Day 1/13

**整数**

整数和我们生活中定义的整数是一个概念，例如：1,2,3,10,100,1000,-1000 等等，都是整数，Python可以处理任意大小的整数。

对于整数，除了生活中常用的十进制以外，计算机由于使用二进制，所以，在计算机中也常用二进制或者十六进制表示整数。相比十进制数，在Python中使用以下方式表示二进制数和十六进制数。

二进制数只使用数字0、1表示，在Python中，二进制整数使用前缀0b表示，比如：0b0110，0b1100。

十六进制数除了0~9十个数字以外，还使用a、b、c、d、e、f，在Python中，十六进制使用前缀0x，比如：0x12ef，0xde2431af。

十进制数、二进制数和十六进制数之间是可以相互转换的，如果感兴趣的同学，可以学习[《编程必备基础知识》](https://coding.imooc.com/learn/list/355.html)了解更多转换的细节。

**浮点数**

浮点数对应我们生活的是小数，例如：0.1,0.5, 12.34, 3.1415926等，都是浮点数。

为什么小数在计算机领域被称为浮点数呢？这是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的（浮动的），比如，1.23x10^9和12.3x10^8是相等的，因此称为浮点数。

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，就必须用科学计数法表示，在Python中，把10用e替代，比如：1.23x10^9就是1.23e9，或者12.3e8，0.000012可以写成1.2e-5，同学可以自行举出更多的例子。

这里有必要提醒一点就是，整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的，而浮点数运算则可能会有四舍五入的误差，如何检验，可以在Python终端中输入以下内容0.1 + 0.2，看看可以得到你期望的结果0.3吗？真实结果可能让你失望了。

  
同样的，这里涉及的原理同学也可以学习上面提到的课程进行进一步原理层面的理解。

**字符串**

字符串对应生活中的就是非数字类型的内容，比如一个句子，一段话，在Python中就是字符串，在Python中，字符串是以''或""括起来的任意文本，比如'abc'，"xyz"等等。请注意，''或""本身只是一种表示方式，不是字符串的一部分，因此，字符串'abc'只有a，b，c这3个字符。

比如之前练习过的第一个Python程序：print('Hello World')，其中的Hello World就是字符串。

**布尔值**

布尔值对应于生活中的就是对和错，在计算机的世界里，大部分判断都是非错则对的，布尔值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，在Python中，可以直接用True、False表示布尔值（请注意大小写，不需要使用字符串符号括起来），也可以通过布尔运算计算出来。

布尔值可以用and、or和not运算（注意and,or,not都是Python语言本身的关键字）。

and运算是与运算，只有所有都为 True，and运算结果才是 True。

or运算是或运算，只要其中有一个为 True，or 运算结果就是 True。

not运算是非运算，它是一个单目运算符，把 True 变成 False，False 变成 True。

**空值**

空值是Python里一个特殊的值，用None表示。

注意，None和0是不一样的，None不能理解为0，因为0是有意义的，比如我有0个苹果表示我没有苹果，0表达的是数值0的意思，而不能表示为我有None个苹果，None是一个特殊的空值。

3-2

**Python定义变量的方法**

**什么是变量**

在Python中，变量的概念基本上和初中代数的方程变量是一致的。例如，对于方程式 y=x\*x ，x就是变量。当x=2时，计算结果是4，当x=5时，计算结果是25。

**合法的变量名**

只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型（字符串、布尔值、空值、字典等）。在Python中，定义一个变量需要遵循一定的约束，否则，Python可能识别不出它是一个变量。

1. 变量名由大小写英文字母、数字和下划线\_组成
2. 变量不能用数字开头
3. 变量尽量不要和Python关键字重合（比如前面学习过的：and、or、not，否则可能导致Python原有关键字发挥不出作用）

以下这些变量的定义都是合法的。

num, count, \_none, min\_value

他们都满足上面的三个条件。

以下这些变量的定义都是不合法的。

1num, 666, 1\_cd, and

**定义变量**

定义变量的方式很简单，通过变量名 = 数据，即可定义一个变量。  
举个例子：

a = 1

在这个例子里面，a就是变量，它的值是一个整数1。

hello = 'Hello'

在这个例子里面，hello就是变量，它的值是一个字符串'Hello'。  
在Python里面，一个变量可以先后存储多种不同类型的数据。

a = 1 # 这个时候a存储的是整数类型

print(a)

a = 'ABC' # 这个时候a存储的是字符串类型

print(a)

这是Python这类语言特有的特性，我们称之为动态语言，与之对应的是静态语言，Python、Javascript等等都是动态语言，Java、C、C++等等属于静态语言。

3.3

从上面可以发现一个规律，整数和浮点数运算后 ，得到的结果不管小数点后是否有值，结果都变成浮点数了，这是合理的，浮点数可以表达整数的结果，但是整数不能表达浮点数的结果。

**注意：在Python2使用除法可能和Python3得到不一样的结果**

# python2

num1 = 10

num2 = 3

result = num1 / num2

print(result) # ==> 3

# python3

num1 = 10

num2 = 3

result = num1 / num2

print(result) # ==> 3.3333333333333335

可以看到在python2，得到的是一个整数的结果，这是因为除数和被除数都是整数时，得到的结果也默认保存为整数了，这是非常不科学的，因此在python3，改进了这一点。

### ****取模运算****

Python数字支持取模运算，使用百分号%表示取模。

print(3 % 2) # ==> 1

print(33 % 10) # ==> 3

print(99 % 30) # ==> 9

恰当使用取模运算，可以判断一个数是否为偶数，当一个数对2取模结果为0时，则这个数为偶数，否则为奇数。

print(3 % 2) # ==> 1 因此3为奇数

print(33 % 2) # ==> 1 因此33为奇数

print(100 % 2) # ==> 0 因此100为偶数

### ****地板除****

Python除了普通除法以外，还有一个特殊的除法被称为地板除，对于地板除，得到的结果会忽略纯小数的部分，得到整数的部分，地板除使用//进行。

10//4 # ==> 2

10//2.5 # ==> 4.0

10//3 # ==> 3

### ****小数点位数****

使用Python计算小数的时候，经常需要保留小数点后若干位，可以使用round()函数来处理，这里先了解round的调用方式，使用两个参数，第一个是需要保留小数点位数的数值，第二个是保留的位数。

num = 10 / 3

print(num) # ==> 3.3333333333333335

# 使用round保留两位小数

round(num, 2) # ==> 3.33

3-4

把True变为False，或者把False变为True：

not True # ==> False

not False # ==> True

这些运算有什么用呢？计算机程序是由无数的逻辑分支组成的，通过布尔运算，可以在计算机中实现条件判断，根据计算结果为True或者False，计算机可以自动执行不同的后续代码，因此学习布尔运算也是非常有必要的。

在Python中，布尔类型还可以与其他数据类型（字符串，数字等）做 and、or和not运算，请看下面的代码：

a = True

print(a and 0 or 99) # ==> 99

得到的计算结果不是布尔类型，而是数字99，这是为什么呢？

因为Python把0、空字符串和None看成False，其他数值和非空字符串都看成True，所以：

True and 0计算结果是0   
继续计算0 or 99计算结果是 99   
因此，结果是99。  
需要注意的是，not计算的优先级是高于and和or的。

True and not False # ==> True

在上述布尔计算中，先计算not False = True，然后再计算True and True，因此得到True的结果。

**短路计算**

1. 在计算a and b时，如果 a 是 False，则根据与运算法则，整个结果必定为 False，因此返回 a；如果 a 是 True，则整个计算结果必定取决与 b，因此返回 b。
2. 在计算a or b时，如果 a 是 True，则根据或运算法则，整个计算结果必定为 True，因此返回 a；如果 a 是 False，则整个计算结果必定取决于 b，因此返回 b。

所以Python解释器在做布尔运算时，只要能提前确定计算结果，它就不会往后算了，直接返回结果。

3-5 Python 字符串

前面我们讲解了什么是字符串。字符串可以用' '或者" "括起来表示。

如果字符串本身包含'怎么办？比如我们要表示字符串 I'm OK ，这时，可以用" "括起来表示：

"I'm OK"

类似的，如果字符串包含"，我们就可以用' '括起来表示：

'Learn "Python" in imooc'

但是，如果字符串既包含'又包含"怎么办？

这个时候，就需要对字符串中的某些特殊字符进行“转义”，Python字符串用\进行转义。

要表示字符串Bob said "I'm OK"  
由于'和"会引起歧义，因此，我们在它前面插入一个\表示这是一个普通字符，不代表字符串的起始，因此，这个字符串又可以表示为

'Bob said \"I\'m OK\".'

注意：转义字符 \不计入字符串的内容中。

常用的转义字符还有：

\n表示换行  
\t 表示一个制表符  
\\表示 \ 字符本身

3-6 raw字符串和多行字符串！

## Python中raw字符串与多行字符串

如果一个字符串包含很多需要转义的字符，对每一个字符都进行转义会很麻烦。为了避免这种情况，我们可以在字符串前面加个前缀r，表示这是一个 raw 字符串，里面的字符就不需要转义了。例如：

r'\(~\_~)/ \(~\_~)/'

但是r'...'表示法不能表示多行字符串，也不能表示包含'和 "的字符串。

如果要表示多行字符串，可以用'''...'''表示：

'''Line 1

Line 2

Line 3'''

上面这个字符串的表示方法和下面的是完全一样的：

'Line 1\nLine 2\nLine 3'

还可以在多行字符串前面添加r，把这个多行字符串也变成一个raw字符串：

r'''Python is created by "Guido".

It is free and easy to learn.

Let's start learn Python in imooc!'''

3-7 format { }里面是可变内容，

## Python的字符串format

字符串是Python程序重要的数据类型，到目前为止，我们输出的字符串的内容都是固定的，但有时候通过字符串输出的内容不是固定的，这个时候需要使用format来处理字符串，输出不固定的内容。  
字符串format由两个部分组成，字符串模板和模板数据内容组成，通过大括号{}，就可以把模板数据内容嵌到字符串模板对应的位置。

# 字符串模板

template = 'Hello {}'

# 模板数据内容

world = 'World'

result = template.format(world) //这里的template 只是一个代名词，相当于

print(result) # ==> Hello World

引用的时候 固定内容{}.format（可变内容的变量名字或者’可变内容’）

如果模板中{}比较多，则容易错乱，那么在format的时候也可以指定模板数据内容的顺序。

# 指定顺序

template = 'Hello {0}, Hello {1}, Hello {2}, Hello {3}.'

result = template.format('World', 'China', 'Beijing', 'imooc')

print(result) # ==> Hello World, Hello China, Hello Beijing, Hello imooc.

# 调整顺序

template = 'Hello {3}, Hello {2}, Hello {1}, Hello {0}.'

result = template.format('World', 'China', 'Beijing', 'imooc')

print(result) # ==> Hello imooc, Hello Beijing, Hello China, Hello World.

除了使用顺序，还可以指定对应的名字，使得在format过程更加清晰。

# 指定{}的名字w,c,b,i

template = 'Hello {w}, Hello {c}, Hello {b}, Hello {i}.'

world = 'World'

china = 'China'

beijing = 'Beijing'

imooc = 'imooc'

# 指定名字对应的模板数据内容

result = template.format(w = world, c = china, b = beijing, i = imooc)

print(result) # ==> Hello World, Hello China, Hello Beijing, Hello imooc.

### 任务

请使用两种format的方式打印字符串Life is short, you need Python。

**?**不会了怎么办

参考答案：

print('Life is short, you need {}'.format('Python'))

print('Life is short, you need {launguage}'.format( launguage = 'Python'))

3-9

字符串由一个个字符组成，每一个字符都有一个唯一的位置。比如字符串'ABC'，第一个字符是A，第二个字符是B，第三个字符是C。  
因此我们可以使用位置的方式取出字符串中特定位置的字符，按照位置取字符串的方式使用中括号[]访问，这个时候可以把字符串看作是一个列表（一种新的数据类型，在后面会继续学习），**不过需要注意的是，在程序的世界中，计数是从**0**开始的，使用**0**来表示第一个。**

s = 'ABC'

a = s[0] # 第一个

b = s[1] # 第二个

c = s[2] # 第三个

print(a) # ==> A

print(b) # ==> B

print(c) # ==> C

有时候，我们会想获取字符串的一部分（子串），这个时候我们采取切片的方式获取，切片需要在中括号[]中填入两个数字，中间用冒号分开，表示子串的开始位置和结束位置，并且这是半闭半开区间，不包括最后的位置。

ab = s[0:2] # 取字符串s中的第一个字符到第三个字符，不包括第三个字符

print(ab) # ==> AB

我们定义一个更长的字符串，了解切片更多的细节。

s = 'ABCDEFGHIJK'

abcd = s[0:4] # 取字符串s中的第一个字符到第五个字符，不包括第五个字符

print(abcd) # ==> ABCD

cdef = s[2:6] # 取字符串s中的第三个字符到第七个字符，不包括第七个字符

print(cdef) # ==> CDEF

注意是半开半闭区间!

和C语言里面的数组类似，效果一致。

4-1

**Python之if语句**

计算机之所以能做很多自动化的任务，因为它可以自己做条件判断，通过条件判断，选择做什么样的逻辑（当然，逻辑是需要我们提前写好的），我们称之为条件分支判断。  
举个例子，在100分试卷的考试中，小于60分我们认为是不及格的，因此，我们可以使用程序自动判断考试是否及格。

score = 59

if score < 60:

    print('抱歉，考试不及格')

# ==> 抱歉，考试不及格

这里，因为score = 59 < 60，所以if的判断是True，因此就会执行print('抱歉，考试不及格')。  
这里有几个需要注意的地方：

1. 可以看到print('抱歉，考试不及格')这行代码明显比上一行代码缩进了，这是因为这行代码是if判断的一个子分支，因此需要缩进，在Python规范中，一般使用4个空格作为缩进
2. 在if语句的最后，有一个冒号:，这是条件分支判断的格式，在最后加入冒号:，表示接下来是分支代码块

## Python之if-else语句

使用if判断，可以在当if条件为True时，执行if的子逻辑分支，但有时候，也想在if条件不为True时，执行别的子逻辑分支。  
比如：在上一节课的例子中，我希望当分数小于60分时输出，抱歉，考试不及格，否则，输出恭喜你，考试及格，这个时候可以使用if-else语句。

score = 59

if score < 60:

    print('抱歉，考试不及格')

else:

    print('恭喜你，考试及格')

这个时候，程序就会根据score的值，走不同的子逻辑分支，同学可以改变score的值试一试。  
在这里，同样需要注意两个地方，第一个是冒号:，在else中，同样需要冒号；其次是缩进，在else的子逻辑分支，同样需要缩进。

elif 这是 else if 在python里面的使用语法

**For循环**

班里考试后，老师要统计几位同学的平均成绩，已知5位同学的成绩用list表示如下：  
L = [75, 92, 59, 68, 99]  
请利用for循环计算出平均成绩。

**?**不会了怎么办

利用一个sum变量累加就可以计算出总成绩。

L = [75, 92, 59, 68, 99]

sum = 0.0

for x in L:

    sum = sum + x

print(sum / 5)

s = 'ABCD'

for ch in s:

    print(ch) # 注意缩进

在上述代码中，ch是在for循环中定义的，意思是把字符串s中的每一个元素依次赋值给ch，然后再把ch打印出来，直到打印出字符串s的最后一个字符为止。

## Python之break跳出循环

用 for 循环或者 while 循环时，如果要在循环体内直接退出循环，可以使用 break 语句。  
比如在前面的无限循环里面，只要在恰当的时机，我们使用break跳出循环，也可以求出1~100的和。

num = 1

sum = 0

while True:

    if num > 100:

        break

    sum = sum + num

    num = num + 1

print(sum)

同样的，对于字符串s = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'，假如希望输出s的前20个字符，而不是所有字符，我们也可以使用break。

s = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

num = 1

for ch in s:

    if num > 20:

        break

    print(ch)

    num = num + 1

## Python之continue继续循环

使用continue，我们可以控制循环继续下去，并跳过continue后面的逻辑，比如，对于字符串s = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'，假如希望输出字符串s中第10个以后的字符，而不是所有字符，这个时候， 我们可以使用continue跳过前面的9个字符。

s = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

num = 1

for ch in s:

    if num < 10:

        num = num + 1

        continue # 当num < 10时，跳过后续循环代码，继续下一次循环

    print(ch)

    num = num + 1

## 什么是容器、什么是list

### ****容器****

生活中，容器指的是可以容纳物品的收纳器，在程序中，容器是一种可以把多个元素放在一起的数据结构，容器中的元素可以逐个地迭代获取，可以用in, not in等关键字判断某个元素是否包含在容器中。  
在Python中，包括列表(list)、元组(tuple)、字典(dict)、集合(set)等，他们都可以放入多个元素，因此都可以算作是容器，这些容器是Python编程中非常重要的数据结构，我们接下来重点学习这些数据结构。

### ****list****

列表(list)是一种有序的容器，放入list中的元素，将会按照一定顺序排列。构造list的方法非常简单，使用中括号[]把需要放在容器里面的元素括起来，就定义了一个列表。  
比如列出所有同学们的成绩：

scores = [45, 60, 75, 86, 49, 100]

列出所有同学们的名字：

names = ['Alice', 'Bob', 'David', 'Ellena'] # 注意，字符串元素仍需要引号

正如我们看到的，list可以放入数字、字符串等数据类型，list不对放入其中的类型进行判断，也就是说，list可以同时放入任意类型的数据，这是Python这门语言决定的，因为Python是动态语言。

L = ['Alice', 66, 'Bob', True, 'False', 100]

对于list，我们可以直接把list的内容打印出来。

L = ['Alice', 66, 'Bob', True, 'False', 100]

print(L)

## Python按顺序访问list

列表是有序的，因此我们可以按顺序访问列表中的元素。

L = ['Alice', 66, 'Bob', True, 'False', 100]

for item in L:

    print(item)

回想一下，在前面，我们学习过字符串，这里使用for循环访问列表中的每一个元素和使用for循环访问字符串中的每一个字符是非常类似的。  
事实上字符串也可以看作是一种特殊的列表，它只能按顺序存放多个字符。通过for循环访问列表每一个元素的方式，我们称为迭代。  
而对于一个空的列表，使用for循环访问，将不会打印任何东西。

L = []

for item in L:

    print(item)

## Python按索引访问list

由于列表list是一个有序的容器，所以在list里面，每一个元素都有一个唯一的位置，我们称之为索引(index)，这和字符串是类似的，因此我们也可以通过位置的方式获取list中的元素，回顾一下前面学习的，我们通过中括号[]来按位置访问对应的元素。  
**注意，这里要注意区分，列表的定义也是通过中括号**[]**来定义的，但这和通过索引访问列表里面的元素并不冲突**

names = ['Alice', 'Bob', 'David', 'Ellena']

print(names[0])

print(names[1])

print(names[2])

print(names[3])

由于names只有四个元素，所以我们最多只能通过索引3访问到最后一个元素。试想一下，如果我们print(names[4])，会怎么样？

names = ['Alice', 'Bob', 'David', 'Ellena']

print(names[4])

事实上，这会引起Python运行的错误，提示索引访问超出范围。

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

IndexError: list index out of range

因此，我们在使用索引的方式访问列表时，一定要特别注意不要越界。  
同时，列表和字符串一样，也支持切片，通过切片的方式，获取到列表的子列表。

names = ['Alice', 'Bob', 'David', 'Ellena']

sub\_names = names[0:2]

print(sub\_names)

这里需要注意一下，如果我们越界切片的话，不会出现Python运行错误，但是按照这样的下标去切片，获取不到任何元素。

names = ['Alice', 'Bob', 'David', 'Ellena']

sub\_names = names[5:10]

print(sub\_names) # ==> []

## Python倒序访问list

Python的列表，除了支持正向顺序索引获取列表中的每一个元素以外，也支持倒序访问list中的每一个元素。

names = ['Alice', 'Bob', 'David', 'Ellena']

对于names列表，Ellena的名字排在最后，也就是我们所说的倒数第一个，在Python中，可以使用-1来表示最后一个元素。

names = ['Alice', 'Bob', 'David', 'Ellena']

print(names[-1]) # ==> Ellena

同样的道理，我们可以通过-2，打印出David的名字，通过-3，打印出Bob的位置。

print(names[-2]) # ==> David

print(names[-3]) # ==> Bob

注意，如果我们使用-5的话，因为不存在倒数第五个名字，因此这也是一种越界，同样会报错。

names = ['Alice', 'Bob', 'David', 'Ellena']

print(names[-5])

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

IndexError: list index out of range

## Python向list添加新的元素

现在班里面有4名同学：

names = ['Alice', 'Bob', 'David', 'Ellena']

今天，班里转来一名新同学Candy，如何把新同学添加到现有的列表当中呢？  
在Python中，list提供了一系列的方法可以让我们操作list中的元素，其中也包含了添加元素的方法。  
第一个办法是用append()方法，把新同学追加到列表的末尾：

names = ['Alice', 'Bob', 'David', 'Ellena']

names.append('Candy')

print(names) # ==> ['Alice', 'Bob', 'David', 'Ellena', 'Candy']

**注意，**append()**方法总是将元素添加到list的尾部。**   
如果上面的列表需要按照首字母排序的话，那么Candy应该是排在第三的位置的，这怎么办呢？  
这就需要使用list的insert()方法，insert()方法和append()方法不一样，insert()方法需要两个参数，分别是需要插入的位置，以及需要插入的元素。

names = ['Alice', 'Bob', 'David', 'Ellena']

names.insert(2, 'Candy')

print(names) # ==> ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

**注意，将Candy插入到第三的位置之后，原来的名字，都将自动往后移动一位，这个时候再使用相同的索引获取后面的元素，将会得到不一样的结果**

names = ['Alice', 'Bob', 'David', 'Ellena']

print(names[2]) # ==> David

names.insert(2, 'Candy')

print(names[2]) # ==>Candy

## Python从list删除元素

如果Ellena因为家庭原因需要转学，那么我们如何把Ellena从已有的列表里面删除呢？  
这个时候我们可以使用列表的pop()方法，pop()方法默认删除列表的最后一个元素，并返回。

L = ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

name = L.pop()

print(name) # ==> Ellena

print(L) # ==> L = ['Alice', 'Bob', 'Candy', 'David']

对于Ellena，由于Ellena恰好位于列表的最后，所以可以直接使用pop()方法把Ellena从列表的最后删除，假如需要转学的不是Ellena，而是Candy，这个时候应该怎么办呢？  
pop()方法，除了可以删除最后一个元素以外，pop()还可以接收一个参数，指定需要删除的元素的位置。

L = ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

name = L.pop(2)

print(name) # ==> Candy

print(L) # ==> ['Alice', 'Bob', 'David', 'Ellena']

## Python替换list中的元素

对于列表，除了可以向列表添加元素，删除列表元素以外，列表已有的元素，也是可以修改的，通过索引指定位置，并赋值新的元素，即可替换列表中原有的元素。  
假如班上同学Candy需要转走了，同时有一个新的同学Canlina转入，那么按照字母排序，Canlina的位置恰好是Candy的位置。

L = ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

L[2] = 'Canlina'

print(L)

我们也可以使用倒序索引来完成同样的功能。

L = ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

L[-3] = 'Canlina'

print(L)

注意，如果替换一个不存在的下标，则同样会引起Python运行错误。

L = ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

L[6] = 'Canlina'

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

IndexError: list assignment index out of range

## Python二维list

有时候，一维list并不能满足所有的要求（上述所有list均为一维list），这个时候需要二维list甚至更高维的list。  
比如：  
Alice最近的三次成绩分别是[100, 89, 92]  
Bob最近的三次成绩分别是[70, 65, 81]  
Candy最近的三次成绩分别是[88, 72, 77]  
如果需要用一个列表存放三个同学的成绩，则需要这样：

alice\_scores = [100, 89, 92]

bob\_scores = [70, 65, 81]

candy\_scores = [88, 72, 77]

all\_scores = [alice\_scores, bob\_scores, candy\_scores]

print(all\_scores) # ==> [[100, 89, 92], [70, 65, 81], [88, 72, 77]]

这个时候得到的就是一个二维list，对于二维list，列表里面的每一个元素仍然是一个列表。这个时候，如果需要从二维list all\_scores获取Bob最近第三次考试的成绩，可以这样写

alice\_scores = [100,89,92]

bob\_scores = [70,65,81]

candy\_scores = [88,72,77]

all\_scores = [alice\_scores, bob\_scores, candy\_scores]

score = all\_scores[1][2] # ==> 81

其中all\_scores[1]得到Bob的最近三次成绩的列表，再通过下标[2]，则可以得到Bob第三次的成绩。

### 任务

有三个长方体，他们的长宽高分别是[1, 2, 3], [5, 3, 2], [7, 3, 2]，定义在数组L中，L = [[1, 2, 3], [5, 3, 2], [7, 3, 2]]，请分别求出三个长方体的表面积。

L = [[1, 2, 3], [5, 3, 2], [7, 3, 2]]

for x in L:

a=x[0]

b=x[1]

c=x[2]

result=a\*b\*2+a\*c\*2+b\*c\*2

print(result)

## 什么是tuple

元组(tuple)和list一样，也是一个有序容器，在元组中，同样可以包含0个或者多个元素，并且也支持索引访问、切片等操作。  
定义元组的方式是使用小括号()将元组内的元素括起来。

T = ('Alice', 'Bob', 'Candy', 'David', 'Ellena')

# 通过下标的方式访问元素

print(T[0]) # ==> Alice

print(T[4]) # ==> Ellena

# 切片

print(T[1:3]) # ==> ('Bob', 'Candy')

元组数据类型可以把不是元组的容器转换为元组，比如将列表转换成元组。

L = ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

print(L) # ==> ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

T = tuple(L)

print(T) # ==> ('Alice', 'Bob', 'Candy', 'David', 'Ellena')

同样的，对于列表数据类型，也可以把元组转换成列表。

T = ('Alice', 'Bob', 'Candy', 'David', 'Ellena')

print(T) # ==> ('Alice', 'Bob', 'Candy', 'David', 'Ellena')

L = list(T)

print(L) # ==> ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

但是，tuple和list不一样的是，tuple是固定不变的，一旦变成tuple，tuple中的每一个元素都不可被改变，同时也不能再往tuple中添加数据，而list是可以的。

T = ('Alice', 'Bob', 'Candy', 'David', 'Ellena')

# 替换元素

T[1] = 'Boby'

# 报错

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

请注意，元组(tuple)的这个特性是非常重要的，在运行上tuple的性能是list的数倍。

## 访问tuple元素的其他方法

由于tuple一旦定义之后便不可修改，所以在实际编程中，tuple经常用于存放固定不变的数据。  
因此在使用上，tuple提供了便捷的方法可以访问tuple中的数据。

### ****count()方法****

count()方法用来统计tuple中某个元素出现的次数。

T = (1, 1, 2, 2, 3, 3, 1, 3, 5, 7, 9)

print(T.count(1)) # ==> 3

print(T.count(5)) # ==> 1

对于不存在的元素，count方法不会报错，而是返回0，这是合理的，因为元组里面有0个不存在的元素。

T = (1, 1, 2, 2, 3, 3, 1, 3, 5, 7, 9)

print(T.count(10)) # ==> 0

### ****index()方法****

index()方法可以返回指定元素的下标，当一个元素多次重复出现时，则返回第一次出现的下标位置。

T = (1, 1, 2, 2, 3, 3, 1, 3, 5, 7, 9)

T.index(9) # ==> 10

T.index(5) # ==> 8

T.index(1) # ==> 0 # 多次出现，返回第一次出现的位置

注意，index()方法和count()方法不一样，当指定的元素不存在时，使用index()方法Python会报错。

T = (1, 1, 2, 2, 3, 3, 1, 3, 5, 7, 9)

T.index(100)

# 报错

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

ValueError: tuple.index(x): x not in tuple

## Python创建单个元素的tuple

tuple和list一样，可以包含 0 个、1个和任意多个元素。  
包含多个元素的 tuple，前面我们已经创建过了。

包含 0 个元素的 tuple，也就是空tuple，直接用()表示：

T = ()

print(T) # ==> ()

接着，我们创建包含一个元素的tuple。

T = (1)

print(T) # ==> 1

这和我们期望的输出有些差异，为什么包含一个元素的元组打印出来之后没有小括号，而是只有一个数字1呢？  
回顾一下前面数字的四则运算。

result = 3 \* 4 - 2

print(result) # ==> 10

# 改变优先级，先运算减法

result = 3 \* (4 - 2)

print(result) # ==> 6

可以看到，改变优先级我们是通过()来实现的，这和元组的定义有冲突，这就解释了前面只有一个元素的元组，为什么打印出来却得到一个数字的结果了。  
因为()既可以表示tuple，又可以作为括号表示运算时的优先级，结果(1)被Python解释器计算出结果 1，导致我们得到的不是tuple，而是整数 1。  
因此，要定义只有一个元素的tuple，需要在元素后面添加一个逗号,。

T = (1, )

print(T) # ==> (1, )

而对于多个元素的tuple，则加和不加这个逗号，效果是一样的。

>>> T = (1, 2, 3,)

>>> print(T) # ==> (1, 2, 3)

tuple的元素也可以是tuple，请判断以下定义T的元素中，有多少个tuple。

T = ((1+2), ((1+2),), ('a'+'b'), (1, ), (1,2,3,4,5))

**?**不会了怎么办

参考答案：

3个  
第一个元素(1+2)，第三个元素('a'+'b')，计算的结果分别是数字3和字符串ab，不是tuple。

## Python的可变tuple

前面我们学习了，对于tuple，它和list一个最大的不同点就是tuple是不可变的，tuple里面的元素，也是不可替换的。但是这针对的是仅包含基础数据类型（数字类型、布尔类型、字符串类型）的数据，对于组合数据类型，则不受这个约束。

T = (1, 'CH', [3, 4])

这里T有三个元素，第一个元素是数字类型，第二个元素是字符串类型，第三个元素是列表类型的，我们尝试修改第三个元素的数据。

T = (1, 'CH', [3, 4])

L = T[2]

print(L) # ==> [3, 4]

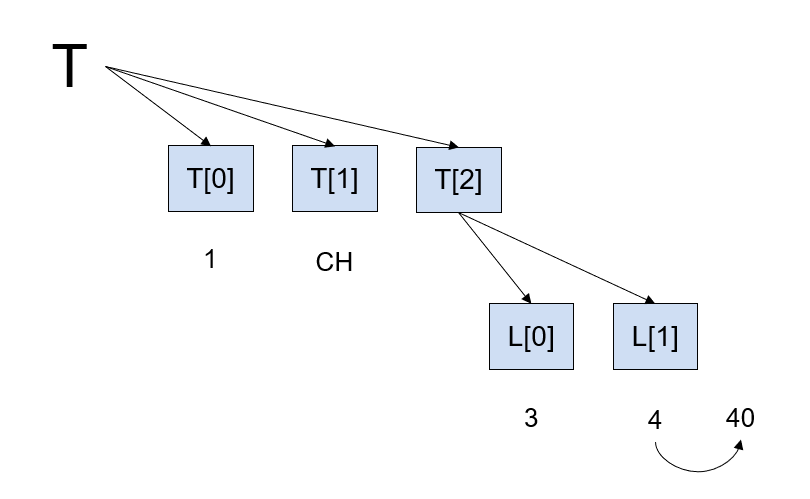
# 尝试替换L中的元素

L[1] = 40

print(L) # ==> [3, 40]

print(T) # ==> (1, 'CH', [3, 40])

这个时候，我们发现，元组T中的第三个元素已经成功被改变了，这就有悖前面的定义，元组是不可改变的。那么这到底是为什么呢？  
这是因为虽然tuple中的list元素改变了，但是tuple本身指向的list仍然是同一个list，list本身并没有改变，改变的只是list里面的一个元素，这是tuple所约束不到的范围。



T = (1, 'CH', [3, 4])

L2 = [3, 40]

# 尝试替换tuple中的list

T[2] = L2

# 报错

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

如果我们直接替换list，这也是不行的。

## 什么是dict

我们已经知道，List 和 tuple 可以用来表示顺序集合，例如，班里同学的名字：

['Alice', 'Bob', 'Candy', 'David', 'Ellena'] # List

('Alice', 'Bob', 'Candy', 'David', 'Ellena') # tuple

或者考试的成绩：

[45, 60, 75, 86, 49] # list

(45, 60, 75, 86, 49) # tuple

如果同学名字的列表和同学成绩的列表是一一对应的，那么通过下标，我们也可以找到每个同学的成绩。

names = ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

scores = [45, 60, 75, 86, 49]

index = 0

for name in names:

score = scores[index]

print('name = {}, score = {}'.format(name, score))

index = index + 1

事实上，我们可以得到这样的映射。

'Alice' ==> 45

'Bob' ==> 60

'Candy' ==> 75

'David' ==> 86

'Ellena' ==> 49

但是使用两个list，始终有些麻烦的，尤其是需要变换一个列表的顺序后，另外一个列表也需要做同样的变换，否则就可能出现对应不上的问题。  
python的dict就是专门保存这种映射的，使用dict可以方便的保存“名字”->“成绩”的映射。  
在dict中，每一项包含一个key和一个value，key和value是一一对应的，在解决上面的问题中，我们可以使用名字作为key，成绩作为value，那么dict的定义如下：

d = {

'Alice': 45,

'Bob': 60,

'Candy': 75,

'David': 86,

'Ellena': 49

}

在定义里，我们使用花括号{}表示这是一个dict，然后key和value之间使用冒号:分割，并且每一组key:value的最后，以逗号,表示这一组的结束。  
我们也可以使用以下的方式定义一个dict。

d = dict()

print(d) # ==> {}

不过这种定义方式，默认得到的是一个空dict，需要调用函数往里面添加数据，我们后面会继续学习。

## Python读取dict元素

我们现在可以创建一个dict，保存名字和成绩的对应关系。

d = {

'Alice': 45,

'Bob': 60,

'Candy': 75,

'David': 86,

'Ellena': 49,

'Gaven': 86

}

此时，如果想通过名字来查询某个同学的成绩，也就是通过key来查询value，这个时候怎么办呢？  
dict提供通过key找到对应value的功能，通过d[key]的形式，就可以得到对应的value。

d = {

'Alice': 45,

'Bob': 60,

'Candy': 75,

'David': 86,

'Ellena': 49,

'Gaven': 86

}

print(d['Bob']) # ==> 60

print(d['Alice']) # ==> 45

这和list通过下标找到对应位置的元素是类似的。  
回顾一下前面使用下标的方式访问list元素的时候，当下标不存在时，就会引发错误，在dict中，也是一样的，当对应的key不存在时，也会引发错误。

d = {

'Alice': 45,

'Bob': 60,

'Candy': 75,

'David': 86,

'Ellena': 49,

'Gaven': 86

}

print(d['Dodo'])

# 抛出异常

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'Dodo'

它的意思是key不存在，因此我们在需要通过key找到value时，一定要先判断key存不存在，然后才使用上面的方式获取对应的value，以避免错误。

if 'Alice' in d:

print(d['Alice']) # ==> 45

if 'Dodo' in d: # Dodo不存在，所以不会走下面的逻辑

print(d['Dodo'])

除了使用这种方法，还有一种方法可以通过key来获取对应的value，这种方法不会引起错误，dict本身提供get方法，把key当作参数传递给get方法，就可以获取对应的value，当key不存在时，也不会报错，而是返回None。

print(d.get('Alice')) # ==> 45

print(d.get('Dodo')) # ==> None

因为通过get方法在代码实现上更加简单，且不会引起错误，因此更加推荐使用get方法来获取dict的元素。

## Python添加dict元素

dict和tuple不一样，dict是可变的，我们随时可以往dict中添加新的key-value，比如对于上节课的成绩dict：

d = {

'Alice': 45,

'Bob': 60,

'Candy': 75,

'David': 86,

'Ellena': 49

}

需要往里面添加Dodo、Mimi的成绩时，可以使用赋值语句往里面添加元素：

d['Mimi'] = 72

d['Dodo'] = 88

print(d)

实际上，value可以是任意类型的元素，可以是list、tuple等，假如Mimi近两次成绩分别是72，73，Dodo近两次的成绩分别是88，90，则可以使用赋值语句往dict中添加list元素。

d['Mimi'] = [72, 73]

d['Dodo'] = [88, 90]

print(d)

此后，如果Mimi、Dodo的第三次成绩也出来了，分别是75，90，则可以先通过key把对应的value查询出来，然后再往类型是list的value中添加第三次的成绩。

d['Mimi'].append(75)

d['Dodo'].append(90)

print(d)

**Python更新dict元素**

上一节课我们学习了往dict中添加元素的方法，通过赋值语句就可以把元素添加到dict中去，但是赋值的时候，我们并不确定key是否已经在dict里面了，如果dict里面已经有对应的key了，那将会发生什么呢？

d = {

'Alice': 45,

'Bob': 60,

'Candy': 75,

'David': 86,

'Ellena': 49

}

d['Bob'] = 75

print(d)

# ==> {'Alice': 45, 'Bob': 75, 'Candy': 75, 'David': 86, 'Ellena': 49}

这个时候我们发现，原来Bob的成绩是60，现在变成75了，因为d['Bob'] = 75的缘故。  
因此我们发现这个赋值语句其实有两个功能：

1. 当key不存在时，往dict中添加对应的key: value元素。
2. 当key存在时，会更新dict，用新的value替换原来的value。

因此，在使用赋值语句往dict中添加元素时，为了避免不必要的覆盖问题，我们需要先判断key是否存在，然后再做更新。

## Python删除dict元素

当同学转校时，我们需要把同学的成绩从已有的成绩dict中删除，这个时候我们就需要学习如何删除dict中的元素。  
dict提供便捷的pop()方法，允许我们快速删除元素，pop()方法需要指定需要删除的元素的key，并返回对应的value。  
假设Alice转校了，需要把Alice的成绩删除，可以这样写：

d = {

'Alice': 45,

'Bob': 60,

'Candy': 75,

'David': 86,

'Ellena': 49

}

print(d) # ==> {'Alice': 45, 'Bob': 60, 'Candy': 75, 'David': 86, 'Ellena': 49}

alice\_score= d.pop('Alice')

print(alice\_score) # ==> 45

print(d) # ==> {'Bob': 60, 'Candy': 75, 'David': 86, 'Ellena': 49}

需要注意的是，pop()方法的参数是dict中的key，当key不存在时，同样会引起错误。比如在上述操作中，已经把Alice的成绩删除了，假如再次pop('Alice')，将会引发错误。

d.pop('Alice')

# 报错

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'Alice'

### 任务

在dict中，使用keys()方法，可以返回dict的所有key，在删除某个元素时，可以通过这个方法先判断某个元素是否存在，请改造前面的程序，使得即使key不存在时，删除也不会抛异常。

**?**不会了怎么办

参考答案：

d = {

'Alice': 45,

'Bob': 60,

'Candy': 75,

'David': 86,

'Ellena': 49

}

name = 'Alice'

if name in d.keys():

d.pop(name)

else:

print('{} not in d'.format(name))

## Python dict的特点

### ****查找速度快****

dict的第一个特点是查找速度快，无论dict有10个元素还是10万个元素，查找速度都一样。而list的查找速度随着元素增加而逐渐下降。  
不过dict的查找速度快不是没有代价的，dict的缺点是占用内存大，还会浪费很多内容，list正好相反，占用内存小，但是查找速度慢。

### ****有序与无序****

在Python3.5之前，dict中的元素是无序的，也就是dict中元素的插入顺序和打印顺序未必一致，比如使用Python3.5之前的版本执行以下代码：

d = {

'Alice': 45,

'Bob': 60,

'Candy': 75,

'David': 86,

'Ellena': 49

}

print(d) # ==> {'Bob': 60, 'Ellena': 49, 'Alice': 45, 'Candy': 75, 'David': 86}

可以看到，打印的顺序和定义的顺序并不一致。  
但是在Python3.6、Python3.7版本中，却得到了有序的结果。

print(d) # ==> {'Alice': 45, 'Bob': 60, 'Candy': 75, 'David': 86, 'Ellena': 49}

为什么在不同的版本中，会得到不一样的结果呢？这是因为底层的实现发生了改变，我们可以认为在Python3.6的版本以后，dict是有序的，但是一般而言，为了避免不必要的误解，一般在需要有序的dict时，我们会使用一种叫做Ordereddict的字典，来确保有序。

### ****key不可变****

对于基础数据类型，字符串、数字等，这些都是不可变的，可以作为dict的key，而对于复杂数据类型，经过前面的学习，我们知道tuple是不可变的，list是可变的，因此tuple可以作为dict的key，但是list不可以作为dict的key，否则将会报错。

key = (1, 2, 3) # 以tuple作为key

d[key] = True

key = [1, 2, 3]

d[key] = True

# 报错

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unhashable type: 'list'

如上所示，如果将list作为dict的key，将会引起错误。  
由于dict是按 key 查找，所以，在一个dict中，key不能重复。

## ****Python遍历dict****

通过直接print(d)，我们打印出来的是完整的一个dict；有时候，我们需要把dict中m一定条件的元素打印出来，比如成绩超过60的，在这种情况下，我们需要则需要遍历dict（这种时候需要使用for循环），并通过条件判断把满足条件的打印出来。  
遍历dict有两种方法， 第一种是遍历dict的所有key，并通过key获得对应的value。

d = {

'Alice': 45,

'Bob': 60,

'Candy': 75,

'David': 86,

'Ellena': 49

}

for key in d: # 遍历d的key

value = d[key]

if value > 60:

print(key, value)

# ==> Candy 75

# ==> David 86

第二种方法是通过dict提供的items()方法，items()方法会返回dict中所有的元素，每个元素包含key和value。

for key, value in d.items():

if value > 60:

print(key, value)

# ==> Candy 75

# ==> David 86

## Python操作dict的其他方法

除了前面学习的增删改查，dict还提供了非常多的工具函数帮助我们方便快捷的使用dict。

### ****获取dict的所有key****

dict提供keys()函数，可以返回dict中所有的key。

d = {'Alice': [50, 61, 66], 'Bob': [80, 61, 66], 'Candy': [88, 75, 90]}

for key in d.keys():

print(key)

# ==> Alice

# ==> Bob

# ==> Candy

### ****获取dict所有的value****

dict提供values()函数，可以返回dict中所有的value。

d = {'Alice': [50, 61, 66], 'Bob': [80, 61, 66], 'Candy': [88, 75, 90]}

for key in d.values():

print(key)

# ==> [50, 61, 66]

# ==> [80, 61, 66]

# ==> [88, 75, 90]

### ****清除所有元素****

dict提供clear()函数，可以直接清除dict中所有的元素。

d = {'Alice': [50, 61, 66], 'Bob': [80, 61, 66], 'Candy': [88, 75, 90]}

print(d) # ==> {'Alice': [50, 61, 66], 'Bob': [80, 61, 66], 'Candy': [88, 75, 90]}

d.clear()

print(d) # ==> {}

## 什么是set

在前面，我们学习了dict，知道dict的key是不重复的，当我们往dict里添加一个相同key的value时，新的value将会覆盖旧的value。  
有的时候，我们只想要 dict 的 key，不关心 key 对应的 value，目的就是保证这个集合的元素不会重复，这时，set就派上用场了。  
set和list类似，拥有一系列元素，但是set和list不一样，set里面的元素是不允许重复的，而list里面可以包含相同的元素；set与list的另一个区别是，set里面的元素是没有顺序的。  
创建set的方式是使用set()，并传入一个list，list的元素将会被转换成set的元素。

s = set([1, 4, 3, 2, 5])

print(s) # ==> set([1, 2, 3, 4, 5])

需要注意的是，上述打印的形式类似 list， 但它不是 list，仔细看还可以发现，打印的顺序和原始 list 的顺序有可能是不同的，因为set内部存储的元素是无序的。  
另外，set不能包含重复的元素，我们传入重复的元素看看会发生什么。

s = set([1, 4, 3, 2, 5, 4, 2, 3, 1])

print(s) # ==> set([1, 2, 3, 4, 5])

可以看到，在传入set()的list中，包含了重复的元素，但是打印的时候，相同的元素只保留了一个，重复的元素都被去掉了，这是set的一个重要特点。

## Python添加set元素

我们通过set()传入list的方法创建了set，如果set在使用过程中需要往里面添加元素，这个时候应该怎么添加呢？  
set提供了add()方法，我们可以使用add()方法，往set里面添加元素。  
比如，班里面来了新的同学，名字叫Gina。

names = ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

name\_set = set(names)

name\_set.add('Gina')

print(name\_set) # ==> set(['Gina', 'Alice', 'Candy', 'David', 'Ellena', 'Bob'])

可以看到，'Gina'已经添加到name\_set里面去了。对于set，如果添加一个已经存在的元素，不会报错，也不会改变什么。

names = ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

name\_set = set(names)

name\_set.add('Alice')

print(name\_set) # ==> set(['Bob', 'Ellena', 'Alice', 'Candy', 'David'])

有些时候需要批量往set里面添加元素，如果一个一个add是比较麻烦的，有没有批量往set里面添加元素的方法呢？  
set提供了update()方法，可以一次性给set添加多个元素。  
比如，新来了一批同学，名字分别是['Hally',  'Isen',  'Jenny',  'Karl']，则可以使用update()方法，批量往set中添加。

names = ['Alice', 'Bob', 'Candy', 'David', 'Ellena']

new\_names = ['Hally', 'Isen', 'Jenny', 'Karl']

name\_set = set(names)

name\_set.update(new\_names) # ==> set(['Jenny', 'Ellena', 'Alice', 'Candy', 'David', 'Hally', 'Bob', 'Isen', 'Karl'])

print(name\_set)

## Python删除set元素

和list、dict一样，有时候我们也需要考虑删除set的元素。  
set提供了remove()方法允许我们删除set中的元素。

name\_set = set(['Jenny', 'Ellena', 'Alice', 'Candy', 'David', 'Hally', 'Bob', 'Isen', 'Karl'])

name\_set.remove('Jenny')

print(name\_set) # ==> set(['Ellena', 'Alice', 'Candy', 'David', 'Hally', 'Bob', 'Isen', 'Karl'])

需要注意的是，如果remove的元素不在set里面的话，那么将会引发错误。

name\_set = set(['Jenny', 'Ellena', 'Alice', 'Candy', 'David', 'Hally', 'Bob', 'Isen', 'Karl'])

name\_set.remove('Jenny')

print(name\_set) # ==> set(['Ellena', 'Alice', 'Candy', 'David', 'Hally', 'Bob', 'Isen', 'Karl'])

name\_set.remove('Jenny') # ==> 重复remove 'Jenny'

# 引起错误

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'Jenny'

因此，使用remove()方法，我们需要格外小心，需要提前判断要remove()的元素是否在set里面，确保存在后，再进行remove。

## Python 操作set的其他方法

### ****不会报错的删除方法discard()****

除了使用remove()方法删除元素以外，还可以使用discard()方法删除元素，并且，和remove()不同的是，当元素不存在时，使用discard()并不会引发错误，所以使用discard()是更加高效的一个方法。

name\_set = set(['Jenny', 'Ellena', 'Alice', 'Candy', 'David', 'Hally', 'Bob', 'Isen', 'Karl'])

name\_set.discard('Jenny')

print(name\_set) # ==> set(['Ellena', 'Alice', 'Candy', 'David', 'Hally', 'Bob', 'Isen', 'Karl'])

name\_set.discard('Jenny')

print(name\_set) # ==> set(['Ellena', 'Alice', 'Candy', 'David', 'Hally', 'Bob', 'Isen', 'Karl']

### ****清除所有元素的方法clear()****

和dict一样，set也提供了clear()方法，可以快速清除set中的所有元素。

name\_set = set(['Jenny', 'Ellena', 'Alice', 'Candy', 'David', 'Hally', 'Bob', 'Isen', 'Karl'])

print(name\_set) # ==> set(['Jenny', 'Ellena', 'Alice', 'Candy', 'David', 'Hally', 'Bob', 'Isen', 'Karl'])

name\_set.clear()

print(name\_set) # ==> set([])

### ****集合的子集和超集****

set提供方法判断两个set之间的关系，比如两个集合set，判断其中一个set是否为另外一个set的子集或者超集。

s1 = set([1, 2, 3, 4, 5])

s2 = set([1, 2, 3, 4, 5, 6, 7, 8, 9])

# 判断s1是否为s2的子集

s1.issubset(s2) # ==> True

# 判断s2是否为s1的超集

s2.issuperset(s1) # ==> True

### ****判断集合是否重合****

有时候需要判断两个集合是否有重合的地方，如果使用传统的方法，需要使用for循环一个一个的去判断，非常麻烦，set提供isdisjoint()方法，可以快速判断两个集合是否有重合，如果有重合，返回False，否则返回True。

s1 = set([1, 2, 3, 4, 5])

s2 = set([1, 2, 3, 4, 5, 6, 7, 8, 9])

s1.isdisjoint(s2) # ==> False，因为有重复元素1、2、3、4、5

## 什么是函数

对于什么是函数，其实在前面的学习过程当中，已经多次接触到函数了，比如在set里面，使用remove()函数进行元素的删除，使用add()函数添加元素，使用update()函数批量添加元素，但是至今为止，我们都没有对函数有个充分的认识。本章，我们将具体学习函数。  
我们知道圆的面积计算公式为：

S = πr²

当我们知道半径r的值时，就可以通过公式计算出面积，假设我们需要计算3个不同大小的圆的面积：

r1 = 12.34

r2 = 9.08

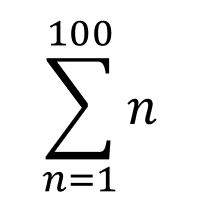
r3 = 73.1

s1 = 3.14 \* r1 \* r1

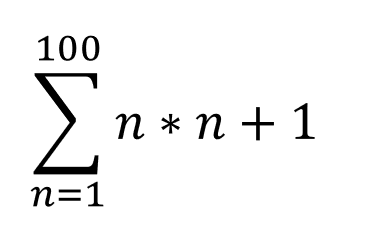
s2 = 3.14 \* r2 \* r2

s3 = 3.14 \* r3 \* r3

可以看到，在上述代码中，出现了几乎完全重复的代码，每次计算圆的面积的时候我们都是通过3.14\*x\*x（其中x是圆的半径）的方式计算出来的，这样写不仅非常麻烦，而且，当我们需要提高圆周率的精度时，把3.14改成3.14159265359的时候，我们需要修改涉及到的三行代码。  
因此，我们可以使用函数，把重复的逻辑代码封装起来，这样子，我们就不需要每次计算的时候，都写3.14\*x\*x这样的代码了；当我们使用函数的时候，我们只需要定义一次，就可以多次使用。  
我们把封装重复逻辑代码的过程就做抽象，抽象是数学中非常常见的概念。  
比如：计算数列的和，比如：1 + 2 + 3 + ... + 100，写起来十分不方便，于是数学家发明了求和符号∑，可以把1 + 2 + 3 + ... + 100记作：



这种抽象记法非常强大，因为我们看到∑就可以理解成求和，而不是还原成低级的加法运算。  
而且，这种抽象记法是可扩展的，比如：

  
还原成加法运算就变成了：

(1 x 1 + 1) + (2 x 2 + 1) + (3 x 3 + 1) + ... + (100 x 100 + 1)

可见，借助抽象，我们才能不关心底层的具体计算过程，而直接在更高的层次上思考问题。  
写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。  
Python不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

## ****Python调用函数****

Python内置了很多有用的函数，我们可以直接调用。比如前面求list的长度len()函数等等，都是Python内置的函数，我们经常会使用到它们。  
在这个文档里面，列举了Python内置的大部分函数，同学们有兴趣可以参考看看。

https://docs.python.org/3/library/functions.html

要调用一个函数，需要知道函数的名称和参数，比如求绝对值的函数 abs()，它接收一个参数。  
对于abs()函数，abs就是函数的名称，括号()内，就是函数的参数，当函数没有参数时，默认就是一个空括号。**abs接收一个参数，这个参数就是需要求绝对值的数，这个参数可以是整数，也可以是浮点数**

abs(-100) # ==> 100

abs(20) # ==> 20

abs(-3.14159) # ==> 3.14159

需要注意的是，传递的参数数量一定要和函数要求的一致，不然将会引起错误，比如，如果在abs()函数中传入两个参数。

abs(1, 2)

# 报错

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: abs() takes exactly one argument (2 given)

在错误提示中，清晰的说明了abs()函数只接收一个参数，但是却传递了两个参数，所以引起了错误。  
其次，如果传入的参数数量是对的，但是参数的类型不能被函数所接受，也会引起错误，比如：求绝对值的函数abs()，只有数字才拥有绝对值，如果传递一个字符串进去，将会引起错误。

abs('3.1415926')

# 报错

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: bad operand type for abs(): 'str'

这里错误提示说，str类型是错误的参数类型。  
除了abs()函数，还有很多常见的函数，比如cmp()函数，可以比较两个数的大小，这个时候，cmp()函数就接收两个参数。  
对于cmp(x, y)，如果x < y 返回 -1，如果x == y 函数返回0，如果x > y函数返回1。

cmp(1, 2) # ==> -1

cmp(2, 1) # ==> 1

cmp(3, 3) # ==> 0

还有基础数据类型的转换函数，int()函数可以将合法的其它类型数据转换为整数，str()函数可以将其它类型的数据转换为字符串。

int('123') # ==> 123

int(12.34) # ==> 12

str(123) # ==> '123'

str(1.23) # ==> '1.23'

在学习Python的过程中，我们将会接触到越来越多的Python内置函数，这些内置函数提供了非常有用的功能，大大降低我们编程的难度。

## Python定义函数

除了使用Python内置的函数以外，在编程过程中，我们也经常需要自己定义函数。  
在Python中，定义一个函数要使用 def 语句，依次写出函数名、括号()、括号中的参数和冒号:，然后，在缩进块中编写函数体，函数的返回值用 return 语句返回。  
我们以定义一个求绝对值的函数my\_abs函数为例：

def my\_abs(x):

    if x >= 0:

        return x

    else:

        return -x

请注意，return表示返回的意思，函数体内部的语句在执行时，一旦执行到return时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。  
我们继续定义一个求列表list所有元素的和的函数：

def list\_sum(L):

    result = 0

    for num in L:

        result = result + num

    return result

这样子我们就定义了一个sum\_list()的函数，注意，在最后return，我们把求和的结果result返回了，这样就可以在外部调用函数后获得result。

L = [1, 3, 5, 7, 9, 11]

result =list\_sum(L) # 调用定义的sum\_list函数并获得return返回的结果

print(result)

sum()函数接收一个list作为参数，并返回list所有元素之和。请计算 1\*1+ 2\*2 + 3\*3 + ... + 100\*100。

**?**不会了怎么办

参考答案：

L = []

x = 1

while x <= 100:

L.append(x \* x)

x = x + 1

print(sum(L))

## Python函数返回值

在上一节课中，我们在函数里面使用return返回了计算的结果，在外部调用这个函数的时候，就可以接收到结果。  
有时候函数是没有返回结果的，这个时候从函数获取到的是一个空值None。  
我们对list\_sum()这个函数进行简单的修改，在函数内把结果打印出来，不通过return返回结果。

def list\_sum(l):

result = 0

for num in l:

result = result + num

print('result is {}'.format(result))

return

l = [1, 3, 5, 7, 9, 11]

result =list\_sum(l) # 调用定义的sum\_list函数并获得return返回的结果

print(result) # ==> None

在print(result)中，我们得到None的结果，这是合理的，因为在函数内部，我们把结果打印出来了，但是没有把结果返回。  
除了返回None、一个值以外，函数也可以返回多个值，在函数中，如果需要返回多个值，多个值之间使用逗号分隔即可，但是需要注意顺序。  
比如，定义一个函数data\_of\_square，接收边长一个参数，同时返回正方形的周长和面积。

def data\_of\_square(side):

C = 4 \* side

S = side \* side

return C, S

C, S = data\_of\_square(16)

print('周长 = {}'.format(C)) # ==> 周长 = 64

print('面积 = {}'.format(S)) # ==> 面积 = 256

也可以使用一个值存储函数返回的多值结果。

result = data\_of\_square(16)

print(result) # ==> (64, 256)

注意打印的result，其实它是tuple类型，如果我们需要取出结果中的周长或者面积，使用对应位置的下标就可以获得对应的结果。

result = data\_of\_square(16)

C = result[0]

S = result[1]

print('周长 = {}'.format(C)) # ==> 周长 = 64

print('面积 = {}'.format(S)) # ==> 面积 = 256

## Python递归函数

在函数内部，还可以调用其他函数，比如实现函数data\_of\_square的时候，它接收边长一个参数，同时返回正方形的周长和面积，而求周长和求面积是完全独立的逻辑，可以定义成两个新的函数，然后在data\_of\_square函数中再调用这两个函数，得到结果并返回。

def square\_area(side):

return side \* side

def square\_perimeter(side):

return 4 \* side

def data\_of\_square(side):

C = square\_perimeter(side)

S = square\_area(side)

return C, S

在函数内部调用其他函数，是非常常见的，通过合理拆分逻辑，可以降低程序的复杂度。如果在一个函数内部调用其自身，这个函数就是递归函数。  
举个例子，我们来计算阶乘 n! = 1 \* 2 \* 3 \* ... \* n，用函数 fact(n)表示，可以看出：

fact(n) = n! = 1 \* 2 \* 3 \* ... \* (n-1) \* n = (n-1)! \* n = fact(n-1) \* n

所以，fact(n)可以表示为 n \* fact(n-1)，只有n=1时需要特殊处理。  
于是，fact(n)用递归的方式写出来就是：

def fact(n):

if n==1:

return 1

return n \* fact(n - 1)

这个fact(n)就是递归函数，可以试试计算得到的结果。

fact(1) # ==> 1

fact(5) # ==> 120

我们可以拆解fact(5)计算的详细逻辑

===> fact(5)

===> 5 \* fact(4)

===> 5 \* (4 \* fact(3))

===> 5 \* (4 \* (3 \* fact(2)))

===> 5 \* (4 \* (3 \* (2 \* fact(1))))

===> 5 \* (4 \* (3 \* (2 \* 1)))

===> 5 \* (4 \* (3 \* 2))

===> 5 \* (4 \* 6)

===> 5 \* 24

===> 120

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。  
使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试计算 fact(10000)。

Traceback (most recent call last):

RecursionError: maximum recursion depth exceeded in comparison

请分别使用循环和递归的形式定义函数，求出1~100的和。

def sumA(n):

sum = 0

x=0

while x<=n:

sum = sum + x

x = x+1

return sum

print(sumA(10))

def sumB(n):

sum = 0

if n==1:

sum =n

else:

sum = n + sumB(n-1)

return sum

print (sumB(10))

## Python函数参数

函数参数是需要传递给函数内部的数据，在前面，我们已经简单接触了函数的参数，现在我们正式来认识它。  
函数参数可以是任意的数据类型，只要函数内部逻辑可以处理即可。

def print\_param(param):

print(param)

对于print\_param函数，由于函数的逻辑是直接打印参数，并没有做任何别的逻辑，所以这个函数可以接受整数、浮点数、list、tuple、dict等等的数据类型。

print\_param(1)

print\_param('3.1415926')

print\_param([1, 2, 3, 4, 5])

但是，有时候由于函数的实现关系，需要特定的参数，就比如前面实现的求绝对值的函数my\_abs()，如果传递一个字符串，就会引起错误。

def my\_abs(x):

if x >= 0:

return x

else:

return -x

my\_abs('str param')

# 报错

TypeError: '>=' not supported between instances of 'str' and 'int'

为了保证函数的正常运行，有时候需要对函数入参进行类型的校验，Python提供isinstance()函数，可以判断参数类型，它接收两个参数，第一个是需要判断的参数，第二个是类型。

isinstance(100, int) # ==> True

isinstance(100.0, int) # ==> False

isinstance('3.1415926', str) # ==> True

有了isinstance，就可以优化my\_abs函数，不在里面运行出错了。

def my\_abs(x):

if not isinstance(x, int) or not isinstance(x, float):

print('param type error.')

return None

if x >= 0:

return x

else:

return -x

## Python函数使用默认参数

定义函数的时候，还可以有默认参数，默认参数的意思是当这个参数没有传递的时候，参数就使用定义时的默认值。  
例如Python自带的 int() 函数，其实就有两个参数，我们既可以传一个参数，又可以传两个参数：

int('123') # ==> 123

int('123', 8) # ==> 83

int()函数的第二个参数是转换进制base，如果不传，默认是十进制 (base=10)，如果传了，就用传入的参数。  
可见，函数的默认参数的作用是简化调用，你只需要把必须的参数传进去。但是在需要的时候，又可以传入额外的参数来覆盖默认参数值。  
我们来定义一个计算 x 的N次方的函数:

def power(x, n):

s = 1

while n > 0:

n = n - 1

s = s \* x

return s

假设计算平方的次数最多，我们就可以把 n 的默认值设定为 2：

def power(x, n=2):

s = 1

while n > 0:

n = n - 1

s = s \* x

return s

这样一来，计算平方就不需要传入两个参数了：

power(5) # ==> 25

另外需要注意的是，由于函数的参数按从左到右的顺序匹配，所以默认参数只能定义在必需参数的后面，否则将会出现错误。

# 错误的定义

def power(n=2, x):

s = 1

while n > 0:

n = n - 1

s = s \* x

return s

### 任务

请定义一个 greet() 函数，它包含一个默认参数，如果没有传入参数，打印 Hello, world.，如果传入参数，打印Hello, 传入的参数内容.

**?**不会了怎么办

参考答案：

def greet(name='world'):

print ('Hello, ' + name + '.')

greet()

greet('Alice')

## Python函数使用可变参数

除了默认参数，Python函数还接收一种参数叫做可变参数，可变参数即任意个参数的意思，可变参数通常使用\*args来表示。

def func(\*args):

print('args length = {}, args = {}'.format(len(args), args))

func('a') # ==> args length = 1, args = ('a',)

func('a', 'b') # ==> args length = 2, args = ('a', 'b')

func('a', 'b', 'c') # ==> args length = 3, args = ('a', 'b', 'c')

注意，在使用上，Python会把可变参数定义为一个tuple，所以在函数内部，把可变参数当作tuple来使用就可以了，比如可以通过位置下标取出对应的元素等。  
定义可变参数的目的也是为了简化调用。假设我们要计算任意个数的平均值，就可以定义一个可变参数：

def average(\*args):

sum = 0

for item in args:

sum += item

avg = sum / len(args)

return avg

这样，在调用的时候，我们就可以这样写：

average(1, 2) # ==> 1.5

average(1, 2, 2, 3, 4) # ==> 2.4

average()

# 报错

Traceback (most recent call last):

ZeroDivisionError: division by zero

在执行average()的时候，却报错了，这是因为在使用可变参数时，没有考虑周全导致的，因为可变参数的长度可能是0，当长度为0的时候，就会出现除0错误。因此需要添加保护的逻辑，这是同学在使用过程中需要特别注意的。

使用Python数据处理时，经常要对数据保留几位小数：  
eg:对x保留两位小数：

round(x,2)

## Python函数使用可变参数

除了默认参数，Python函数还接收一种参数叫做可变参数，可变参数即任意个参数的意思，可变参数通常使用\*args来表示。

def func(\*args):

print('args length = {}, args = {}'.format(len(args), args))

func('a') # ==> args length = 1, args = ('a',)

func('a', 'b') # ==> args length = 2, args = ('a', 'b')

func('a', 'b', 'c') # ==> args length = 3, args = ('a', 'b', 'c')

注意，在使用上，Python会把可变参数定义为一个tuple，所以在函数内部，把可变参数当作tuple来使用就可以了，比如可以通过位置下标取出对应的元素等。  
定义可变参数的目的也是为了简化调用。假设我们要计算任意个数的平均值，就可以定义一个可变参数：

def average(\*args):

sum = 0

for item in args:

sum += item

avg = sum / len(args)

return avg

这样，在调用的时候，我们就可以这样写：

average(1, 2) # ==> 1.5

average(1, 2, 2, 3, 4) # ==> 2.4

average()

# 报错

Traceback (most recent call last):

ZeroDivisionError: division by zero

在执行average()的时候，却报错了，这是因为在使用可变参数时，没有考虑周全导致的，因为可变参数的长度可能是0，当长度为0的时候，就会出现除0错误。因此需要添加保护的逻辑，这是同学在使用过程中需要特别注意的

## Python函数使用可变关键字参数

可变参数在使用上确实方便，函数会把可变参数当作tuple去处理，tuple在使用上有一定的局限性，比如有时候想找到特定位置的参数，只能通过下标的方式去寻找，如果顺序发生变化得时候，下标就会失效，函数逻辑就得重新修改实现。  
Python函数提供可变关键字参数，对于可变关键字参数，可以通过关键字的名字key找到对应的参数值，想想这和我们之前学习过的什么类似？是的没错，dict，Python会把可变关键字参数当作dict去处理；对于可变关键字参数，一般使用\*\*kwargs来表示。  
例如，想要打印一个同学的信息，可以这样处理：

def info(\*\*kwargs):

print('name: {}, gender: {}, age: {}'.format(kwargs.get('name'), kwargs.get('gender'), kwargs.get('age')))

info(name = 'Alice', gender = 'girl', age = 16)

对于一个拥有必需参数，默认参数，可变参数，可变关键字参数的函数，定义顺序是这样的：

def func(param1, param2, param3 = None, \*args, \*\*kwargs):

print(param1)

print(param2)

print(param3)

print(args)

print(kwargs)

func(100, 200, 300, 400, 500, name = 'Alice', score = 100)

# ==> 100

# ==> 200

# ==> 300

# ==> (400, 500)

# ==> {'name': 'Alice', 'score': 100}

当然，这么多类型的参数，很容易导致出错，在实际使用上，不建议定义这么多的参数。

**Python3 list 排序函数详解**

一、           列表的sort排序函数

**函数原型：**

         list.sort(key=None,reverse=False)

**函数功能：**

对原列表进行排序，完成排序后，原列表变为有序列表。默认情况（不传入任何参数时）按字典顺序排序。

**函数参数：**

(1)     key: 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中。指定可迭代对象中的一个元素来进行排序指定排序时使用的单个元素或多个元素、lambda表达式；

(2)     reverse：指字排序规则是升序还是降序，默认为升序排序；

二、           排序示例

1.      字符串排序

1. def list\_sort\_string():
2. list=["delphi","Delphi","python","Python","c++","C++","c","C","golang","Golang"]
3. list.sort() *#按字典顺序升序排列*
4. print("升序:",list)
6. list.sort(reverse=True) *#按降序排列*
7. print("降序:",list)

排序结果：

1. 升序: ['C', 'C++', 'Delphi', 'Golang', 'Python', 'c', 'c++', 'delphi', 'golang', 'python']
2. 降序: ['python', 'golang', 'delphi', 'c++', 'c', 'Python', 'Golang', 'Delphi', 'C++', 'C']

2.      数值型排序

1. def list\_sort\_number():
2. list=[30,40,10,50,50.1,80,60,100,90]
3. list.sort()
4. print("升序:",list)
6. list.sort(reverse=True)
7. print("降序:",list)

排序结果：

1. 升序: [10, 30, 40, 50, 50.1, 60, 80, 90, 100]
2. 降序: [100, 90, 80, 60, 50.1, 50, 40, 30, 10]

3.      根据列表中类对象的属性排序

1. class element(object):
2. def \_\_init\_\_(self,id="",name=""):
3. self.id=id
4. self.name=name
5. def \_\_lt\_\_(self, other): *# override <操作符*
6. if self.id<other.id:
7. return True
8. return False
10. def \_\_str\_\_(self): *# override \_\_str\_\_*
11. return "id={0},name={1}".format(self.id,self.name)
13. def sort\_by\_attribute():
14. list=[element(id="130",name="json"),
15. element(id="01",name="jack"),element(id="120",name="tom")]
16. list.sort()
17. for item in list: