys.float_info
```

3.3.3 分数 fractions

在Python中，不止有浮点数（float），而且还有分数（Fraction）这个类型。

要使用分数，必须引入一个模块：

```
import fractions
```

然后就可以声明一个分数了：

```
x = fractions.Fraction(1,2)
```

这样就声明了一个二分之一的分数，打印这个变量则会得出：1/2

声明二分之一还可以用另一种方式：

```
x = fractions.Fraction(0.5)
```

Python会自动转换浮点数为分数，结果也是1/2.

需要注意的是，Python会自动进行约分：

```
import fractions
x = fractions.Fraction(0.2)
y = fractions.Fraction(2/10)
x == y
```

结果为True。

3.3.4 复数 complex

复数对象有两个属性**real**和**imag**用于查看**实部**和**虚部。**

- 语法：real + imagj
- 虚数不能单独存在，它们总是和一个值为0.0的实数部分一起构成一个复数
- 实数部分和虚数部分都是浮点数
- 虚数部分必须有 j 或 J

```
num.real   # 返回该复数的实数部分
```

```
num.imag   # 返回该复数的虚数部分
```

```
num.conjugate()   # 返回该复数的共轭复数
```

3.3.4.1 complex() 函数

complex() 函数用于创建一个复数或者将一个数或字符串转换为复数形式，其返回值为一个复数。该函数的语法为：

```
complex(real,imag)
```

其中，real可以为int、float或字符串类型；而imag只能为int或float类型。

```
complex(1)   # 数字
complex('1')   # 当作字符串处理
complex('1 + 2j')   # 会出错，+号两边不能有空格
complex('1+2.0')
```

Note
如果第一个参数为字符串，第二个参数必须省略；若第一个参数为其他类型，则第二个参数可以选择。
第一个参数为字符串，如果仍添加第二个参数时会报错。

```
>>> complex('a',1)
# TypeError: complex() can't take second arg if first is a string
```

3.4 字符串

- 字符串是字符组成的序列。可以用单引号、双引号和三引号。
- 单引号和双引号用法没有区别。
- 三引号允许一个字符串跨多行，其中可以包含换行符、制表符以及其他特殊字符。
- 如果字符串中出现转义符，可以用r或R来定义原始字符串，这样可以不让转义字符生效。

```
### 单引号、双引号和三引号

name_1 = 'Evan'
name_2 = "Evan"
name_3 = '''Evan'''

print(name_1,name_2,name_3)
```

3.4.1 创建字符串

```
str = 'cda123cda'
```

3.4.2 字符串运算符

- 字符串连接：使用 + 相当于拼接两个字符串

```
'Hello' + 'Python'     # HelloPython
```

- 重复输出字符：使用 *n 相当于把字符串重复n次

```
'cda' * 3    # cdacdacda
```

3.4.3 转义字符

\可以转义很多字符，比如**\n表示换行**，**\t表示制表符**，字符\本身也要转义，所以\\表示的字符就是\。

Python还允许用 r'' 表示 '' 内部的字符串默认不转义。

```
### 字符串有转义符

print(r'\t\n')    # \t\n
print('\\t\\n')   # 结果和上面一样，但是需要对\转义，也就是使用\\
```

3.4.4 多行字符串

如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，Python允许用 '''...'''

的格式表示多行内容。

同时也可以结合 r'' 使用。

```
text = r'''My
Apple\/
Pen'''

# text
# print(text)
```

3.4.5 字符串切片

字符串也是一个可迭代对象，也就是说每一个字符串实际上都有一个对应的索引值。

```
str_me = 'dfedsfdf'
str_me[0]
str_me[1]
```

有时候我们不仅仅想取出字符串的其中一个字符，还想要去除其中一部分，这时候应该怎么做呢?

```
x = 'abcdefgabcd'
x[0:3:1]   # 取出前三个
```

字符串切片的标准语法:

string[开始位置:终止位置:步长和方向]
注意：选区的区间属于**左闭右开**型，即从“起始”位开始，到“结束”位的前一位结束（不包含结束位本身）。

```
str[0]      # 获取第一个元素
str[-2]     # 获取倒数第二个元素
str[1:3]    # 获取从偏移为1的字符一直到偏移为3的字符串，不包括偏移为3的字符串
str[1:]     # 获取从偏移为1的字符一直到字符串的最后一个字符（包括最后一个字符）
str[:3]     # 获取从偏移为0的字符一直到偏移为3的字符串，不包括偏移为3的字符串
str[:-1]    # 获取从偏移为0的字符一直到最后一个字符（不包括最后一个字符串）
str[:]      # 获取字符串从开始到结尾的所有元素
str[-3:-1]  # 获取偏移为-3到偏移为-1的字符，不包括偏移为-1的字符
str[-1:-3]和str[2:0]    # 获取的为空字符，系统不提示错误
str[::2]    # 步长为2，方向从左到右
str[-1:-5:-2] # 步长为2，方向从右往左
```

3.4.6 字符串方法

3.4.6.1 find

检测 str 是否包含在mystr中，如果是则返回开始的索引值，否则返回-1

mystr.find(str, start=0, end=len(mystr))

```
a = 'abcdefgabc'

q = a.find('a',2,3)
q

a[a.find('e')]

k = a.find('abc',3)
a[k:k+3]
```

3.4.6.2 count

返回 str 在 start 和 end 之间在mystr里里面出现的次数

mystr.count(str, start=0, end=len(mystr))

```
a.count('e',0,5)   # 1

a.count('ba')      # 0
```

3.4.6.3 replace

把mystr中的 str1 替换成 str2，如果count指定，则替换不超过count次

mystr.replace(str1, str2, mystr.count(str1))

```
a.replace('a','哈哈',1)

a = a.replace('a','哈哈哈')

a.replace('a','k')  # 这里我们要注意一下a到底变没变，为什么?
```

3.4.6.4 split

以 str 为分隔符切片mystr，如果maxsplit有指定值，则仅分隔maxsplit个字符串

mystr.split(str=" ", 2)

```
a = 'abcdefgabc'

a.split('b')

a.split('e')

a.split('q')

a.split('c')
```

3.4.6.5 splitlines

按照行分隔，返回一个包含各行作为元素的列表

mystr.splitlines()

```
data = '''关于你
关于你，我有太多东西关于你。
清醒的时候放不下矜持，不敢说我喜欢你，只有在某个夜晚多愁善感又萦绕在心头，或是朋友聚会上的大醉，才敢借着情绪说，我喜欢你，喜欢了好久好久。'''

lst1 = data.splitlines(True)

lst1[1] # 我们也可以进行索引
```

3.4.6.6 lower

转换mystr中所有大写字符为小写

mystr.lower()

3.4.6.7 upper

转换 mystr 中的小写字母为大写

mystr.upper()

3.4.6.8 strip

删除mystr字符串两端的指定字符，如果不指定默认是空格

mystr.strip()

```
b = ' goo gle   '
b.strip()
```

3.4.6.9 partition

把mystr以str分割成三部分，str前，str和str后

mystr.partition(str)

```
a = 'abcdefgabc'

# 比较split和partition
a.split('e')
a.partition('e')
```

3.4.6.10 join

mystr中每个字符后面插入str，构造出一个新的字符串

mystr.join(str)

```
'-->'.join('北京大学在北京')
```

3.4.6.11 判断字符串内的元素类型

- isalpha：如果mystr所有字符都是字母，则返回True，否则返回False

  mystr.isalpha()
- isdigit：如果mystr只包含数字，则返回True，否则返回False

  mystr.isdigit()
- isalnum：如果mystr所有字符都是字母或数字，则返回True，否则返回False

  mystr.isalnum()
- isspace：如果mystr中只包含空格，则返回True，否则返回False

  mystr.isspace()
- islower：如果mystr中的字母全是小写，则返回True，否则返回False

  mystr.islower()
- isupper：如果mystr中的字母全是大写，则返回True，否则返回False

  mystr.isupper()
- istitle：如果mystr中的首字母大写，则返回True，否则返回False

  mystr.istitle()

3.4.6.12 方法小结

由于str是**不可变**的序列，其实只有查找方法，增删改不会改变字符串本身，只是结果改变。

查

1）检测字符串是否包含子串：index和find
两者区别在于如果没有找到，find返回-1，index报错
'abc123'.index('c1') # 2
'abc123'.find('c1') # 2
'abc123'.find('c2') # -1
'abc123'.index('c2') # 出错
2）统计子串出现次数：count
'abc123abc'.count('b') # 2

增

1）用分隔符拼接字符串：join
'1234'.join('abcd') # a1234b1234c1234d
2）连接字符串：使用运算符 +
3）复制字符串：使用运算符 *

删

1）**mystr.strip(|指定字符|)** 删除字符串两端的指定字符，如果不指定默认是空格
'#abc##'.strip('#') # abc
' abc '.strip() # abc
另外，lstrip和rstrip，分别是删除左边空格和删除右边空格

改

1）**转换小写：lower()**
'aBcD'.lower() # abcd
2）**转换大写：upper()**
'aBcD'.upper() # ABCD
3）**首字母大写，其他小写：capitalize()**
'aBcD'.capitalize() # Abcd
4）**交换大小写：swapcase()**
'aBcD'.swapcase() # AbCd

分割

split([分隔符]) 默认是空格，\t，\n分割
'I love you'.split() # 返回3个元素的列表
'a#b#c'.split('#') # 返回3个元素['a','b','c']的列表

检测

startswith(prefix[,start [,end]]) # 是否以prefix开头
endswith(suffix[,start[,end]]) # 以suffix结尾
isalnum() # 是否全是字母和数字，并至少有一个字符
isalpha() # 是否全是字母，并至少有一个字符
isdigit() # 是否全是数字，并至少有一个字符
isspace() # 是否全是空白字符，并至少有一个字符
islower() # str中的字符是否全是小写
isupper() # str中的字符是否全是大写
istitle() # str是否是首字母大写

3.4.7 函数

```
len(s)    # 求字符串的长度
max(s)    # 求字符串的最大数
min(s)    # 求字符串的最小数
```

3.4.8 数字和字符串的相互转换

```
# float: 将其他类型数据转换为浮点数
float(1)
float('1.23')
float('1.2e-3')
```

```
# str: 将其他类型数据转换为字符串
str(1)
str(1.0)
str(1.0e-5)
```

```
# int: 将其他类型数据转换为整数
int(3.14)     # 3
int(3.5)      # 3 扔掉小数部分
int(True)     # 1
int('3.5')    # 报错
int(float('3.5'))    # 3
```

```
# bool: 把其他类型转换为布尔类型
bool(0)      # False
bool(-1)     # 所有非0数值都为真
bool('a')    # True
bool('')     # False
```

```
# chr和ord 整数和字符
chr(65)
ord('a')
```

3.4.9 字符串的格式化

3.4.9.1 字符串格式化输出

方法一：format_string % obj

```
print('%s的年龄是%d' % ('张三',20))    # 将每个值放在一个圆括号内，逗号隔开
```

方法二：str.format( ) 普通形式

```
'{0}的年龄是{1}'.format('张三',20)    # 和上面的区别是用{}代替%，但功能更强大
```

```
"{0}就是一个大城市{1}，{0}房价很贵，{0}是一个古城".format("郑州","的代表")
```

方法三：str.format( ) 接受参数形式

```
'{name}的年龄是{age}'.format(age=20,name='张三')    # 参数位置可以不按顺序显示。
```

```
姓名 = '李先生'
节日名称 = '春节快乐'

print('''亲爱的{填写姓名}:
    您好，祝您{填写节日名称}!'''.format(填写姓名=姓名,填写节日名称=节日名称))
```

除此之外还有一些稍微复杂一点的格式化方法。

3.4.9.2 填充与对齐

填充常跟对齐一起使用，^<>分别是居中、左对齐、右对齐，后面带宽度

:号后面带填充的字符，只能是一个字符，不指定的话默认是用空格填充

```
# 我想填一个占宽8个位置的，如果不足就用空格填充
'{:^8}'.format('2333')    # 居中
'{:<8}'.format('2333')    # 左对齐
'{:>8}'.format('2333')    # 右对齐
```

3.4.9.3 精度与类型

```
# 保留两位有效数字
'{:.2f}'.format(3.1415926)
```

3.4.9.4 金额的千位分隔符

```
'{:,}'.format(1234567890)
```

3.5 表达式与运算符

3.5.1 表达式概念

- 表达式是由一个或多个操作数及零个或运算符组成的序列，其计算结果为一个值、对象、方法或命名空间。
- 表达式可以包含文本值、方法调用、运算符以及其操作数、或简单名称。简单名称可以是变量名、类型成员名、方法参数名、命名空间或类型名。
- 表达式可以使用运算符（运算符又可以使用其他表达式作为参数）或方法调用（方法调用的参数又可以是其他方法参数），因此表达式可以很简单，也可以非常复杂。

3.5.2 运算符种类

运算符类型有算数运算符、关系运算符、赋值运算符、逻辑运算符和条件运算符。

按照运算需要的操作数目，可以分为一元，二元，三元运算符：

- 一元：就是只需要一个操作数。例：+，-
- 二元：需要两个操作数。大多数都是二元。
- 三元：需要三个操作数，条件运算是三元运算符。例：b if a else c

3.5.2.1 算数运算符

算术运算符是一个二元运算符，主要包括：

运算符	描述	实例
+	加	两个对象相加a+b
-	减	得到负数或是一个数减去另一个数a-b
*	乘	两数相乘或返回一个被重复若干次的字符串a*b
/	除	a除以b
//	整除	返回商的整数部分（向下取整）
**	乘方（幂）	返回x的y次幂
%	取模（取余）	返回除法的余数a%b

注意：只要有任意一个操作数是浮点数，结果就会是浮点数。

3.5.2.2 关系运算符

用于比较两个表达式的值，关系运算符包括：

运算符	描述
=	检查两个操作数的值是否相等，如果是则条件变为真
!=	检查两个操作数的值是否相等，如果值不相等，则条件变为真
>	检查左操作数的值是否大于右操作数的值，如果是，则条件成立
<	检查左操作数的值是否小于右操作数的值，如果是，则条件成立
>=	检查左操作数的值是否大于或等于右操作数的值，如果是，则条件成立
<=	检查左操作数的值是否小于或等于右操作数的值，如果是，则条件成立

3.5.2.3 赋值运算符

运算符	描述
=	赋值
*=	复合赋值

运算符	描述	实例
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c *= a 等效于 c = c * a
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a 等效于 c = c ** a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

除此之外，还有序列赋值：

```
x,y,z = '1','2','3'    # 实质：右边是元组的一种写法，x=1,y=2,c=3
x,y,z = '123'    # 等价于上面
x,y,z = '1234'  # 报错，参数太多
x,y,z = '1','2','3','4'    # 报错，参数太多
```

3.5.2.4 逻辑运算符

对于逻辑 **“与”，“或”，“非”**，我们使用**and，or，not**这几个关键字。

逻辑运算符and和or也称作短路运算符：它们的参数从左向右解析，一旦结果可以确定就停止。例如，如果A和C为真而B为假，A and B and C不会解析C。在作用于一个普通的非逻辑值时，短路运算符的返回值通常是能够最先确定结果的那个操作数。

关系运算可以通过逻辑运算符and和or组合，比较的结果可以用not来取反意。逻辑运算符的优先级又低于关系运算符，在它们之中，not具有最高的优先级，or优先级最低，所以 A and not B or C 等于 (A and (not B)) or C 。

运算符	逻辑表达式	描述
and	x and y	布尔“与”，如果x为False，x and y 返回False，否则它返回y的计算值
or	x or y	布尔“或”，如果x是True，它返回True，否则它返回y的计算值
not	not x	布尔“非”，如果x为True，返回False；如果x为False，它返回True

下面是一些实例：

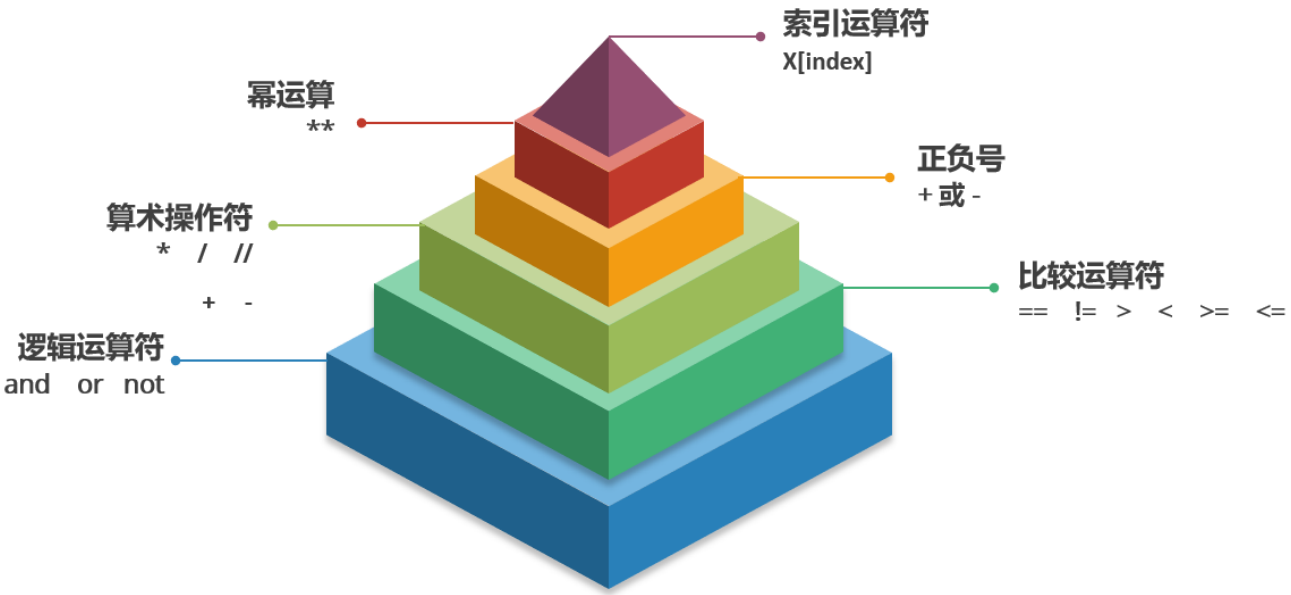
```
5 and 4   # 4

0 and 4   # 0

False or 3 or 0   # 3

2 > 1 and not 3 > 5 or 4   # True
```

3.5.3 运算的优先级



3.5.4 math科学计算库

math模块的内置常量：

```
math.pi    # 3.141592653589793
math.e     # 2.718281828459045
```

Python内置的一些计算函数：

内置计算函数	描述
abs(x)	返回x的绝对值，类型随x
max(n1,n2, . . .)	返回最大值
min(n1,n2, . . .)	返回最小值
round(x,[,n])	**四舍六入，五找偶数** 如给出n值，则代表会四舍五入到小数点后的n位<br /
divmod	取模，返回商和余数。例：divmod(5,2) 返回 (2,1)
sum	求和。例：sum([1,2,3,4]) 返回 10

Python还提供科学计算等库，例如math，导入math库后，常用的函数有：

math	描述
ceil(x)	取x的上入整数
floor(x)	取x的下入整数
exp(x)	返回e的x次幂，e是自然常数
sqrt(x)	返回x的平方根，返回值是float类型
modf(x)	返回x的整数部分和小数部分，两部分的符号与x相同， 整数部分以浮点型表示
log10(x)	返回以10为底的x的对数，返回值类型是浮点数
log(x,y)	返回以y为底的x的对数，返回值类型是浮点数
pow(x,y)	返回x的y次幂，即 x**y

四、组合数据类型

下面介绍着数据类型：**列表list，元组tuple，字典dict，集合set。**

根据官方文档的数据，Python没有数组类型。在后面会学习到的NumPy中有一个类型是ndarray（n维数组）

上面组合类型又可以分为三大类型：

1. **序列**

Python内部的序列类型分为两类：

- 不可变序列

  tuple，str

- 可变序列

  list

2. **集合set**

   集合对象是一个由互不相同的hashable对象组成的无序集合。

   set可变，frozenset不可变。

3. **字典dict**

   str的运算符，函数，list和tuple都具备。

4.1 列表

列表（list）目的是存储或操作一组数据的集合。

当我有一百个数据要进行储存的时候，我可以选择用100个变量分别进行储存，但显然这不是一个更好的方案。我们可以用一个集合形式的数据结构，把这一百个数据储存到一块，需要的时候再分别进行提取。

这就是列表、集合等数据类型存在的意义。

我们可以把它想象成一个大桶，当我们有一堆东西需要找个地方临时存放在一起，以便后续进行排序、筛选等操作时，就可以弄一个列表，把这些东西放进去。

4.1.1 函数list

鉴于不能像修改列表那样修改字符串，因此在有些情况下使用字符串来创建列表很有帮助。为此，可使用函数list。

```
list('Hello')
```

请注意，我们可以将任何序列（而不仅仅是字符串）作为list的参数。

```
d = {'a':1, 'b':2}
list(d)
```

如果要将字符列表（如上述代码中的字符列表）转换为字符串，可使用下面的表达式：

```
''.join(somelist)
```

其中somelist是要转换的列表。这到底是什么意思呢？我们3.4.6.10节对此做了说明。

4.1.2 列表的定义方法

- 列表是一种可变的数据类型
- 列表中的数据类型不限
- 列表中的多个值之间用逗号进行分割

```
# 列表中的内容可以是任意的数据类型
list1 = [1234,'Hello',3.14,True,'abc']
```

```
list2 = [33434,
         '你好', 4545.565555,
         'iii',
         True]
```

```
list3 = [1, 2, 3, 'Hello', [1, 2, 3, 4, 5] ]   # 二维列表
```

```
list4 = [1, 2, 3, 'Hello', [1, 2, [1, [1, 2, 3], 3], 4, 5] ]   # 二维列表
```

4.1.3 列表的索引和切片

列表：有序的序列，所以每一个数据都有唯一对应的索引值。

- **语法：列表[start:stop:step]**
- 列表[起始位置:终止位置:步长和方向]，开始的位置包含在内，而终止的位置不包含在内
- 步长的默认值为1，当步长设置为正整数，代表方向从左往右，当步长设置为负整数，代表从后往前进行切片
- 起始位置和终止位置，如果不填，代表从头到尾所有数据

4.1.4 二维列表的索引方法

```
list5 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# 索引出1
list5[0][0]

# 索引出[7,8,9]
list5[-1]

# 反转
list5[::-1]
```

4.1.5 更改列表中的值

```
list5 = ['零', '一', '二', '三', '四', '五', '六']

# 将“一”改为“壹”
list5[1] = "壹"

# 将（二，三，四）改为（2，3，4）
list5[2:5] = 2,3,4
```

小练习
test = [['python','C','C++','Java'],['Ruby','R','shell'],['go','Scala']] 1. 索引出C++，索引出R 2. 同时索引出（C++和 Java） 3. 把列表中的go改成大写 4. 把python和C的大小写进行交换

4.1.6 列表的常用方法

查	
list.index(x)	同str
list.count(x)	同str
list.sort()	对列表中的元素就地进行排序
list.reverse()	就地倒排列表中的元素

增	
list.append(x)	把一个元素添加到列表的结尾，相当于a[len(a):] = [x]
list.extend(x)	将一个给定列表中的所有元素都添加到另一个列表中
list.insert(i,x)	在指定位置插入一个元素

删	
list.clear()	从列表中删除所有元素，相当于del a[:]
list.remove(x)	删除列表中值为x的第一个元素。如果没有，就会返回一个错误。
list.pop([i])	pop up的缩写, 默认"弹出"最后一个元素, 并返回给调用者 可以通过参数index指定弹出元素的索引值 若不写参数，a.pop()返回最后一个元素。

改
用切片方法修改

注：insert，remove或者sort这些修改列表的方法没有打印返回值-它们返回None。在Python中对所有可变的数据类型这是统一的设计原则。

4.1.7 列表的拼接和重复

4.1.7.1 序列相加

可使用加法运算符来拼接序列。

```
[1,2,3]+[4,5,6]   # [1,2,3,4,5,6]

'Hello,'+ 'world!'   # 'Hello,world!'

[1,2,3]+'world!'
# TypeError: can only concatenate list (not "str") to list
```

从错误消息可知，不能拼接列表和字符串，虽然它们都是序列。一般而言，不能拼接不同类型的序列。

4.1.7.2 乘法

将序列与数n相乘时，将重复这个序列n次来创建一个新序列：

```
a = ['a','b','c']
a * 3
# ['a','b','c','a','b','c','a','b','c']
```

4.1.8 成员资格

检查特定值是否包含在序列中，我们可使用运算符in。它检查是否满足指定的条件，并返回相应的值：满足时返回True，不满足时返回False。

```
x = 'abcd'

'a' in a  # True
'e' in a  # False
```

```
users = ['Evan','John','Sherry']
input('Enter your user name:') in users

# Enter your user name: Evan
# True
```

4.1.9 列表的浅复制和深复制

方法copy复制列表，常规复制只是将另一个名称关联到列表。

```
a = [1,2,3]
b = a
b[1] = 4
a
# [1,4,3]
```

要让a和b指向不同的列表，就必须将b关联到a的副本。

```
a = [1,2,3]
b = a.copy()
b[1] = 4
a
# [1,2,3]
```

这类似于使用a[:]或list(a)，它们也都复制a。

```
a = [1,2,3]
b =a[:]
b
# [1,2,3]
```

4.1.10 删除元素

从列表中删除元素也很容易，只需要用del语句即可。

del 序列：从内存中删除：del list1。

但是，del list1[:]是删除数据。

4.1.11 列表的清空

方法clear就地清空列表的内容。

```
lst = [1,2,3]
lst.clear()
lst
# []
```

这类似于切片赋值语句lst[:] = []

4.1.12 列表推导式

```
[<表达式> for x1 in <序列1> [...for xN in <序列N> if <条件表达式>]]
```

列表推导式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。

4.1.12.1 一个简单的例子

如果我们想要生成一个由2, 4, 6...20的数字组成的列表,都有什么办法呢?

```
a = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

list(range(2, 22, 2))

L = []
for i in range(1,11):
    L.append(i*2)
```

另一种优雅的实现方式就是使用列表推导式（List Comprehensions）：

```
[i*2 for i in range(1,11)]
```

4.1.12.2 基本语法

我们具体来分析上面给出的例子:

我们将列表推导式写在一个方括号内，因为它最终构建的是一个列表。

列表推导式主要由两部分构成:

- 循环变量表达式（`i * 2`）
- for 循环头部（`for i in L`）

是不是非常简单!

4.1.12.3 工作原理

Python在执行列表推导式时，会对可迭代对象`L`进行迭代，将每一次迭代的值赋给循环变量`x`，然后收集变量表达式`i * 2`的计算结果，最终由这些结果构成了新的列表，也就是列表推导式所返回的值。

4.1.12.4 高级语法

- **带有 if 语句**

我们可以在`for`语句后面跟上一个`if`判断语句，用于过滤掉那些不满足条件的结果项。

实例:

```
square = [x**2 for x in range(10) if x!=5]
```

上面可以这么理解：先创建空列表，然后是for循环，里面做一个if判断，最后再把空列表做累加，也就是：

```
square = []
for x in range(10):
    if x!=5:
        lst.append(x**2)
```

- **带有for嵌套**

在复杂一点的列表推导式中，可以嵌套有多个 `for` 语句。按照从左至右的顺序，分别是外层循环到内层循环。

```
[x + y for x in 'ab' for y in 'jk']
```

- **更复杂的列表表达式：**

既有if语句又有for嵌套：

```
[[x,y] for x in [1,2,3] for y in [3,1,4] if x!=y]
```

再例如，下面的代码输出了0~4之间的偶数和奇数的组合。每一个 `for` 循环后面都有可选的 `if` 语句。

```
[(x, y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]
```

等价于下面的一般 `for` 循环：

```
L = []
for x in range(5):
    if x % 2 == 0:
        for y in range(5):
            if y % 2 == 1:
                L.append((x,y))
L
```

4.2 元组

Python的元组与列表类似，元组也是序列，唯一的差别在于元组是**不能修改**的。

4.2.1 元组的语法

元组语法很简单，只要将一些值用逗号分隔，就能自动创建一个元组。

```
1, 2, 3
# (1,2,3)
```

但我们通常采用圆括号的形式表示：

```
aTuple = (1,2,3)
aTuple
# (1,2,3)
```

那如何表示只包含一个值的元组呢？

```
tp = (1)
tp
# 1
```

这样定义出来的是数字，而不是元组。虽然只有一个值，也必须在它后面加上逗号。

```
tp1 = (1,)
tp1
# (1,)
```

4.2.2 函数tuple

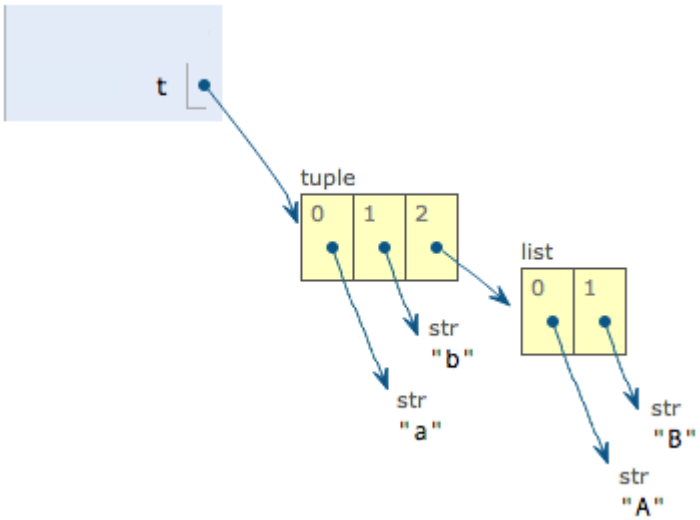
函数tuple的工作原理与list很像：它将一个序列作为参数，并将其转换为元组。如果参数已经是元组，那么就原封不动地返回它。

```
tuple([1,2,3])
# (1,2,3)
```

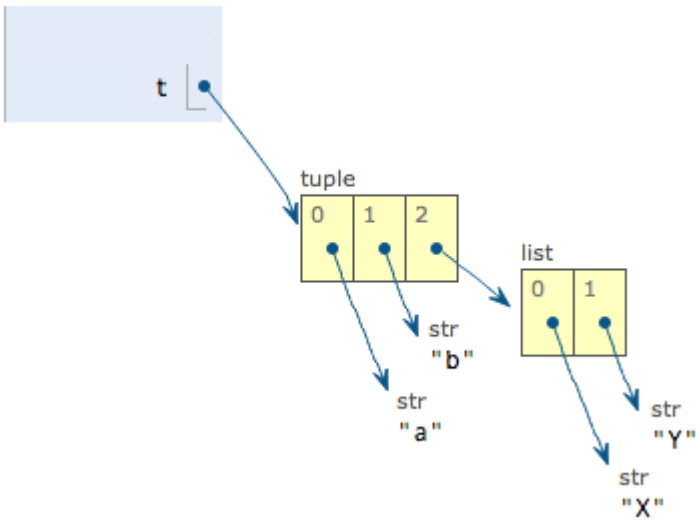
4.2.3 元组不可变的含义

```
tp2 = ('a', 'b', ['A', 'B'])

tp2[2][0] = 'X'
tp2[2][1] = 'Y'
tp2
# ('a','b',['X','Y'])
```



当我们把list的元素'A'和'B'修改为'X'和'Y'后，tuple变为：



表面上看，tuple的元素确实变了，但其实变得不是tuple的元素而是list的元素。tuple一开始指向的list并没有改成别的list，所以tuple所谓的 "不变" 是指，tuple的每个元素，指向永远不变。即指向'a'，就不能改成指向'b'，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！

4.2.4 元组的拼接和重复

同其他序列一样，元组的拼接用+号连接，元组的重复仍使用*n的方式。

4.2.5 元组的方法

元组的创建方式及其元素的访问方式与其他序列是相同的，但在元组中，**增删改是不被允许的。**

4.2.6 元组与列表区别

- 元组和列表很类似，它们经常被用来在不同的情况和不同的用途。
- 元组有很多用途。例如(x,y)坐标对，数据库中的员工记录等等。
- 元组就像是字符串，是不可变的。通常包含不同种类的元素并通过分拆或索引访问。

- 列表内容是可变的，它们的元素通常是相同类型的，并通过迭代访问。

对于某些领域来说，能用元组，就不用列表，因为；

- 元组速度更快，它是敞亮的。
- 使用元组不需要对修改的数据进行写保护，使得数据更安全。

4.3 集合

- 集合是一个**无序不重复**元素的集。
- 基本功能包括**关系测试**和**消除重复元素**。
- 集合对象还支持union（联合），intersection（交），difference（差）和sysmmetric difference（对称差集）等数学运算。

4.3.1 创建

4.3.1.1 符号创建

即用大括号{}创建：

```
set1 = {'a','b','c'}
```

但是需要注意，如果我们想要创建空集合，就不能单独使用{}，因为这是创建空字典的形式。我们应该使用函数创建方法进行实现。

4.3.1.2 函数创建

方法1：用set+字符串创建

```
set2 = set('abc')
```

方法2： 用set+列表/元组创建

```
set3 = set(['a','b','c'])
set4 = set(('a','b','c'))
```

创建空集合：

```
set5 = set(())
set6 = set([])
set7 = set({})
set8 = {}    # 这是字典创建的方式
```

Tips：如果创建不可修改集合用frozenset

4.3.2 基本操作

4.3.2.1 添加元素

语法：

```
set.add(x)
```

实例：

将元素x添加到集合s中，如果元素已存在，则不进行任何操作：

```
s = set(('Google','Facebook','Alibaba','Amazon'))
s.add('Baidu')
print(s)
```

- **还有一个方法，也可以添加元素，且参数可以是列表，元组，字典等，语法格式如下：**

```
set.update(x)
```

x可以有多个，用逗号分开：

```
s = set(('Google','Facebook','Alibaba','Amazon'))
s.update({1,2,3})
print(s)

s.update([1,2],[3,4])
print(s)
```

4.3.2.2 移除元素

语法：

```
set.remove(x)
```

实例：

将元素x从集合s中移除，如果元素不存在，则会发生错误：

```
s = set(('Google','Facebook','Alibaba','Amazon'))
s.remove('Amazon')
print(s)

s.remove('Baidu')
```

- **此外还有一个方法也是移除集合中的元素，且如果元素不存在，不会发生错误。格式如下所示：**

```
set.discard(x)
```

实例：

```
s = set(('Google','Facebook','Alibaba','Amazon'))
s.discard('Baidu')    # 不存在不会发生错误
print(s)
```

- **我们也可以设置随机删除集合中的一个元素，语法格式如下：**

```
set.pop()
```

实例：

```
s = set(('Google','Facebook','Alibaba','Amazon'))
x = s.pop()

print(x)
```

在交互模式，pop是删除集合的第一个元素（整理后的集合的第一个元素）。

```
s = set(('Google','Facebook','Alibaba','Amazon'))
s.pop()

print(s)
```

4.3.2.3 计算集合元素个数

语法：

```
len(set)
```

实例：

计算集合s元素个数：

```
s = set(('Google','Facebook','Alibaba','Amazon'))
len(s)
```

4.3.2.4 清空集合

语法：

```
set.clear()
```

实例：

```
s = set(('Google','Facebook','Alibaba','Amazon'))
s.clear()
print(s)
```

4.3.2.5 判断元素是否存在于集合中

语法：

```
x in s
```

实例：

判断元素x是否在集合s中，存在返回True，不存在返回False。

```
s = set(('Google','Facebook','Alibaba','Amazon'))
'Facebook' in s
'Baidu' in s
```

4.3.3 数学运算符所对应的集合内置方法

运算符	对应方法
s1 & s2 交集	s1.intersection(s2)
s1 | s2 并集	s1.union(s2)
s1 - s2 差集	s1.difference(s2)
s1 ^ s2 对称差	s1.symmetric_difference(s2)
s1 <= s2 是否是s2的子集	s1.issubset(s2)
s1 >= s2 是否是s2的超集	s1.issuperset(s2)
s1 |= s2 用s2更新s1	s1.update(s2)

4.3.3.1 set.intersection()

intersection() 方法用于返回两个或更多集合中都包含的元素，即交集。

语法：

```
set.intersection(set1,set2,...)
```

参数：

- set1：必需，代表要查找相同元素的集合。
- set2：可选，代表其他要查找相同元素的集合。这里可以是多个，但需要使用逗号隔开。

返回值：

返回一个新的集合。

实例：

返回一个新集合，该集合的元素既包含在集合x中，又包含在集合y中：

```
x = {'apple','banana','lemon'}
y = {'Google','Facebook','apple'}

z = x.intersection(y)

print(z)
```

计算多个集合的并集：

```
x = {'a','b','c'}
y = {'c','d','e'}
z = {'f','g','c'}

result = x.intersection(y,z)

print(result)
```

4.3.3.2 set.union()

union() 方法返回两个集合的并集，即包含了所有集合的元素，重复的元素只会出现一次。

语法：

```
set.union(set1,set2,...)
```

参数：

- set1：必需，代表合并的目标集合。
- set2：可选，代表其他要查找相同元素的集合。这里可以是多个，但需要使用逗号隔开。

返回值：

返回一个新集合。

实例：

合并两个集合，重复元素只会出现一次：

```
x = {'apple','banana','lemon'}
y = {'Google','Facebook','apple'}

z = x.union(y)

print(z)
```

合并多个集合：

```
x = {'a','b','c'}
y = {'c','d','e'}
z = {'f','g','c'}

result = x.union(y,z)

print(result)
```

4.3.3.3 set.difference()

difference() 方法用于返回集合的差集，即返回的集合元素包含在第一个集合中，但不包含在第二个集合（方法的参数）中。

语法：

```
set.difference(set)
```

参数：

- set：必需，用于计算差集的集合。

返回值：

返回一个新的集合。

实例：

返回一个集合，元素包含在集合x中，但不在集合y中：

```
x = {'apple','banana','lemon'}
y = {'Google','Facebook','apple'}

z = x.difference(y)

print(z)
```

4.3.3.4 set.symmetric_difference()

symmetric_difference() 方法返回两个集合中不重复的元素集合，即会**移除**两个集合中都存在的元素。

语法：

```
set.symmetric_difference(set)
```

参数：

- set：集合

返回值：

返回一个新的集合。

实例：

返回两个集合组成的新集合，但会移除两个集合的重复元素：

```
x = {'apple','banana','lemon'}
y = {'Google','Facebook','apple'}

z = x.symmetric_difference(y)

print(z)
```

4.3.3.5 set.issubset()

issubset() 方法用于判断集合的所有元素是否都包含在指定集合中，如果是则返回True，否则返回False。

语法：

```
set.issubset(set)
```

参数：

- set：必需，用来比较查找的集合。

返回值：

返回布尔值，如果都包含返回True，否则返回False。

实例：

判断集合x的所有元素是否都包含在集合y中：

```
x = {'a','b','c'}
y = {'f','e','d','c','b','a'}

z = x.issubset(y)

print(z)
```

如果没有全部包含返回False：

```
x = {'a','b','c'}
y = {'f','e','d','c','b'}

z = x.issubset(y)

print(z)
```

4.3.3.6 set.issuperset()

issuperset() 方法用于判断指定集合的所有元素是否都包含在原始的集合中，如果是则返回True，否则返回False。

语法：

```
set.issuperset(set)
```

参数：

- set：必需，用来比较查找的集合。

返回值：

返回布尔值，如果都包含返回True，否则返回False。

实例：

判断集合y的所有元素是否都包含在集合x中：

```
x = {'f','e','d','c','b','a'}
y = {'a','b','c'}

z = x.issuperset(y)

print(z)
```

如果没有全部包含返回False：

```
x = {'f','e','d','c','b'}
y = {'a','b','c'}

z = x.issuperset(y)

print(z)
```

4.3.3.7 set.update()

update() 方法用于修改当前集合，可以添加新的元素或集合到当前集合中，如果添加的元素在集合中已存在，则该元素只会出现一次，重复的会忽略。

语法：

```
set.update(set)
```

参数：

- set：必需，可以是元素或集合。

返回值：

无

实例：

合并两个集合，重复元素只会出现一次：

```
x = {'apple','banana','lemon'}
y = {'Google','Facebook','apple'}

x.update(y)

print(x)
```

4.3.4 其他内置方法

4.3.4.1 set.isdisjoint()

isdisjoint() 方法用于判断两个集合是否包含相同的元素，如果没有返回True，否则返回Fasle。

语法：

```
set.isdisjoint(set)
```

参数：

- set：必需，要比较的集合

返回值：

返回布尔值，如果不包含返回True，否则返回False。

实例：

判断集合y中是否有包含集合x的元素：

```
x = {'apple','banana','lemon'}
y = {'Google','Facebook','Amazon'}

z = x.isdisjoint(y)

print(z)
```

如果包含返回False：

```
x = {'apple','banana','lemon'}
y = {'Google','Facebook','apple'}

z = x.isdisjoint(y)

print(z)
```

4.3.4.2 set.symmetric_difference_update()

symmetric_difference_update() 方法移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中。

语法：

```
set.symmetric_difference_update(set)
```

参数：

- set：要检测的集合

返回值：

None

实例：

在原始集合x中移除与y集合中的重复元素，并将不重复的元素插入到集合x中：

```
x = {'apple','banana','lemon'}
y = {'Google','Facebook','apple'}

x.symmetric_difference_update(y)

print(x)
```

4.3.5 集合推导式

类似列表推导式，这里有一种集合推导式语法：

```
a = {x for x in 'abracadabra' if x not in 'abc'}
```

4.4 字典

- 字典是另一种可变容器模型，且可存储任意类型的对象。
- 字典是包含一个索引的集合，称为键和值的集合。

4.4.1 创建

方法一：

字典的每个**键值（key→value）对**用**冒号**分割，每个对之间用**逗号**分割，整个字典包括在花括号{}中：

```
d = {key1:value1,key2:value2}
```

其中键必须是唯一的，但值则不必。

值可以取任何数据类型，但键必须是不可变的，如字符串、数字和元组：

```
d = {'a':'1111','b':'2222','c':'3333'}
```

方法二：

将键/值对看成一个元素，格式为：

```
d = dict([(k1,v1),(k2,v2),(k3,v3)])
```

实例：

```
d = dict([('a','1111'),('b','2222'),('c','3333')])
```

方法三：

这种方法，键只能为字符串类型，并且不能加引号，格式为：

```
d = dict(k1=v1,k2=v2)
```

实例：

```
d = dict(a=1,b=2)
```

4.4.2 访问字典里的值

将相应的键放入到方括号中，如下所示：

```
d = {'Name':'Evan','Age':20,'Class':'First'}

print('Name:',d['Name'])
print('Age:',d['Age'])
```

如果用字典里没有的键访问数据，会输出错误如下：

```
d = {'Name':'Evan','Age':20,'Class':'First'}

print('Name:',d['Sherry'])

# KeyError: 'Sherry'
```

4.4.3 修改字典

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对如下实例：

```
d = {'Name':'Evan','Age':20,'Class':'First'}

d['Age'] = 22       # 更新Age
d['School'] = 'CDA数据分析研究院'    # 添加信息

print('Age:',d['Age'])
print('School:',d['School'])
```

4.4.4 删除字典元素

字典能删除单一的元素也能清空，删除一个字典用del命令，如下所示：

```
d = {'Name':'Evan','Age':20,'Class':'First'}

del d['Name']    # 删除键'Name'
d.clear()        # 清空字典
del d            # 删除字典

print('Age:',d['Age'])
```

执行del命令后，定义的字典就不存在于内存中了。

4.4.5 字典键的特性

我们知道字典的值可以任何的Python对象，但键不行。

- 创建时同一个键不允许出现两次。如果同一个键被赋值两次， 后一个值会被记住，如下所示：

```
d = {'Name':'Evan','Age':20,'Name':'Joe'}

print('Name:',d['Name'])
```

- 键必须不可变，所以可以用字符串、数字或元组充当，但不能用列表，如下实例：

```
d = {['Name']:'Evan','Age':20,'Name':'Joe'}
```

4.4.6 字典内置函数/方法

4.4.6.1 len(dict)

计算字典元素个数，即键的总数。

```
d = {'Name':'Evan','Age':20,'Class':'First'}
len(d)
```

4.4.6.2 str(dict)

输出字典，以可打印的字符串表示。

```
d = {'Name':'Evan','Age':20,'Class':'First'}
str(d)
```

4.4.6.3 dict.keys()

返回包含该字典键的列表：

```
d = {'Name':'Evan','Age':20,'Class':'First'}
d.keys()
```

4.4.6.4 dict.values()

返回包含该字典值的列表：

```
d = {'Name':'Evan','Age':20,'Class':'First'}
d.values()
```

4.4.6.5 dict.items()

将键/值对看成一个元素，并返回列表：

```
d = {'Name':'Evan','Age':20,'Class':'First'}
d.items()
```

4.4.6.6 dict.get(key, default=value)

如果key对象存在，则返回value，value默认是None。

```
d = {'Name':'Evan','Age':20,'Class':'First'}
d.get('Name')
```

4.4.6.7 dict.update()

顾名思义，update使用于更新字典的方法。

```
d = {'Name':'Evan','Age':20,'Class':'First'}
d1 = {'Name':'Evan','Age':27,'Height':188}
d.update(d1)
```

字典d1相比字典d，有两个重复的元素、一个key相同但值不同的元素，以及一个新元素，据此测试更新函数的更新效果。

Point:

- 更新原则是有新的部分则更换或添加新的部分，其他部分保留；
- 更新是有方向的，d.update(d1)和d1.update(d)有本质的区别：

- 更新过程会在原来对象基础上对其进行修改。

4.4.6.8 dict.setdefault方法

setdefault方法会根据键值对原本是否存在进行有选择性的修改。

setdefault方法使用过程中，第一个参数位输入备选key，第二个参数位置输入备选value：

- 若该备选key在原字典中存在，则返回该key对应的value，第二个参数位输入的value无任何作用，且**原字典不会被修改**；
- 若该备选key在原字典中存在，则返回该key对应的备选value，备选key-value将被添加至字典中，**原字典将会被修改**。

五、流程控制语句

在Python中，有三种控制流类型：

- 顺序结构
- 分支结构
- 循环结构

复杂的语句也都是由这三个基本的控制流组成的。

5.1 顺序结构

顺序结构就是普通的自上而下运行的代码结构。

```
a = '顺序结构'
print(a)
b = '自上而下'
print(b)
c = '逐条运行'
print(c)
```

5.2 分支结构

又称为条件控制语句。Python条件语句是通过一条或多条语句的执行结果（True或者False）来决定执行的代码块。

语句为：

```
if <条件判断1>:
    <执行语句块1>

elif <条件判断2>:
    <执行语句块2>

elif <条件判断3>:
    <执行语句块3>

else:
    <执行语句块4>
```

Tips：if语句可以独立使用。

例如，判断输入的年龄是否已成年：

```
# 单分支结构
age = 2
if age >= 18:
    print('your age is',age)
    print('adult')


# 搭配else使用：二分支结构
age = 3
if age >= 18:
    print('your age is',age)
    print('adult')
else:
    print('your age is',age)
    print('teenager')


# 搭配elif使用，新增判断条件：多分支结构
age = 7
if age >= 18:
    print('your age is',age)
    print('adult')
elif age >=6 :
        print('teenager')
else:
    print('kid')
```

5.3 循环控制语句

循环结构用来控制一段语句重复执行。

5.3.1 while循环

语句为：

```
while <条件判断>:
    <执行语句块1>
    # 完成后再次返回while

else:
    <执行语句块2>    # 在条件语句为False时执行else的语句块
    # else分可以省略
```

```
n = 5
while n > 0:
    print(n)
    n = n-1
```

```
n = 8
while n > 0:
    print('n为：',n,'不满足要求')
    n = n-1
else:
    print('n为：',n,'满足要求')
```

5.3.2 for循环

语句为：

```
for i in <可迭代对象>:
    <执行该语句块1>
else:
    <执行语句块2>  # else中的语句会在循环正常执行完（即for不是通过break跳出而中断的）的情况下执行
```

迭代对象可以是：字符串，对象，列表，元组，字典，迭代器，range等实现了迭代方法的对象。

```
name = ['Evan','Lucy','Henry','Sherry','Jessie']
for i in name:
    print('成员：',i)
else:
    print('Done!')
```

5.3.3 终止语句

5.3.3.1 Break语句用来终止最内层的循环

```
name = ['Evan','Lucy','Henry','Sherry','Jessie']
for i in name:
    print(i)
    break
```

```
name = ['Evan','Lucy','Henry','Sherry','Jessie']
n = 0
for i in name:
    print(i)
    n = n+1
    if n==3:
        break
```

5.3.3.2 continue用来跳过最内层当前次的循环

```
name = ['Evan','Lucy','Henry','Sherry','Jessie']
n = 0
for i in name:
    n = n+1
    if n==3:
        continue
    print(i)
```

```
n = 8
while n > 0:
    if n==5:
        n = n-1
        continue
    print(n)
    n = n-1
```

```
n = 0
while True:
    print(n)
    n = n+1
    if n == 10:
        break
```

5.3.4 占位语句

Tips：for循环常与range连用。

语法为：range(开始，结束，步长)

```
for i in range(4):


'''File "<ipython-input-2-908dfb2f9c4c>", line 2

    ^
SyntaxError: unexpected EOF while parsing'''
```

此时，我们的程序结构并非完整，而应该是：

```
for _ in range(1,100,20):
    print(_)
```

range的结果不能直接查看，需要**list(range(10))**或**[*range(10)]**

如果你不想返回任何结果，但又必须保证我们程序结构的完整性，这时候使用pass空语句：

```
for i in range(4):
    pass
```

当然定义函数也可以采用同样的方法：

```
def func():
    pass
```

占位语句不止在控制语句中使用，可以用到大多数的地方，用途是表示程序结构的完整，在编写预留功能或语句时方便简洁。

六、函数

6.1 懒惰是一种美德

当在一个地方编写了一些代码，但需要在另一个地方再次使用时，就可以使用**自定义函数**的方式将代码存储起来。

假设你编写了一段代码，它计算一些**斐波那契数**(一种数列，其中每个数都是前两个数的和)。

```
fibs = [0,1]
for i in range(8):
    fibs.append(fibs[-2] + fibs[-1])
```

运行上述代码后，fibs将包含前10个斐波那契数：

```
fibs

# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

我们也可以修改前述for循环，使其处理动态的范围，即让用户指定最终要得到的序列的长度。

```
fibs = [0, 1]
num = int(input('How many Fibonacci numbers do you want? '))
for i in range(num-2):
    fibs.append(fibs[-2] + fibs[-1])
    print(fibs)
```

如果要使用这些数字做其他事情，你可以在需要时再次编写这个循环，但试想每次使用都要重新修改，而程序优势非常复杂的类型，这种形式显然不能满足我们的需求了。

真正的程序员让程序更抽象。要让前面的程序更抽象，可以像下面这样做(输入12，将会打印以下结果)：

```
num = input('How many fibonacci numbers do you want?')
print (fibs)
```

在这里，只具体地编写了这个程序独特的部分(读取数字并打印结果)。实际上，斐波那契数的计算是以抽象的方式完成的:

- 我们只是让计算机这样做，而没有具体地告诉它如何做。
- 我们创建了一个名为fibs的函数，并在需要计算斐波那契数时调用它。
- 如果需要在多个地方计算斐波那契数，这样的方式很简单便捷。

6.2 浅谈抽象

抽象可节省人力，但实际上还有个更重要的优点:抽象是程序能够被人理解的关键所在。

- **e.g.**

  如果你向人打听怎么去电影院：

  - 就不希望对方回答: “向前走10步，向左转90度，接着走5步，再向右转45度，然后走123步。”
  - 而希望对象这样回答: “沿这条街往前走，看到过街天桥后走到马路对面，电影院就在你左边。”
  - 组织计算机程序时，也采取类似的方式。程序应非常抽象，如下载网页、计算使用频率、 打印每个单词的使用频率。
- 下面就将前述简单描述转换为一个Python程序。

```
page = download_page()
freqs = compute_frequencies(page)
for word, freq in freqs:
        print(word, freq)
```

看到这些代码，任何人都知道这个程序是做什么的。然而，至于具体该如何做，我们未置一词。 我们只是想让计算机去下载网页并计算使用频率，至于这些操作的具体细节，将在介绍自定义函数时详细的说明。

6.3 自定义函数

6.3.1 怎么理解函数

函数实际上是：

- 代码的一种组织形式
- 一个函数一般完成一项特定的任务
- 函数使用
    - 函数需要先定义
    - 使用函数，称之为调用
- 函数是组织好的，可以重复使用的，用来**实现单一**或者**相关联功能**的代码段。
- 函数能提高应用的**模块性**和代码的**重复利用率。**
- 其实我们已经接触到了很多python的**内建函数**，比如input(),print(),type()等等。
- 但自己创建函数也是熟练使用python必备的技能之一。这种**自建函数**也称作自定义函数。

6.3.2 构建规则

我们可以自己定义一个函数，但是需要遵循以下的规则：

- 函数的代码块以**def关键字**开头，后接函数标识符名称和圆括号（）。
- 圆括号用来存储要传入的**参数**和**变量**，这个参数可以是**默认**的也可以是**自定义**的。
- 函数的第一行语句可以选择性的使用文档字符串--用于存储文档说明。
- 函数内容以**冒号起始**，并且有**强制缩进。**
- return[表达式]结束函数。选择性的返回一个值给对方调用。不带表达式的return相当于返回None。

6.3.3 回忆一下内置函数的调用

```
import math
x = 1
y = math.sqrt
callable(x)
callable(y)
```

这里的 callable() 方法用来检测对象是否可被调用，可被调用指的是对象能否使用()括号的方法调用。

从返回结果来看，x是不可调用对象，所以调用肯定不成功；y是可调用对象，结果返回True。

6.3.4 使用def语句定义函数

定义函数的格式如下：

```
def 函数名():
    代码
```

先来定义一个函数，让它能够完成打印信息的功能：

```
def printInfo():
    print ('-------------------------------------')
    print ('         人生苦短，我用Python')
    print ('-------------------------------------')

printInfo()
```

再来看个例子：

```
def hello(name):
    return 'Hello, ' + name + '!'
```

上述代码实际上是定义了一个名为hello的新函数。它返回一个字符串，其中包含向唯一参数指定的人发出的问候语。

我们可像使用内置函数那样使用这个函数：

```
print(hello('world'))
print(hello('Evan'))
```

下面我们将之前用循环写成的**斐波那契数列**改成自定义函数的形式：

```
def fibs(num):
    result = [0, 1]
    for i in range(num-2):
        result.append(result[-2] + result[-1])
    return result
```

现在直接调用函数fibs就可以实现我们想要的功能啦：

```
fibs(15)
```

在这个示例中，num和result也可以使用其他名字，但return语句非常重要。return语句用于从函数返回值(在前面的hello函数中，return语句的作用也是一样的)。

小练习

要求：定义一个函数，能够输出自己的姓名和年龄，并且调用这个函数让它执行。

6.3.5 给函数编写说明文档

还有另一种编写注释的方式，就是在def语句后面，添加一段说明字符串很有用。我们将这种放在函数开头的字符串称为文档字符串(docstring)，将作为函数的一部分存储起来。

为确保其他人能够理解程序，要给函数编写文档，可以通过以下两种方式：

- 添加注释(以#打头的内容)：
- 添加独立的字符串

下面的代码演示了如何给函数添加文档字符串：

```
def test(a,b):
    """用来完成对2个数求和
    参数:
    a:我们的第一个参数用来相加
    b:这是相加的第二个参数
    return
    很抱歉，这个函数没有返回值"""
    print(a+b)
```

访问说明文档和文档字符串可以通过以下形式：

```
test?
```

```
print(test.__doc__)
```

注意： doc 是函数的一个属性。属性名中的双下划线表示这是一个特殊的属性。

特殊的内置函数help在交互式解释器中，可使帮助我们获取有关函数的信息，其中包含函数的文档字符串。

```
help(test)
```

6.3.6 偶遇参数

思考一个问题
现在需要定义一个函数，这个函数能够完成2个数的加法运算，并且把结果打印出来，该怎样设计？下面的代码可以吗？有什么缺陷吗？

```
def add2num():
   a = 11
   b = 22
   c = a+b
   print (c)
```

为了让一个函数更通用，即想让它计算哪两个数的和，就让它计算哪两个数的和，在定义函数的时候可以让函数**接收数据**，就解决了这个问题，这就是**函数的参数。**

```
def add2num(a, b):
    c = a+b
    print (c)
```

小练习
1. 定义一个函数，完成前2个数完成加法运算，然后对第3个数，进行减法；然后调用这个函数。
2. 完成一个函数, 功能和abs相同。

6.3.7 函数的返回值

6.3.7.1 return（返回值）的作用

定义了一个函数，完成了获取操作，那么函数应该把获取值返回给调用函数的人，只有调用者拥有了这个返回值，才能够根据当前的值做适当的调整。

```
def abs_me_2(x):
    if x >= 0:
        return (x)
    else:
        return (-x)
```

6.3.7.2 return的注意事项

在函数中, 一旦运行到return 这行代码, 则函数退出。

```
def abs_me_3(x):
    if x >= 0:
        print('return前面的代码')
        return (x)
        print('这行代码有没有被运行呢?')
    else:
        return (-x)
        print('else这行代码有没有被运行呢?')
    print('这行代码呢?')
```

如你所见，跳过了第二条print语句。(这有点像在循环中使用break，但跳出的是函数。)

6.3.7.3 返回多个值

```
def test(a, b):
    c = a + b
    d = a * b
    e = a / b
    return (c, d, e)
```

我们可以直接执行函数以返回结果：

```
test(3,2)

# (5, 6, 1.5)
```

也可以用三个变量依次接收返回的三个结果：

```
c,d,e = test(3,2)

# 相当于c, d, e = 5, 6, 1.5
```

小练习
1. 完成一个函数功能和sum相同
2. 写一个函数可以实现divmod的功能

6.4 函数的参数详解

6.4.1 值从哪里来

定义函数时，你可能心存疑虑：参数的值是怎么来的呢?

大可不用为值从何来而费心。编写函数旨在为当前程序(甚至其他程序)提供服务，而我们的职责是确保它在提供的参数正确时完成任务，并在参数不对时通过**异常**和**断言**进行体现。

6.4.2 形参和实参

在def语句中，位于函数名后面的变量通常称为**形参**，而调用函数时提供的值称为**实参**。在很重要的情况下，我会将实参称为**值**，以便将其与类似于变量的形参区分开来。

```
def test(x): # 这里的x作为一个变量存在，并不是函数运行时候的真实值
    return x + 1
```

```
test(5) # 这里的5，是真正传入进入函数的参数
```

6.4.3 位置参数和关键字参数

前面我们使用的参数都是**位置参数**，因为它们的位置至关重要——事实上比名称还重要。

我们可以来看下面这个函数：

```
def test(a, b, c):
    print('a的值是:', a)
    print('b的值是:', b)
    print('c的值是:', c)
```

我们可以直接按照参数的顺序赋值，但有时候，参数的排列顺序可能难以记住，尤其是参数很多时。为了简化调用工作，可指定参数的名称。

```
test(1, 2, 3)

test(a=1, b=2, c=3)

test(c=3, b=2, a=1)/
```

6.4.4 调用函数时参数的顺序

- **关键字参数必须放在位置参数后面**

我们可结合使用位置参数和关键字参数，但必须先指定所有的位置参数，否则解释器将不知道它们是哪个参数(即不知道参数对应的位置)。

还是以刚才的函数为例，我们可以这样指定：

```
test(1, 2, c=3)
```

但下面的指定方式就会出现错误：

```
test(a=1, 2, 3)

test(1, a=2, 3)

test(b=2, a=1)
```

注意：通常不应结合使用位置参数和关键字参数，除非我们已经知道这样做的后果。一般而言，除非必不可少的参数很少，而带**默认值的可选参数很多**，否则不应结合使用关键字参数和位置参数。

所以我们可以发现关键字参数的一些优势：

- 使用名称指定的参数，有助于澄清各个参数的作用。这样，函数调用不再像下面这样怪异而神秘。
- 使用关键字参数使每个参数的作用清晰明了。另外，参数的顺序错了也没关系。
- 另外，关键字参数最大的优点还在于可以指定默认值。

6.4.5 默认参数

先来编写一个求幂值的函数：

```
def power(x, y):
    n = 1
    i = 0
    while i < y:
        n *= x
        i += 1
    return n
```

因为我们大多数时候,都是需要求平方, 因此后面的参数取2的可能性最大：

```
def power(x, y=2): # y的默认值是2
    n = 1
    i = 0
    while i < y:
        n *= x
        i += 1
    return n
```

像这样给参数指定默认值后，调用函数时可不提供它！可以根据需要：

- 一个参数值也不提供
- 提供部分参数值

- 提供全部参数值

注意：

- 必选参数在前，默认参数在后，否则会报错。
- 默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

6.4.6 收集参数

6.4.6.1 一个星号

有时候，允许用户提供任意数量的参数很有用。例如在下面的例子中，我想存储多个姓名，这样做会更加符合实际应用场景：

```
store(data, name1, name2, name3)
```

这时候我们要允许提供任意数量的姓名，我们可以像下面这样对函数进行定义：

```
def print_params(*params):
    print(params)
```

我们在这里虽然看起来只指定了一个参数，但会发现在参数前面有个**星号**，这时候参数变为可变长参数，函数可以接收1个、2个到任意个，还可以是0个值作为参数。

在例子中看看传入这类参数的函数的使用和调用：

```
print_params('Python')

# ('Python',)
```

这里打印的是一个元组，因为里面有一个逗号。这样的话是不是说，前面有星号的参数将被放在元组中呢？来试试复数的params：

```
print_params(1, 2, 3)
```

从结果可以看出参数前面的星号将提供的所有值都放在一个元组中，也就是将这些值收集起来。

再来编写一个函数：

```
def print_params_1(title, *params):
    print(title)
    print(params)
```

并尝试调用它：

```
print_params_2('Params:', 1, 2, 3)
```

所以，星号意味着收集余下的位置参数。如果没有可供收集的参数，params将返回空元组。

```
print_params_2('Params:')
```

还记得之前我们写的一个函数，用来计算列表的加和么？现在我们想不需要传入列表，同样实现这样一个功能，可以传入任意个参数，然后计算参数的加和：

```
def sum_me(x_list):
    n = 0
    for i in x_list:
        n += i
    return n
```

现在我们想这样做：

```
sum_me(2, 3, 4, 5, 6, 7)
```

但现在的函数不允许，这时候我们在参数前面加个星号：

```
def sum_you(*x_list):
    n = 0
    for i in x_list:
        n += i
    return n
```

6.4.6.2 两个星号

带星号的参数可放在其他位置，而不是一定要放在最后，但在这种情况下我们需要做些额外的工作：

```
def test(x, *y, z):
    print(x, y, z)
```

```
test(1,2,3,4,5,z=6)
# 1 (2, 3, 4, 5) 7

test(1,2,3,4,5,6)
# 报错
```

此时，一个星号不会收集关键字参数，要收集关键字参数，可使用**两个星号。**

```
def test_1(**params):
    print(params)
```

```
test_1(a=1, b=2, c='qqq', d=True)
```

一个星号就是打包成元组传入进来，两个星号就是打包成字典传入进来。

这样，我们得到的就是一个字典而不是元组。

那么如果普通参数，可变长参数，可变长关键字参数结合在一起定义会怎么样呢?

```
def test_2(a, b, *args, **kwagrs):
    """可变参数演示示例"""
    print ("a =", a)
    print ("b =", b)
    print ("args =", args)
    print ("kwargs: ")
    print(kwargs)
```

我们试试加入实参:

```
test_2(1,'xxx')
test_2(1,'xxx',2,3)
test_2(1,'xxx',2,3,4,5,6,7,100,1000,'asdfg')
test_2(1,2,3,4,5,m=6,n=7,p=8)
test_2(1,2,3,m=6,n=7,p=8,4,5)
```

注意: 这里同样遵循关键字参数必须在位置参数之后的顺序。

小练习
给定一组数字a，b，c……，请计算$a^2 + b^2 + c^2 + \ldots$

6.5 作用域

函数变量的作用域其实就是平时我们所说的**变量可见性。** 在python当中，程序的变量并不是在任何位置都能被访问，访问权限取决于变量是在什么位置赋值的。 变量的作用域决定了变量在哪一部分程序可以被访问。 **python的作用域一共有4种**:

```
- L （Local）局部作用域
- E （Enclosing）闭包函数外的函数中
- G （Global）全局作用域
- B （Built-in）内建作用域
```

以 L --> E --> G --> B 的规则查找：

即在局部找不到，则会去局部外的局部找，比如闭包。再找不到，就会去全局找，再者则会去内建中找。

- 内建作用域

```
x = int(2.9)
```

- 全局作用域

```
y = 2
```

- 闭包和局部作用域

```
def outer(n):
    o_count = n     # 闭包外的局部作用域

    def inner():
        return o_count + 1     # 闭包的局部作用域

    return inner()

print(outer(10))
```

但你直接调用o_count是不行的：

```
o_count

# NameError: name 'o_count' is not defined
```

python中只有**模块（module），类(class)以及函数(def ,lamda)**才会引入新的作用域，其他的代码块（如**if/else/elif/、for/while、try/except**等）是不会引入新的作用域的，也就是说这些语句内定义的变量，外部也可以访问，如下代码：

```
if True:
    ...
    x = 'I am from CDA'
    ...
```

```
x
```

实例中 **x变量定义在 if 语句块中**，但外部还是**可以访问**的。

如果将 **x定义在函数中**，则它就是局部变量，外部**不能访问**：

```
def test():
    ...
    x_inner = 'I am from Python'
    ...
    return x_inner
```

这里报错的原因在于：对于函数外部而言，x_inner没有被定义过，也就是说Python找不到x_inner这个变量。这是因为x_inner只是一个**局部变量**，它的作用范围只在它的地盘上 "test()" 函数的定义范围内有效，出了这个范围，就不属于x_inner的作用域了，它将不起任何作用。

总结一下，在函数里面定义的参数以及变量，都称为局部变量，出了这个函数，这些变量就是无效的。事实上的原理是，Python在运行函数的时候，利用栈（Stack）进行存储，当执行完函数之后，函数中所有的数据都会被自动删除。所以在函数外边是无法访问到函数内部的局部变量的。

6.6 全局变量和局部变量

定义在**函数内部**的变量拥有一个**局部作用域**，定义在**函数外**的拥有**全局作用域。 局部变量**只能在其被声明的**函数内部**访问，而**全局变量**可以在**整个程序**范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。如下实例：

```
total = 0 # 这是一个全局变量
# 返回2个参数的和
def sum(arg1,arg2):
    total = arg1 + arg2    # total 在这里就是局部变量
    print('函数内是局部变量: ',total)
    return total

#调用sum函数
sum(11,22)
print("函数外是全局变量: ",total)
```

- 如果全局变量的名字和局部变量的名字相同，那么使用的是局部变量的，小技巧：**强龙不压地头蛇**

```
# 定义全局变量
a = 100
```

```
def test_3():
    a = 300
    print("test_3中a是: ", a)
```

```
def test_4():
    print("test_4中a是: ", a)
```

注意

全局变量在整个代码段中都是可以访问到的，但是不要试图在函数内部去修改全局变量。因为全局变量的修改实际上是重新定义了一个新变量，内存地址发生了改变。而在局部对全局变量进行修改实际上是创建了与全局变量相同的局部变量去代替全局变量。那么全局变量在局部就没有任何意义了。

总结一下:

- 在函数外边定义的变量叫做全局变量
- 全局变量能够在所有的函数中进行访问
- 如果在函数中修改全局变量，那么就需要使用global进行声明，否则出错
- 如果全局变量的名字和局部变量的名字相同，那么使用的是局部变量的，小技巧：强龙不压地头蛇

6.7 global关键字

小姐姐问：一个自然而然的需求, 能不能在函数内部修改全局变量呢?

小哥哥答：函数内部可以调用全局变量的值, 但是不能修改哟~

```
a = 1
def f():
    a += 1
    print (a)
f()
```

通过上述例子，我们可以发现，函数中修改全局变量可能导致程序可读性变差、出现莫名其妙的bug、代码的维护成本高。因此不建议在函数内部修改全局变量。

如果小姐姐非要在函数内部修改全局变量, 怎么办呢? 小哥哥可以这样做：

```
# 定义全局变量
a = 1
def f():
    global a  # 需要使用全局变量用global关键字说明
    a += 1
    print (a)
f()
```

其实，我们在函数中不使用global声明全局变量时就不能修改全局变量的本质是不能修改全局变量的指向，即不能将全局变量指向新的数据。

对于**不可变类型的全局变量**来说，因其指向的数据不能修改，所以不使用global时无法修改全局变量。

对于**可变类型的全局变量**来说，因其指向的数据可以修改，所以不使用global时也可修改全局变量。

比如下面这段代码执行后显然是会报错的，因为我没有声明：

```
a = 1 # 不可变的
def f():
    a += 1
    print (a)
f()
```

现在我们把a变成可变序列：

```
li = []
def f(x):
    li.append(x)
```

```
li = [1,]
def f2():
    li.append(1)
    print (li)
f2()
```

6.8 内嵌函数和nonlocal关键字

6.8.1 内嵌函数

Python的函数定义可以是嵌套的，也就是允许在函数内部创建另一个函数，这种函数叫做内嵌函数或内部函数。

```
def func_1():
    print("func_1()正在被调用...")

    def func_2():
        print("func_2()正在被调用...")
    func_2()
func_1()
```

这是函数嵌套最简单的例子。关于内部函数的使用，有一个比较值得注意的地方，就是**内部函数整个作用域都在外部函数之内。**就像例子中的func_2()整个函数的作用域都在func_1()里面。

需要注意的地方是，除了在func_1()这个函数可以调用func_2()这个内部函数，出了func_1()就没有地方可以调用func_2() 了。如果非要尝试，那报错是在所难免的了~

6.8.2 nonlocal关键字

同样，想要在内嵌的函数局部更改局部外局部的变量，就需要用到nonlocbal关键字：

```
def outer():
    num = 10
    def inner():
        nonlocal num   # nonlocal关键字声明
        num = 100
        print(num)
    inner()
    print(num)
outer()
```

6.9 闭包

在作用域小节中，我们提到了闭包这一名词。

闭包是函数式编程的重要语法结构，Pyhon中的闭包从形式上定义为：如果在一个内部函数里，对在外部作用域（但不是全局作用域）的变量进行引用，那么内部函数就被认为是闭包。

所以当时我们说闭包的作用域是闭包函数外（但不是全局）的函数中（外部函数中）。

```
def funcX(x):
    def funcY(y):
        return x *y
    return funcY
```

```
i = funcX(8)
i(5)
```

```
funcX(8)(5)
```

如果在一个内部函数里（funcY）对外部作用域的，但不是全局作用域的变量进行引用，这里特指x（它在funcX区域中，但不在全局），这个内部函数funcY就是一个闭包。

需要注意的是，funcY也是定义在函数内部的函数，它的概念就是由嵌套函数的内部函数概念引申而来，因此，内部函数也不能在外部调用。

6.10 递归

递归属于比较高级的范畴，但是对于想要写出漂亮程序的程序员来说，是一个非常好的编程思路。生活中有很多递归的例子：比如汉诺塔游戏、结构树的定义、谢尔宾斯基三角形、女神自拍。

前边我们已经学过函数的嵌套了。那么一个函数内部可以嵌套其他函数, 函数内部如果再**嵌套一个函数自己**, 就是我们所说的递归函数了。

在程序上，**递归**实质上是**函数调用自身的行为**。

```
def recursion():
    return recursion()

recursion()
```

显然，这样去定义是毫无意义的，虽然与刚才的 “递归” 定义一样。但在运行一段时间后，这个程序崩溃了（出现了异常）。

从理论上说，这个程序将不断运行下去，但每次调用函数时，都将消耗内存。因此，在函数调用次数达到一定程度后（且之前的函数调用未返回），将耗尽所有的内存空间，导致程序终止并显示错误消息 “超过最大递归深度” 。

在这个函数中的递归成为**无穷递归**（类似 while True 打头，并且不包含 break 和 return 语句的循环，我们成为无限循环一样），从理论来看它永远不会结束。而我们希望的是有意义的递归函数，这样的递归函数通常包含以下两个部分。

- **基线条件**：针对最小的问题，满足这种条件时函数将直接返回一个值。
- **递归条件**：包含一个或者多个调用，这些调用旨在解决问题的一部分。

这里的关键点是，我们通过将问题分解为较小的部分，避免递归没完没了，因此问题终将被分解成基线条件可以解决的最小问题。

> **如何理解函数调用自身?**
>
> 每次调用函数时，都将为此创建一个新的命名空间。这就意味着函数在调用自身时，是两个不同的函数。
> 我们可以把它理解为，是两个不同的版本，即命名空间不同的同一函数在做交流。我们也可以将其视为两个属于相同物种的动物在彼此交流。

6.10.1 阶乘和幂

一起来计算数字n的阶乘：$n! = 1 * 2 * 3 * ... * n$

我们可以使用循环的方式：

```
def factorial(n):
    result = 1
    for i in range(1,n+1):
        result *= i
    return result
```

或者我反向相乘，先将result设置为n，再将其依次乘以1到n-1的每个数字，最后返回result。

```
def factorial(n):
    result = n
    for i in range(1, n):
        result *= i
    return result
```

我们可以很容易发现**阶乘的规律**：

```
1! = 1
2! = 2 × 1 = 2 × 1!
3! = 3 × 2 × 1 = 3 × 2!
4! = 4 × 3 × 2 × 1 = 4 × 3!
...
n! = n × (n-1)!
```

下面来考虑如何使用递归函数来实现这个定义。理解这个定义后，实现起来其实非常简单。

```
def calnum(num):
    if num > 1:
        return num * calnum(num-1)
    else:
        return 1
```

别忘了函数调用calnum(num)和calnum(num – 1)是不同的实体。

小练习
用递归实现求幂函数，就像内置函数**pow**和运算符**所做的那样

递归函数与循环结构相比，在大多数情况下，使用循环的效率可能更高。然而，在很多情况下，使用递归的代码**可读性更高**，且有时要高得多，尤其对递归的定义有了深入理解之后。另外，虽然我们完全能够避免编写递归函数，但作为程序员，我们必须能够读懂其他人编写的递归算法和函数。

6.11 lambda匿名函数

6.11.1 lambda的作用

lambda能创建一个匿名函数，程序员觉得每次创建一个函数，即使是很简单的函数也需要去起名太麻烦了。

于是就有了lambda，它可以直接写进一个需要简单函数的地方，而不需要单独去进行定义一个函数。

6.11.2 lambda的标准语法

用**lambda关键词**能创建小型匿名函数，这种函数得名于省略了用def声明函数的标准步骤。

lambda函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

这里有个小函数：

```
def fun(x):
    return x + 1
```

用lambda函数格式如下：

```
f = lambda x:x+1
```

根据语法，我们可以发现lambda同样可以接受**多个参数**：

```
k = lambda x, y, z:(x + y - z)
```

同样，加入**条件判断语句**也能实现我们想要的结果：

```
d = lambda x:x+1 if x > 10 else x - 1
```

但此时，不能省略else，判断语句也要写完整：

```
d = lambda x:x+1 if x > 10 # 报错，if else都必须写
```

然而，这种简单的句法限制了 lambda 函数的定义体只能使用纯表达式。换句话说，lambda 函数的定义体中**不能赋值，也不能使用 while和 try** 等 Python 语句。

6.11.3 lambda函数的使用场景

应用场景：函数作为参数进行传递

- Lambda函数能接收任何数量的参数，但只能返回一个表达式的值
- 匿名函数不能直接调用print，因为lambda需要一个表达式

6.11.3.1 把函数功能屏蔽

比如，创建一个让程序休眠的程序：

```
import time

time.sleep(3) # 让程序休眠
print('------zZZ-------')
```

```
active = True
while active:
    time.sleep(3)
    print('开始运行')
```

用lambda把这个休眠功能屏蔽掉：

```
time.sleep = lambda x : None
```

6.11.3.2 作为高级函数的参数

我们使用Python中几个定义好的全局函数：**map，filter，sorted**

- **map**

```
map(self, /, *args, **kwargs)
```

创建一个迭代器，该迭代器使用来自每个可迭代项的参数计算函数。

简单来说，**map()** 会根据提供的函数对指定序列做**映射**。第一个参数 function 以参数序列中的每一个元素调用 function 函数，返回包含每次 function 函数返回值的新列表。

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
list(map(lambda x:x*2+10, a))
```

map() 函数接收两个参数，一个是函数，一个是iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的iterator返回。

由于结果是一个iterator，而iterator是惰性序列，因此需要通过list() 函数让它把整个序列都计算出来并返回一个list。

- **filter**

```
filter(self, /, *args, **kwargs)
```

Python内建的filter() 函数用于过滤序列。

和map() 类似，filter() 也接收一个函数和一个序列。但有一点不同，filter() 把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

```
def f(x):
    if x % 2==0:
        return x

list(filter(f,a))
```

我们用lambda实现：

```
list(filter(lambda x:x if x % 2 == 0 else None, a))
```

再简化一下：

```
list(filter(lambda x:x%2==0, a))
```

- **sorted**

```
sorted(iterable, /, *, key=None, reverse=False)
```

sorted() 函数也是一个高阶函数，对所有可迭代的对象进行排序操作。

它还可以接收一个key函数来实现自定义的排序，例如按绝对值大小排序。

```
a = [2, 5, 6, 1, 3, 4, 55, 22, 77, 8, 1, 2, 3, 4, 10]

sorted(a)
sorted(a,reverse=True)
sorted(a, key=lambda x:x%2==1)
sorted(a, key=lambda x: abs(5 - x))
```

6.12 小结

我们介绍了抽象的基本知识以及函数：

抽象：抽象是隐藏不必要细节的艺术。通过定义处理细节的函数，可让程序更抽象。

函数定义：函数是使用**def语句**定义的。函数由语句块组成，它们从外部接收值(参数)，并可能返回一个或多个值(计算结果)。

参数：函数通过参数(调用函数时被设置的变量)接收所需的信息。在Python中，参数有两类：位置参数和关键字参数。通过给参数指定默认值，可使其变成可选的。

作用域：变量存储在作用域(也叫命名空间)中。在Python中，作用域分两大类:全局作用域和局部作用域。作用域可以嵌套。

递归：函数可调用自身，这称为递归。可使用递归完成的任何任务都可使用循环来完成，但有时使用递归函数的可读性更高。

函数式编程：Python提供了一些函数式编程工具，其中包括lambda表达式以及函数map、filter和sorted。