**变量和数据类型**

数字

布尔

函数

运算符

| 函数 | 描述 |
| --- | --- |
| abs(number) | 返回数字的绝对值 |
| cmath.sqrt(number) | 返回平方根,可用于负数 |
| float(object) | 将字符串和数字转换为浮点数 |
| help() | 提供交互式帮助 |
| input(prompt) | 获取用户输入,参数为合法的python表达式 |
| int(object) | 将字符串和数字转换为整数 |
| long(object) | 将字符串和数字转换为长整型数 |
| math.ceil(number) | 向上取整,返回值为浮点数>>>math.ceil(1.2)  2 |
| math.floor(number) | 向下取整,返回值为浮点数>>> math.floor(0.7)  0 |
| math.sqrt(number) | 平方根,不能用于负数 |
| pow(x, y[, z]) | 返回x的y次幂(所得结果对z取模) |
| raw\_input(prompt) | 获取用户输入,返回的类型为字符串 |
| repr(object) | 返回值得字符串表示形式 |
| round(number[, ndigiths]) | 根据给定的精度对数字进行四舍五入 |
| str(object) | 将值转换为字符串 |
| cmp(x, y) | 比较两个值 |
| len(seq) | 返回序列的长度 |
| list(seq) | 把序列转换成列表 |
| max(args) | 返回序列或参数集合中的最大值 |
| min(args) | 返回序列或参数集合中的最小值 |
| reversed(seq) | 对序列进行反向迭代 |
| sorted(seq) | 返回已排序的包含seq所有元素的列表 |
| tuple(seq) | 把序列转换成元组 |

**Python变量和数据类型**

**数据类型**

**整数**

**浮点数**浮点数也就是小数，

**字符串**字符串是以''或""括起来的任意文本，比如'abc'，"xyz"等等。

**布尔值**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（1），要么是False（0），在Python中，可以直接用True、False表示布尔值（请注意大小写），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。

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。

None不能理解为0，因为0是有意义的，而None是一个特殊的空值。

**print语句**

print语句可以向屏幕上输出指定的文字。

**注意：**

* 当我们在Python交互式环境下编写代码时，>>>是Python解释器的提示符，不是代码的一部分。
* 当我们在文本编辑器中编写代码时，千万不要自己添加 >>>。

print语句也可以跟上多个字符串，用逗号,隔开，就可以连成一串输出：

print会依次打印每个字符串，知识点：**遇到逗号,会输出一个空格.**

**注释**

Python的注释以#开头，后面的文字直到行尾都算注释

**注释：多行注释**'''下面是一行被注释代码

下面是两行被注释代码

'''

**什么是变量**

知识点：**变量名必须是大小写英文、数字和下划线 \_ 的组合，且不能用数字开头。**

在Python中，等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量

**定义字符串**

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：

知识点：转义

这个时候，就需要对字符串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\".'

# 在要保留原状的字符串前面加上右斜杠

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 "I'm OK\n",
* \t 表示一个制表符
* \\ 表示 \ 字符本身

**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'\(~\_~)/ \(~\_~)/'

如何获得系统的默认编码？

#coding=utf-8

目的是告诉Python解释器，用UTF-8编码读取源代码。

Python3默认就是utf-8

**整数和浮点数**

Python支持对整数和浮点数直接进行四则混合运算，运算规则和数学上的四则运算规则完全一致。

基本的运算：

1 + 2 + 3 # ==> 6

4 \* 5 - 6 # ==> 14

7.5 / 8 + 2.1 # ==> 3.0375

使用括号可以提升优先级，这和数学运算完全一致，注意只能使用小括号，但是括号可以嵌套很多层：

(1 + 2) \* 3 # ==> 9

(2.2 + 3.3) / (1.5 \* (9 - 0.3)) # ==> 0.42145593869731807

和数学运算不同的地方是，Python的整数运算结果仍然是整数，浮点数运算结果仍然是浮点数：

1 + 2 # ==> 整数 3

1.0 + 2.0 # ==> 浮点数 3.0

但是整数和浮点数混合运算的结果就变成浮点数了：

1 + 2.0 # ==> 浮点数 3.0

为什么要区分整数运算和浮点数运算呢？

这是因为整数运算的结果永远是精确的，而浮点数运算的结果不一定精确，因为计算机内存再大，也无法精确表示出无限循环小数，比如 0.1 换成二进制表示就是无限循环小数。

那整数的除法运算遇到除不尽的时候，结果难道不是浮点数吗？我们来试一下：

11 / 4 # ==> 2

令很多初学者惊讶的是，Python的整数除法，即使除不尽，结果仍然是整数，余数直接被扔掉。不过，Python提供了一个**求余**的运算 % 可以计算余数：

11 % 4 # ==> 3

如果我们要计算 11 / 4 的精确结果，按照“整数和浮点数混合运算的结果是浮点数”的法则，把两个数中的一个变成浮点数再运算就没问题了：

11.0 / 4 # ==> 2.75

**Python集合类型:list和tuple**

**创建list**

Python内置的一种数据类型是列表：list。list是一种有序的集合，可以随时添加和删除其中的元素。

比如，列出班里所有同学的名字，就可以用一个list表示：

>>> ['Michael', 'Bob', 'Tracy']

['Michael', 'Bob', 'Tracy']

list是数学意义上的有序集合，也就是说，list中的元素是按照顺序排列的。

构造list非常简单，按照上面的代码，直接用 [ ]把list的所有元素都括起来，就是一个list对象。通常，我们会把list赋值给一个变量，这样，就可以通过变量来引用list：

>>> classmates = ['Michael', 'Bob', 'Tracy']

>>> classmates # 打印classmates变量的内容

['Michael', 'Bob', 'Tracy']

由于Python是动态语言，所以list中包含的元素并不要求都必须是同一种数据类型，我们完全可以在list中包含各种数据：

>>> L = ['Michael', 100, True]

一个元素也没有的list，就是空list：

>>> empty\_list = []

**Python按照索引访问list**

由于list是一个有序集合，所以，我们可以用一个list按分数从高到低表示出班里的3个同学：

>>> L = ['Adam', 'Lisa', 'Bart']

那我们如何从list中获取指定第 N 名的同学呢？方法是通过索引来获取list中的指定元素。

需要特别注意的是，**索引从 0 开始**，也就是说，第一个元素的索引是0，第二个元素的索引是1，以此类推。

因此，要打印第一名同学的名字，用 L[0]:

所以，使用索引时，**千万注意不要越界**。

知识点：正序从0开始，逆序从-1开始是最好一个list内容。

当索引数字为负数时，表示逆序读出List中的内容，记住List的最后一个空间的编号为-1开始

**倒序访问list**

使用倒序索引时，也要注意**不要越界。**

**list添加新元素(append insert)**

第一个办法是用 list 的 append() 方法，把新同学追加到 list 的末尾：

>>> L = ['Adam', 'Lisa', 'Bart']

>>> L.append('Paul')

>>> print L

['Adam', 'Lisa', 'Bart', 'Paul']

如果 Paul 同学表示自己总是考满分，要求添加到第一的位置，怎么办？

方法是用list的 insert()方法，它接受两个参数，第一个参数是索引号，第二个参数是待添加的新元素：

>>> L = ['Adam', 'Lisa', 'Bart']

>>> L.insert(0, 'Paul')

>>> print L

['Paul', 'Adam', 'Lisa', 'Bart']

L.insert(0, 'Paul') 的意思是，'Paul'将被添加到索引为 0 的位置上（也就是第一个），而原来索引为 0 的Adam同学，以及后面的所有同学，都自动向后移动一位。

**list删除元素(pop)**

Paul同学刚来几天又要转走了，那么我们怎么把Paul 从现有的list中删除呢？

如果Paul同学排在最后一个，我们可以用list的pop()方法删除：

>>> L = ['Adam', 'Lisa', 'Bart', 'Paul']

>>> L.pop()

'Paul'

>>> print L

['Adam', 'Lisa', 'Bart']

pop()方法总是**删掉list的最后一个元素，并且它还返回这个元素，**所以我们执行 L.pop() 后，会打印出 'Paul'。

如果Paul同学不是排在最后一个怎么办？比如Paul同学排在第三：

>>> L = ['Adam', 'Lisa', 'Paul', 'Bart']

要把Paul踢出list，我们就必须先定位Paul的位置。由于Paul的索引是2，因此，用 pop(2)把Paul删掉：

>>> L.pop(2)

'Paul'

>>> print L

['Adam', 'Lisa', 'Bart']

两种方式：直接pop()默认删除第一个，括号内指定参数：索引，删除索引位置上。

**创建tuple**

tuple是另一种有序的列表，中文翻译为“ 元组 ”。tuple 和 list 非常类似，但是，知识点：**tuple一旦创建完毕，就不能修改了。**

* [算术运算符](http://www.runoob.com/python3/python3-basic-operators.html#ysf1)
* [比较（关系）运算符](http://www.runoob.com/python3/python3-basic-operators.html#ysf2)
* [赋值运算符](http://www.runoob.com/python3/python3-basic-operators.html#ysf3)
* [逻辑运算符](http://www.runoob.com/python3/python3-basic-operators.html#ysf4)
* [位运算符](http://www.runoob.com/python3/python3-basic-operators.html#ysf5)
* [成员运算符](http://www.runoob.com/python3/python3-basic-operators.html#ysf6)
* [身份运算符](http://www.runoob.com/python3/python3-basic-operators.html#ysf7)
* [运算符优先级](http://www.runoob.com/python3/python3-basic-operators.html#ysf8)

接下来让我们一个个来

## Python算术运算符

以下假设变量a为10，变量b为21：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| + | 加 ： 两个对象相加 | a + b 输出结果 31 |
| - | 减 ： 得到负数或是一个数减去另一个数 | a - b 输出结果 -11 |
| \* | 乘 ： 两个数相乘或是返回一个被重复若干次的字符串 | a \* b 输出结果 210 |
| / | 除 ：x 除以 y | b / a 输出结果 2.1 |
| % | 取模 ： 返回除法的余数 | b % a 输出结果 1 |
| \*\* | 幂 ： 返回x的y次幂 | a\*\*b 为10的21次方 |
| // | 取整除 ： 向下取接近除数的整数 | >>> 9//2  4  >>> -9//2  -5 |

## 比较运算符

以下假设变量a为10，变量b为20：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **实例** |
| == | 等于 - 比较对象是否相等 | (a == b) 返回 False。 |
| != | 不等于 - 比较两个对象是否不相等 | (a != b) 返回 True。 |
| > | 大于 - 返回x是否大于y | (a > b) 返回 False。 |
| < | 小于 - 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。 | (a < b) 返回 True。 |
| >= | 大于等于 - 返回x是否大于等于y。 | (a >= b) 返回 False。 |
| <= | 小于等于 - 返回x是否小于等于y。 | (a <= b) 返回 True。 |

以下实例演示了Python所有比较

Python语言支持逻辑运算符，以下假设变量 a 为 10, b为 20:

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **逻辑表达式** | **描述** | **实例** |
| 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。 | (a and b) 返回 20。 |
| or | x or y | 布尔"或" - 如果 x 是 True，它返回 x 的值，否则它返回 y 的计算值。 | (a or b) 返回 10。 |
| not | not x | 布尔"非" - 如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。 | not(a and b) 返回 False |

以上实例输出结果：

以下表格列出了从最高到最低优先级的所有运算符：

|  |  |
| --- | --- |
| **运算符** | **描述** |
| \*\* | 指数 (最高优先级) |
| ~ + - | 按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@) |
| \* / % // | 乘，除，取模和取整除 |
| + - | 加法减法 |
| >> << | 右移，左移运算符 |
| & | 位 'AND' |
| ^ | | 位运算符 |
| <= < > >= | 比较运算符 |
| == != | 等于运算符 |
| = %= /= //= -= += \*= \*\*= | 赋值运算符 |
| is is not | 身份运算符 |
| in not in | 成员运算符 |
| and or not | 逻辑运算符 |

以下实例演示了Python所有运算符优

**Python的条件判断和循环语句**

**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。

比如，输入用户年龄，根据年龄打印不同的内容，在Python程序中，可以用if语句实现：

age = 20

if age >= 18:

print 'your age is', age

print 'adult'

print 'END'

注意: Python代码的缩进规则。**具有相同缩进的代码被视为代码块，**上面的3，4行 print 语句就构成一个代码块（但不包括第5行的print）。如果 if 语句判断为 True，就会执行这个代码块。

知识点: **缩进请严格按照Python的习惯写法：4个空格，不要使用Tab**，更不要混合Tab和空格，否则很容易造成因为缩进引起的语法错误。

注意: if 语句后接表达式，然后用:表示代码块开始。

如果你在Python交互环境下敲代码，还要特别留意缩进，并且退出缩进需要多敲一行回车：

>>> age = 20

>>> if age >= 18:

... print 'your age is', age

... print 'adult'

...

your age is 20

adult

**if-else**

当 if 语句判断表达式的结果为 True 时，就会执行 if 包含的代码块：

if age >= 18:

print 'adult'

如果我们想判断年龄在18岁以下时，打印出 'teenager'，怎么办？

方法是再写一个 if:

if age < 18:

print 'teenager'

或者用 not 运算：

if not age >= 18:

print 'teenager'

细心的同学可以发现，这两种条件判断是“非此即彼”的，要么符合条件1，要么符合条件2，因此，完全可以用一个 if ... else ... 语句把它们统一起来：

if age >= 18:

print 'adult'

else:

print 'teenager'

利用 if ... else ...语句，我们可以根据条件表达式的值为 True 或者 False ，分别执行 if 代码块或者 else 代码块。

注意: else 后面有个:。

failed

**if-elif-else**

有的时候，一个 if ... else ... 还不够用。比如，根据年龄的划分：

* 条件1：18岁或以上：adult
* 条件2：6岁或以上：teenager
* 条件3：6岁以下：kid

**要避免嵌套结构的 if ... else ...，**我们可以用 if ... 多个elif ... else ... 的结构，一次写完所有的规则：

if age >= 18:

print 'adult'

elif age >= 6:

print 'teenager'

elif age >= 3:

print 'kid'

else:

print 'baby'

elif 意思就是 else if。这样一来，我们就写出了结构非常清晰的一系列条件判断。

**特别注意**: 这一系列条件判断会从上到下依次判断，如果某个判断为 True，执行完对应的代码块，后面的条件判断就直接忽略，不再执行了。

请思考下面的代码：

age = 8

if age >= 6:

print 'teenager'

elif age >= 18:

print 'adult'

else:

print 'kid'

当 age = 8 时，结果正确，但 age = 20 时，为什么没有打印出 adult？

如果要修复，应该如何修复？

知识点解答: 因为当age=20.第一个条件>=6满足就短路了。  
因此我们在设置条件应该从严格到松泛.

age = 20

if age >= 18:

print 'teenager'

elif age >= 6:

print 'adult'

else:

print 'kid'

**for循环**

list或tuple可以表示一个有序集合。如果我们想依次访问一个list中的每一个元素呢？比如 list：

L = ['Adam', 'Lisa', 'Bart']

print L[0]

print L[1]

print L[2]

如果list只包含几个元素，这样写还行，如果list包含1万个元素，我们就不可能写1万行print。

这时，循环就派上用场了。

Python的 for 循环就可以依次把list或tuple的每个元素迭代出来：

L = ['Adam', 'Lisa', 'Bart']

for name in L:

print name

注意: name 这个变量是在 for 循环中定义的(这是一个临时变量名字可自定义)，意思是，依次取出list中的每一个元素，并把元素赋值给 name，然后执行for循环体（就是缩进的代码块）。

这样一来，遍历一个list或tuple就非常容易了。

**编程任务**

班里考试后，老师要统计平均成绩，已知4位同学的成绩用list表示如下：L = [75, 92, 59, 68]

请利用for循环计算出平均成绩。

实现代码:

L = [75, 92, 59, 68]

sum = 0.0

for x in L:

sum =sum+x

print sum / 4

运行结果：

73.5

**while循环**

和 for 循环不同的另一种循环是 while 循环，while 循环不会迭代 list 或 tuple 的元素，而是根据表达式判断循环是否结束。

比如要从 0 开始打印不大于 N 的整数：

N = 10

x = 0

while x < N:

print x

x = x + 1

while循环每次先判断 x < N，如果为True，则执行循环体的代码块,否则，退出循环。

在循环体内，x = x + 1 会让 x 不断增加，最终因为 x < N 不成立而退出循环。

如果没有这一个语句，while循环在判断 x < N 时总是为True，就会无限循环下去，变成死循环，所以要特别留意while循环的**退出条件。**

**编程任务**

利用while循环计算100以内奇数的和。

实现代码:

sum = 0

x = 1

while x<=100:

sum=sum+x

x=x+2

print sum

知识点: 奇数只需要从1开始不断加2都是奇数。

运行结果：

2500

**break退出循环**

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。

比如计算1至100的整数和，我们用while来实现：

sum = 0

x = 1

while True:

sum = sum + x

x = x + 1

if x > 100:

break

print sum

咋一看， while True 就是一个死循环，但是在循环体内，我们还判断了 x > 100 条件成立时，用break语句退出循环，这样也可以实现循环的结束。

**continue继续循环**

在循环过程中，可以用break退出当前循环，还可以用continue跳过后续循环代码，继续下一次循环。

假设我们已经写好了利用for循环计算平均分的代码：

L = [75, 98, 59, 81, 66, 43, 69, 85]

sum = 0.0

n = 0

for x in L:

sum = sum + x

n = n + 1

print sum / n

现在老师只想统计及格分数的平均分，就要把 x < 60 的分数剔除掉，这时，利用continue，可以做到当 x < 60的时候，不继续执行循环体的后续代码，直接进入下一次循环：

for x in L:

if x < 60:

continue

sum = sum + x

n = n + 1

coutinue: 跳过下面的代码。开始下一次循环。

**重要的数据类型Dict和Set**

**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：

['Adam', 'Lisa', 'Bart']

或者考试的成绩列表：

[95, 85, 59]

但是，要根据名字找到对应的成绩，用两个 list 表示就不方便。

如果把名字和分数关联起来，组成类似的查找表：

'Adam' ==> 95

'Lisa' ==> 85

'Bart' ==> 59

给定一个名字，就可以直接查到分数。

Python的 dict 就是专门干这件事的。用 dict 表示名字-成绩的查找表如下：

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

我们把名字称为key，对应的成绩称为value，dict就是通过 key来查找 value。

花括号 {} 表示这是一个dict，然后按照 key: value, 写出来即可。**最后一个 key: value 的逗号可以省略。**

知识点： 区别小课堂

* 单元素的tuple必须在后面多加一个逗号。
* dict最后的逗号可以省略

由于dict也是集合，len() 函数可以计算任意集合的大小：

>>> len(d)

3

知识点：**注意: 一个 key-value 算一个，因此，dict大小为3。**

**编程任务**

新来的Paul同学成绩是 75 分，请编写一个dict，把Paul同学的成绩也加进去。

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

实现代码:

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59,

'Paul': 75

}

**访问dict**

我们已经能创建一个dict，用于表示名字和成绩的对应关系：

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

那么，如何根据名字来查找对应的成绩呢？

可以简单地使用 d[key] 的形式来查找对应的 value，这和 list 很像，不同之处是，list 必须使用索引返回对应的元素，而dict使用key：

>>> print d['Adam']

95

>>> print d['Paul']

Traceback (most recent call last):

File "index.py", line 11, in <module>

print d['Paul']

KeyError: 'Paul'

注意: 通过 key 访问 dict 的value，只要 key 存在，dict就返回对应的value。如果key不存在，会直接报错：KeyError。

知识点：避免 KeyError 发生，有两个办法：

* 是先判断一下 key 是否存在，用 in 操作符：

if 'Paul' in d:

print d['Paul']

如果 'Paul' 不存在，if语句判断为False，自然不会执行 print d['Paul'] ，从而避免了错误。

* 是使用dict本身提供的一个get方法，在Key不存在的时候，返回None：

>>> print d.get('Bart')

59

>>> print d.get('Paul')

None

**编程任务**

根据如下dict：

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

请打印出：  
Adam: 95  
Lisa: 85  
Bart: 59

实现代码:

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

print 'Adam:',d['Adam']

print 'Lisa:',d.get('Lisa')

print 'Bart:',d['Bart']

运行结果：

Adam: 95

Lisa: 85

Bart: 59

**dict的特点**

**知识点**：dict查找速度快。list查找速度随着元素增加而逐渐下降。  
缺点：内存占用大。list慢但内存占用小。

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。

由于dict是按 key 查找，所以，**在一个dict中，key不能重复。**

dict的第二个特点就是存储的key-value序对是没有顺序的！这和list不一样：

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

当我们试图打印这个dict时：

>>> print d

{'Lisa': 85, 'Adam': 95, 'Bart': 59}

打印的顺序不一定是我们创建时的顺序，而且，不同的机器打印的顺序都可能不同，这说明 知识点:**dict内部是无序的，不能用dict存储有序的集合。**

知识点：dict的第三个特点是作为 key 的元素必须不可变，Python的基本类型如字符串、整数、浮点数都是不可变的，都可以作为 key。 但是list是可变的，就不能作为 key。

可以试试用list作为key时会报什么样的错误。

不可变这个限制仅作用于key，value是否可变无所谓：

{

'123': [1, 2, 3], # key 是 str，value是list

123: '123', # key 是 int，value 是 str

('a', 'b'): True # key 是 tuple，并且tuple的每个元素都是不可变对象，value是 boolean

}

最常用的key还是字符串，因为用起来最方便。

**编程任务**

请设计一个dict，可以根据分数来查找名字，已知成绩如下：

Adam: 95,

Lisa: 85,

Bart: 59.

实现代码:

d = {

95:'Adam',

85:'Lisa',

59:'Bart'

}

运行结果：无

**更新dict**

dict是可变的，也就是说，我们可以随时往dict中添加新的 key-value。比如已有dict：

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

要把新同学'Paul'的成绩 72 加进去，用赋值语句：

>>> d['Paul'] = 72

再看看dict的内容：

>>> print d

{'Lisa': 85, 'Paul': 72, 'Adam': 95, 'Bart': 59}

**如果 key 已经存在，则赋值会用新的 value 替换掉原来的 value：**

>>> d['Bart'] = 60

>>> print d

{'Lisa': 85, 'Paul': 72, 'Adam': 95, 'Bart': 60}

**编程任务**

请根据Paul的成绩 72 更新下面的dict：

d = {

95: 'Adam',

85: 'Lisa',

59: 'Bart'

}

实现代码:

d = {

95: 'Adam',

85: 'Lisa',

59: 'Bart'

}

d[72] = 'Paul'

print d

运行结果：

{72: 'Paul', 59: 'Bart', 85: 'Lisa', 95: 'Adam'}

**遍历dict**

由于dict也是一个集合，所以，遍历dict和遍历list类似，都可以通过 for 循环实现。

直接使用for循环可以遍历 dict 的 key：

>>> d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }

>>> for key in d:

... print key

...

Lisa

Adam

Bart

由于通过 key 可以获取对应的 value，因此，在循环体内，可以获取到value的值。

注：这里的key只是一个约定俗称的变量，可以改为其他名字。但是推荐用key。

**编程任务**

请用 for 循环遍历如下的dict，打印出 name: score 来。

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

实现代码：

d = {

'Adam': 95,

'Lisa': 85,

'Bart': 59

}

for key in d:

print key+":",d[key]

运行结果:

Lisa: 85

Adam: 95

Bart: 59

**什么是set**

dict的作用是建立一组 key 和一组 value 的映射关系，dict的key是**不能重复的**。

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 持有一系列元素，这一点和 list 很像，但是set的元素**没有重复，而且是无序的**，这点和 dict 的 key很像。

知识点: 创建 set 的方式是**调用 set() 并传入一个 list**，list的元素将作为set的元素：

>>> s = set(['A', 'B', 'C'])

可以查看 set 的内容：

>>> print s

set(['A', 'C', 'B'])

请注意，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。**

因为set不能包含重复的元素，所以，当我们传入包含重复元素的 list 会怎么样呢？

>>> s = set(['A', 'B', 'C', 'C'])

>>> print s

set(['A', 'C', 'B'])

>>> len(s)

3

结果显示，set会**自动去掉重复的元素**，原来的list有4个元素，但set只有3个元素。

**编程任务**

请用set表示班里的4位同学：Adam, Lisa, Bart, Paul

实现代码:

s = set(['Adam', 'Lisa', 'Bart', 'Paul'])

print s

运行结果:

set(['Lisa', 'Paul', 'Adam', 'Bart'])

**访问set**

由于set存储的是**无序集合**，所以我们**没法通过索引**来访问。

访问 set中的某个元素实际上就是判断一个元素是否在set中。

例如，存储了班里同学名字的set：

>>> s = set(['Adam', 'Lisa', 'Bart', 'Paul'])

我们可以用 in 操作符判断：

Bart是该班的同学吗？

>>> 'Bart' in s

True

Bill是该班的同学吗？

>>> 'Bill' in s

False

bart是该班的同学吗？

>>> 'bart' in s

False

知识点：**大小写很重要**，'Bart' 和 'bart'被认为是两个不同的元素。

**编程任务**

由于上述set不能识别小写的名字，请改进set，使得 'adam' 和 'bart'都能返回True。

既然大小写是不同的。那我们的set中就把大小写都包含。

实现代码:

s = set(['Adam', 'Lisa', 'Bart', 'Paul','adam', 'lisa', 'bart', 'paul'])

print 'adam' in s

print 'bart' in s

运行结果.

True

True

**set的特点**

**set的内部结构和dict很像，唯一区别是不存储value**，因此，判断一个元素是否在set中速度很快。

**set存储的元素和dict的key类似，必须是不变对象**，因此，任何可变对象是不能放入set中的。

最后，set存储的元素也是没有顺序的。

set的这些特点，可以应用在哪些地方呢？

星期一到星期日可以用字符串'MON', 'TUE', ... 'SUN'表示。

假设我们让用户输入星期一至星期日的某天，如何判断用户的输入是否是一个有效的星期呢？

可以用 if 语句判断，但这样做非常繁琐：

x = '???' # 用户输入的字符串

if x!= 'MON' and x!= 'TUE' and x!= 'WED' ... and x!= 'SUN':

print 'input error'

else:

print 'input ok'

注意：if 语句中的...表示没有列出的其它星期名称，测试时，请输入完整。

如果事先创建好一个set，包含'MON' ~ 'SUN'：

weekdays = set(['MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT', 'SUN'])

再判断输入是否有效，只需要判断该字符串是否在set中：

x = '???' # 用户输入的字符串

if x in weekdays:

print 'input ok'

else:

print 'input error'

这样一来，代码就简单多了。

**编程任务**

月份也可以用set表示，请设计一个set并判断用户输入的月份是否有效。  
月份可以用字符串'Jan', 'Feb', ...表示。

实现代码:

months = set(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul','Aug','Sep','Oct','Nov','Dec'])

x1 = 'Feb'

x2 = 'Sun'

if x1 in months:

print 'x1: ok'

else:

print 'x1: error'

if x2 in months:

print 'x2: ok'

else:

print 'x2: error'

运行结果:

x1: ok

x2: error

**遍历set**

由于 set 也是一个集合，所以，遍历 set 和遍历 list 类似，都可以通过 for 循环实现。

直接使用 for 循环可以遍历 set 的元素：

>>> s = set(['Adam', 'Lisa', 'Bart'])

>>> for name in s:

... print name

...

Lisa

Adam

Bart

注意: 观察 for 循环在遍历set时，**元素的顺序和list的顺序很可能是不同的，**而且不同的机器上运行的结果也可能不同。

**编程任务**

请用 for 循环遍历如下的set，打印出 name: score 来。

s = set([('Adam', 95), ('Lisa', 85), ('Bart', 59)])

上面这个set中的每一个元素又是一个字典。

set([ ])是壳子。

('Adam', 95), ('Lisa', 85), ('Bart', 59)才是真正的内容

实现代码：

s = set([('Adam', 95), ('Lisa', 85), ('Bart', 59)])

for name,score in s:

print name,':',score

运行结果:

Lisa : 85

Adam : 95

Bart : 59

**更新set(add remove)**

由于set存储的是一组不重复的无序元素，因此，更新set主要做两件事：

* 是把新的元素添加到set中
* 是把已有元素从set中删除。(前提是如果有)

添加元素时，用set的add()方法：

>>> s = set([1, 2, 3])

>>> s.add(4)

>>> print s

set([1, 2, 3, 4])

如果添加的元素已经存在于set中，add()不会报错，但是不会加进去了：

>>> s = set([1, 2, 3])

>>> s.add(3)

>>> print s

set([1, 2, 3])

删除set中的元素时，用set的remove()方法：

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

>>> s.remove(4)

>>> print s

set([1, 2, 3])

如果删除的元素不存在set中，remove()会**报错**：

>>> s = set([1, 2, 3])

>>> s.remove(4)

Traceback (most recent call last):

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

KeyError: 4

所以用add()可以直接添加，而remove()前需要判断。

**编程任务**

针对下面的set，给定一个list，对list中的每一个元素，如果在set中，就将其删除，如果不在set中，就添加进去。

s = set(['Adam', 'Lisa', 'Paul'])

L = ['Adam', 'Lisa', 'Bart', 'Paul']

实现代码：

s = set(['Adam', 'Lisa', 'Paul'])

L = ['Adam', 'Lisa', 'Bart', 'Paul']

for name in L:

if name in s:

s.remove(name)

else:

s.add(name)

print s

**函数定义与调用**

Python之什么是函数  
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，就可以根据公式计算出面积。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

当代码出现有规律的重复的时候，你就需要当心了，每次写3.14 \* x \* x不仅很麻烦，而且，如果要把3.14改成3.14159265359的时候，得全部替换。

有了函数，我们就不再每次写s = 3.14 \* x \* x，而是写成更有意义的函数调用 s = area\_of\_circle(x)，而函数 area\_of\_circle本身只需要写一次，就可以多次调用。

**抽象**是数学中非常常见的概念。举个例子：

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：

100

∑n

n=1

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：

100

∑(n²+1)

n=1

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。

**编程任务**

写一个函数

实现代码：

s = area\_of\_circle(x)

area\_of\_circle(x)

运行结果：

**调用函数,内置函数**

Python内置了很多有用的函数，我们可以直接调用。

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。

可以直接从Python的官方网站查看文档：  
[http://docs.python.org/2/library/functions.html#abs](https://link.jianshu.com?t=http%3A%2F%2Fdocs.python.org%2F2%2Flibrary%2Ffunctions.html%23abs)

也可以在交互式命令行通过 help(abs)查看abs函数的帮助信息。

调用 abs 函数：

>>> abs(100)

100

>>> abs(-20)

20

>>> abs(12.34)

12.34

调用函数的时候，如果传入的参数数量不对，会报TypeError的错误，并且Python会明确地告诉你：abs()有且仅有1个参数，但给出了两个：

>>> abs(1, 2)

Traceback (most recent call last):

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

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

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报TypeError的错误，并且给出错误信息：str是**错误的参数类型**：

>>> abs('a')

Traceback (most recent call last):

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

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

而比较函数 cmp(x, y) 就需要两个参数，如果 x<y，返回 -1，如果 x==y，返回0，如果 x>y，返回 1：

>>> cmp(1, 2)

-1

>>> cmp(2, 1)

1

>>> cmp(3, 3)

0

Python内置的常用函数还包括**数据类型转换函数**，比如 int()函数可以把其他数据类型转换为整数：

>>> int('123')

123

>>> int(12.34)

12

str()函数把其他类型转换成 str：

>>> str(123)

'123'

>>> str(1.23)

'1.23'

**编程任务**

sum()函数接受一个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。

实现代码：

L = []

L = []

x = 1

while x <= 100:

L.append(x \* x)

x = x + 1

print sum(L)

运行结果:

338350

**编写函数**

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。

我们以自定义一个求绝对值的 my\_abs 函数为例：

def my\_abs(x):

if x >= 0:

return x

else:

return -x

请注意，**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。

知识点； **如果没有return语句，函数执行完毕后也会返回结果，**只是结果为 None。  
return None可以简写为return。

**编程任务**

请定义一个 square\_of\_sum 函数，它接受一个list，返回list中每个元素平方的和。

实现代码:

def square\_of\_sum(L):

sum = 0

for x in L:

sum = x\*x+sum

return sum

print square\_of\_sum([1, 2, 3, 4, 5])

print square\_of\_sum([-5, 0, 5, 15, 25])

运行结果:

55

900

**函数之返回"多值"**

函数可以返回多个值吗？答案是肯定的。

比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的坐标：

**import /from import / import as**

math包提供了sin()和 cos()函数，我们先用import引用它：

import math

math.sqrt(x)

from math import sqrt

sqrt(x)

from math import \*

sqrt(x)

**import as**

将导入对象赋值给变量

这样我们就可以同时获得返回值：

>>> x, y = move(100, 100, 60, math.pi / 6)

>>> print x, y

151.961524227 70.0

但其实这只是一种**假象，Python函数返回的仍然是单一值：**

>>> r = move(100, 100, 60, math.pi / 6)

>>> print r

(151.96152422706632, 70.0)

知识点：**用print打印返回结果，原来返回值是一个tuple！**

但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的值，所以，知识点：**Python的函数返回多值其实就是返回一个tuple，**但写起来更方便。

**编程任务**

一元二次方程的定义是：ax² + bx + c = 0

请编写一个函数，返回一元二次方程的两个解。

注意：Python的math包提供了sqrt()函数用于计算平方根。

实现代码:

import math

def quadratic\_equation(a, b, c):

t = math.sqrt(b\*b - 4\*a\*c)

return (-b + t) / (2 \* a),( -b - t )/ (2 \* a)

print quadratic\_equation(2, 3, 0)

print quadratic\_equation(1, -6, 5)

mark

运行结果:

(0.0, -1.5)

(5.0, 1.0)

**递归函数**

在函数内部，可以调用其他函数。知识点: **如果一个函数在内部调用自身本身，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)

上面就是一个递归函数。可以试试：

>>> fact(1)

1

>>> fact(5)

120

>>> fact(100)

93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

如果我们计算fact(5)，可以根据函数定义看到计算过程如下：

===> 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**。知识点: 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。

知识点: 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)。

**编程任务(天涯)**

汉诺塔 ([http://baike.baidu.com/view/191666.htm](https://link.jianshu.com?t=http%3A%2F%2Fbaike.baidu.com%2Fview%2F191666.htm)) 的移动也可以看做是递归函数。

我们对柱子编号为a, b, c，将所有圆盘从a移到c可以描述为：

如果a只有一个圆盘，可以直接移动到c；

如果a有N个圆盘，可以看成a有1个圆盘（底盘） + (N-1)个圆盘，首先需要把 (N-1) 个圆盘移动到 b，然后，将 a的最后一个圆盘移动到c，再将b的(N-1)个圆盘移动到c。

请编写一个函数，给定输入 n, a, b, c，打印出移动的步骤：

move(n, a, b, c)

例如，输入 move(2, 'A', 'B', 'C')，打印出：

A --> B

A --> C

B --> C

实现代码：

def move(n, a, b, c):

if n ==1:

print a, '-->', c

return

move(n-1, a, c, b)

print a, '-->', c

move(n-1, b, a, c)

move(4, 'A', 'B', 'C')

运行结果:

A --> B

A --> C

B --> C

A --> B

C --> A

C --> B

A --> B

A --> C

B --> C

B --> A

C --> A

B --> C

A --> B

A --> C

B --> C

**定义默认参数**

定义函数的时候，还可以有默认参数。

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

知识点: int()函数的第二个参数是**转换进制**，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

知识点: 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：**

# OK:

def fn1(a, b=1, c=2):

pass

# Error:

def fn2(a=1, b):

pass

个人: 这里我们可以把自己想象成计算机。在自己感到为难不知道哪个是哪个的时候。  
那么恭喜你，计算机也不知道。

**编程任务**

请定义一个 greet()函数，它包含一个默认参数，如果没有传入，打印 'Hello, world.'，如果传入，打印 'Hello, xxx.'

实现代码:

def greet(x = 'World'):

print 'Hello,'+x+'.'

greet()

greet('mtianyan')

运行结果:

Hello,World.

Hello,mtianyan.

**知识点: 定义可变参数**

如果想让一个函数能接受任意个参数，我们就可以定义一个可变参数：

def fn(\*args):

print args

可变参数的名字前面有个 \* 号，我们可以传入0个、1个或多个参数给可变参数：

>>> fn()

()

>>> fn('a')

('a',)

>>> fn('a', 'b')

('a', 'b')

>>> fn('a', 'b', 'c')

('a', 'b', 'c')

可变参数也不是很神秘，Python解释器会把传入的一组参数组装成一个tuple传递给可变参数，因此，在函数内部，直接把变量 args 看成一个 tuple 就好了。

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):

...

这样，在调用的时候，可以这样写：

>>> average()

0

>>> average(1, 2)

1.5

>>> average(1, 2, 2, 3, 4)

2.4

**编程任务**

请编写接受可变参数的 average() 函数。

def average(\*args):

sum = 0.0

if len(args) == 0:

return sum

for x in args:

sum = sum + x

return sum / len(args)

print average()

print average(1, 2)

print average(1, 2, 2, 3, 4)

运行结果:

0.0

1.5

2.4

**切片操作**

**对list进行切片**

取一个list的**部分元素**是非常常见的操作。比如，一个list如下：

>>> L = ['Adam', 'Lisa', 'Bart', 'Paul']

取前3个元素，应该怎么做？

笨办法：

>>> [L[0], L[1], L[2]]

['Adam', 'Lisa', 'Bart']

之所以是笨办法是因为扩展一下，取前N个元素就没辙了。

取前N个元素，也就是索引为0-(N-1)的元素，可以用循环：

>>> r = []

>>> n = 3

>>> for i in range(n):

... r.append(L[i])

...

>>> r

['Adam', 'Lisa', 'Bart']

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，Python提供了切片（Slice）操作符，能大大简化这种操作。

对应上面的问题，取前3个元素，用一行代码就可以完成切片：

>>> L[0:3]

['Adam', 'Lisa', 'Bart']

L[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3。即索引0，1，2，正好是3个元素。

知识点： **[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3。即索引0，1，2，正好是3个元素。**

* 如果第一个索引是0，还可以省略：

>>> L[:3]

['Adam', 'Lisa', 'Bart']

* 也可以从索引1开始，取出2个元素出来：

>>> L[1:3]

['Lisa', 'Bart']

* 只用一个 : ，表示从头到尾：

>>> L[:]

['Adam', 'Lisa', 'Bart', 'Paul']

因此，L[:]实际上**复制出了一个新list。**

知识点: 切片操作还可以指定第三个参数：

>>> L[::2]

['Adam', 'Bart']

第三个参数表示**每N个取一个**，上面的 L[::2] 会每两个元素取出一个来，也就是隔一个取一个。

把list换成tuple，切片操作完全相同，只是切片的结果也变成了tuple。

**编程任务**

range()函数可以创建一个数列：

>>> range(1, 101)

[1, 2, 3, ..., 100]

请利用切片，取出：

1. 前10个数；
2. 3的倍数；
3. 不大于50的5的倍数。

实现代码:

L = range(1, 101)

print L[:10]

print L[2::3]

print L[4:50:5]

运行结果:

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

[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]

[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

**倒序切片**

对于list，既然Python支持L[-1]取倒数第一个元素，那么它同样支持倒数切片，试试：

>>> L = ['Adam', 'Lisa', 'Bart', 'Paul']

>>> L[-2:]

['Bart', 'Paul']

>>> L[:-2]

['Adam', 'Lisa']

>>> L[-3:-1]

['Lisa', 'Bart']

>>> L[-4:-1:2]

['Adam', 'Bart']

记住倒数第一个元素的索引是-1。知识点：**倒序切片包含起始索引，不包含结束索引。**

**编程任务**

利用倒序切片对 1 - 100 的数列取出：

* 最后10个数；
* 最后10个5的倍数。

实现代码：

L = range(1, 101)

print L[-10:]

print L[-46::5]

**1、字符串**

定义：它是一个有序的字符的集合，用于存储和表示基本的文本信息，‘’或“”或‘’‘ ’‘’中间包含的内容称之为字符串  
特性：  
1.只能存放一个值  
2.不可变  
3.按照从左到右的顺序定义字符集合，下标从0开始顺序访问，有序  
补充：  
　　1.字符串的单引号和双引号都无法取消特殊字符的含义，如果想让引号内所有字符均取消特殊意义，在引号前面加r，如name＝r'l\thf'  
　　2.unicode字符串与r连用必需在r前面，如name＝ur'l\thf'

**2、字符串常用操作**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # 1字母处理：  .upper()    # 全部大写  .lower()    # 全部小写  .swapcase()    # 大小写互换  .capitalize()    # 首字母大写，其余小写  .title()    # 首字母大写 |

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # 2格式化相关    .ljust(width)     # 获取固定长度，左对齐，右边不够用空格补齐  .rjust(width)     # 获取固定长度，右对齐，左边不够用空格补齐  .center(width)  # 获取固定长度，中间对齐，两边不够用空格补齐  .zfill(width)      # 获取固定长度，右对齐，左边不足用0补齐 |

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # 3 字符串搜索相关    .find()    # 搜索指定字符串，没有返回-1  .index()    # 同上，但是找不到会报错  .rfind()    # 从右边开始查找  .count()    # 统计指定的字符串出现的次数    # 上面所有方法都可以用index代替，不同的是使用index查找不到会抛异常，而find返回-1 |

s='hello world'

print(s.find('e')) # 搜索指定字符串,没有返回-1

print(s.find('w',1,2)) # 顾头不顾尾，找不到则返回-1不会报错，找到了则显示索引

print(s.index('w',1,2)) # 同上，但是找不到会报错

print(s.count('o')) # 统计指定的字符串出现的次数

print(s.rfind('l')) # 从右边开始查找

[复制代码](javascript:void(0);)

# 4字符串替换

.replace('old','new') # 替换old为new

.replace('old','new',次数) # 替换指定次数的old为new

s='hello world'

print(s.replace('world','python'))

print(s.replace('l','p',2))

print(s.replace('l','p',5))

执行结果：

hello python

heppo world

heppo worpd

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

# 5字符串去空格及去指定字符

.strip() # 去两边空格

.lstrip() # 去左边空格

.rstrip() # 去右边空格

.split() # 默认按空格分隔

.split('指定字符') # 按指定字符分割字符串为数组

s=' h e-l lo '

print(s)

print(s.strip())

print(s.lstrip())

print(s.rstrip())

print(s.split('-'))

print(s.split())

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

# 6字符串判断相关

.startswith('start') # 是否以start开头

.endswith('end') # 是否以end结尾

.isalnum() # 是否全为字母或数字

.isalpha() # 是否全字母

.isdigit() # 是否全数字

.islower() # 是否全小写

.isupper() # 是否全大写

.istitle() # 判断首字母是否为大写

.isspace() # 判断字符是否为空格  
  
# 补充

bin() # 十进制数转八进制  
hex() # 十进制数转十六进制  
range() # 函数：可以生成一个整数序列  
type() # 查看数据类型  
len() # 计算字符串长度  
format() # 格式化字符串，类似%s，传递值能多不能少

[复制代码](javascript:void(0);)

**3、python中str函数isdigit、isdecimal、isnumeric的区别**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | isdigit()  True: Unicode数字，byte数字（单字节），全角数字（双字节），罗马数字  False: 汉字数字  Error: 无    isdecimal()  True: Unicode数字，，全角数字（双字节）  False: 罗马数字，汉字数字  Error: byte数字（单字节）    isnumeric()  True: Unicode数字，全角数字（双字节），罗马数字，汉字数字  False: 无  Error: byte数字（单字节） |

**4、内置函数**

* 数学运算(7个)
* 类型转换(24个)
* 序列操作(8个)
* 对象操作(7个)
* 反射操作(8个)
* 变量操作(2个)
* 交互操作(2个)
* 文件操作(1个)
* 编译执行(4个)
* 装饰器(3个)

**数学运算**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | abs：求数值的绝对值  abs(-2)    divmod：返回两个数值的商和余数  divmod(5,2)  divmod(5.5,2)    max：返回迭代对象中的元素的最大值或者所有参数的最大值  max(1,2,3)    # 传入3个参数 取3个中较大者  max('1234')    # 传入1个可迭代对象，取其最大元素值  max(-1,0,key=abs)    # 传入了求绝对值函数，则参数都会进行求绝对值后再取较大者    min：返回可迭代对象中的元素的最小值或者所有参数的最小值  min(1,2,3)　　# 传入3个参数 取3个中较小者  min('1234')    # 传入1个可迭代对象，取其最小元素值  min(-1,-2,key=abs)    # 传入了求绝对值函数，则参数都会进行求绝对值后再取较小者    pow：返回两个数值的幂运算值或其余指定整数的模值  pow(2,3)    round：对浮点数进行四舍五入求值  round(1.1111,1)    sum：对元素类型是数值的可迭代对象中的每个元素求和  sum((1,2,3,4))    # 传入可迭代对象、元素类型必须是数值型 |

**类型转换**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91 | bool：根据传入的参数的逻辑值创建一个新的布尔值  bool()或bool(0)     # 数值0、空值为False    int：根据传入的参数创建一个新的整数  int()     # 不传入参数时，得到结果0    float：根据传入的参数创建一个新的浮点数  float()    # 不提供参数的时候，返回0.0    complex：根据传入参数创建一个新的复数  complex()    # 当两个参数都不提供时，返回复数 0j    str：返回一个对象的字符串表现形式(给用户)    bytearray：根据传入的参数创建一个新的字节数组  bytearray('中文','utf-8')  bytearray(b'\xe4\xb8\xad\xe6\x96\x87')    bytes：根据传入的参数创建一个新的不可变字节数组  bytes('中文','utf-8')  b'\xe4\xb8\xad\xe6\x96\x87'    memoryview：根据传入的参数创建一个新的内存查看对象  v=memoryview(b'asdf')  print(v[0])    # 97  print(v[-1])    # 102    ord：返回Unicode字符对应的整数  print(ord('a'))    chr：返回整数所对应的Unicode字符  print(chr(97))    bin：将整数转换成2进制字符串  oct：将整数转化成8进制数字符串  hex：将整数转换成16进制字符串    tuple：根据传入的参数创建一个新的元组  list：根据传入的参数创建一个新的列表  dict：根据传入的参数创建一个新的字典  set：根据传入的参数创建一个新的集合    frozenset：根据传入的参数创建一个新的不可变集合  a=frozenset(range(10))  print(a)  # frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})    enumerate：根据可迭代对象创建枚举对象  l1=['one','two','three','five']  print(list(enumerate(l1)))  # [(0, 'one'), (1, 'two'), (2, 'three'), (3, 'five')]  print(list(enumerate(l1,start=1)))  # 指定起始值  # [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'five')]    range：根据传入的参数创建一个新的range对象  iter：根据传入的参数创建一个新的可迭代对象  a=iter('asdf')  print(a)    # <str\_iterator object at 0x00000190B4D99668>  print(next(a))  # a  print(next(a))  # s  print(next(a))  # d  print(next(a))  # f  print(next(a))  # 报错StopIteration    slice：根据传入的参数创建一个新的切片对象  c1=slice(5)  print(c1)   # slice(None, 5, None)  c1=slice(2,5)  print(c1)   # slice(2, 5, None)  c1=slice(1,4,7)  print(c1)   # slice(1, 4, 7)    super：根据传入的参数创建一个新的子类和父类关系的代理对象  # 定义父类A类  class A(object):      def \_\_init\_\_(self):          print(A.\_\_init\_\_)    # 定义子类，继承A  class B(A):      def \_\_init\_\_(self):          print(B.\_\_init\_\_)          super().\_\_init\_\_()    # super调用父类方法  b=B()  print(b)  <function B.\_\_init\_\_ at 0x0000023DB0CA76A8>  <function A.\_\_init\_\_ at 0x0000023DB0CA7620>    object：创建一个新的object对象 |
| 1 | <span style="font-size: 14pt"><strong>序列操作</strong></span> |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | all：判断可迭代对象的每个元素是否都为True值  print(all([1,2]))    # 列表中每个元素逻辑值均为True，返回True  print(all([0,2]))     # 列表中0的逻辑值为False，返回False    any：判断可迭代对象的元素是否有为True值的元素  # 列表元素有一个为True，则返回True  # 列表元素全部为False，则返回False    filter：使用指定方法过滤可迭代对象的元素    map：使用指定方法去作用传入的每个可迭代对象的元素，生成新的可迭代对象    next：返回可迭代对象中的下一个元素值  # 传入default参数后，如果可迭代对象还有元素没有返回，则依次返回其元素值，如果所有元素已经返回，则返回default指定的默认值而不抛出StopIteration 异常    reversed：反转序列生成新的可迭代对象    sorted：对可迭代对象进行排序，返回一个新的列表    zip：聚合传入的每个迭代器中相同位置的元素，返回一个新的元组类型迭代器 |

**对象操作**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | help：返回对象的帮助信息  dir：返回对象或者当前作用域内的属性列表  id：返回对象的唯一标识符  hash：获取对象的哈希值  type：返回对象的类型，或者根据传入的参数创建一个新的类型  len：返回对象的长度  ascii：返回对象的可打印表字符串表现方式  format：格式化显示值    vars：返回当前作用域内的局部变量和其值组成的字典，或者返回对象的属性列表  class A(object):      pass    a=A()  print(a.\_\_dict\_\_)   # {}  print(vars(a))      # {}  a.name='buer'  print(a.\_\_dict\_\_)   # {'name': 'buer'}  print(vars(a))      # {'name': 'buer'} |

**反射操作**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | \_\_import\_\_：动态导入模块  print(\_\_import\_\_('os'))  print(\_\_import\_\_('time'))    # <module 'os' from 'D:\\Python36\\lib\\os.py'>  # <module 'time' (built-in)>    isinstance：判断对象是否是类或者类型元组中任意类元素的实例  issubclass：判断类是否是另外一个类或者类型元组中任意类元素的子类    hasattr：检查对象是否含有属性  class Student:      def \_\_init\_\_(self,name):          self.name=name    s=Student('Ethan')  print(hasattr(s,'name'))    # 含有name属性为True  print(hasattr(s,'age'))     # 不含有age属性为False    getattr：获取对象的属性值  print(getattr(s,'name'))    # 存在属性name，Ethan  print(getattr(s,'age',20))  # 不存在属性age，但提供了默认值，返回默认值  print(getattr(s,'age'))     # 不存在属性age，未提供默认值，调用报错  报错如下：  Traceback (most recent call last):    File "D:/test.py", line 30, in <module>      print(getattr(s,'age'))  AttributeError: 'Student' object has no attribute 'age'    setattr：设置对象的属性值  print(s.name)   # Ethan  setattr(s,'name','Tom')   # name属性存在，做赋值操作  setattr(s,'age',18)     # age属性不存在，创建这个属性  print(s.name)   # Tom  print(s.age)    # 18    delattr：删除对象的属性  class Student:      def \_\_init\_\_(self,name):          self.name=name      def foo(self):          print('hello %s' % self.name)    a=Student('Ethan')    print(a.name)   # Ethan  print(a.foo())  # hello Ethan    print(delattr(a,'name'))    # name属性被删除  print(a.name)   # 调用报错  Traceback (most recent call last):    File "D:/test.py", line 50, in <module>      print(a.name)   # 调用报错  AttributeError: 'Student' object has no attribute 'name'    callable：检测对象是否可被调用  class B:      def \_\_call\_\_(self, \*args, \*\*kwargs):          print('instances are callable now')    print(callable(B))  # 类B是可调用对象  b=B()   # 调用类B  print(callable(b))  # 实例b是可调用对象  print(b())  # 调用实例b成功  # instances are callable now |

**变量操作**

|  |  |
| --- | --- |
| 1  2 | globals：返回当前作用域内的全局变量和其值组成的字典  locals：返回当前作用域内的局部变量和其值组成的字典 |

**交互操作**

|  |  |
| --- | --- |
| 1  2  3 | print：向标准输出对象打印输出  input：读取用户输入值  user=input('please input your name:') |

**文件操作**

ascii只支持英文，所以各国自己搞了编码，如我们的GB2312。为了统一支持各国编码，出现了unicode：2个字节一个字符，2\*\*16-1=65535，基本网罗全世界的字符。看似是一统江湖了，但是让英文也俩字符有点浪费。于是产生了UTF-8，对英文字符只用1Bytes表示，对中文字符用3Bytes，所以utf-8被称作可变长度编码。你想，咱们写的代码，除了输入输出和注释之外，还真没几个汉字，保存成utf-8太省空间了。很多网站也用utf-8编码，也是为了传输起来快。你可能会问，内存里为啥用unicode呢？因为它转二进制快。你输入了个字符串，放到内存时unicode直接换成俩字节，要是用utf-8还得判断一下子该用几个字节，就慢下来了。

记事本，默认编码ansi(GBK)，那如果是读取数据文件到程序中呢？其实倒也不用非得utf-8，默认ansi（GBK）就能正常读取，如下：  
with open("test.txt","r") as f:

                print(f.read())

如果test.txt是utf-8编码的，反而会报错 ：'gbk' codec can't decode……。

我们可以在读过来的时候就指定编码，open("test.txt","r",encoding="utf-8") ，这样就在读入内存时就准确地按utf-8规则转成unicode，print函数在命令提示符下自动完成了转换输出。当然，你也可以用"rb"模式读取二进制，然后print时，decode('utf-8')成unicode，效果和第一种一样。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | open：使用指定的模式和编码打开文件，返回文件读写对象  # 写入文件  a= open('a.text','w')  a.write('124sdgadgahg ggadh')    # 读取文件  a= open('a.text','rt')  print(a.read())  a.close() |

**编译执行**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | compile：将字符串编译为代码或者AST对象，使之能够通过exec语句来执行或者eval进行求值  # 流程语句使用exec  code1='for i in range(5):print(i)'  compile1=compile(code1,'','exec')  exec (compile1)  # 0  # 1  # 2  # 3  # 4    # 简单求值表达式用eval  code2='1+2+3+4'  compile2=compile(code2,'','eval')  print(eval(compile2))   # 10    eval：执行动态表达式求值  print(eval('1+2+3+4'))  # 10  print(eval('2\*2\*2'))    # 8  print(eval('10/2+2\*2')) # 9.0    exec：执行动态语句块  exec ('a=1+2')  print(a)    # 3  exec ('b=4\*3/2-1')  print(b)    # 5.0    repr：返回一个对象的字符串表现形式(给解释器)  a='hello world'  print(str(a))   # hello world  print(repr(a))  # 'hello world' |

**装饰器**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | property：标示属性的装饰器  class A:      def \_\_init\_\_(self):          pass      @property      def foo(self):          print('1111111111')  a=A()  print(a.foo)    # 访问属性，不需要加()执行foo    classmethod：标示方法为类方法的装饰器  class B(object):      def \_\_init\_\_(self):          pass        @classmethod      def foo(cls):          print(cls)    print(B.foo())  # 类对象调用类方法  # <class '\_\_main\_\_.B'>  b=B()  print(b.foo())  # 类实例对象调用类方法  # <class '\_\_main\_\_.B'>    staticmethod：标示方法为静态方法的装饰器  class C(object):      def \_\_init\_\_(self):          pass      @staticmethod      def f1():          print('hahahha')    print(C.f1())   # 类调用  c=C()  print(c.f1())   # 类实例对象调用 |

　补充：

[复制代码](javascript:void(0);)

"""

python内置装饰器

在python中有三个内置的装饰器，都是跟class相关的：staticmethod、classmethod、property.

@staticmethod 是类的静态方法，其跟成员方法的区别是没有self参数，并且可以在类不进行实例化的情况下调用

@classmethod 与成员方法的区别在于所接收的第一个参数不是self（类实例的指针），而是cls（当前类的具体类型）

@property 是属性的意思，表示可以通过类实例直接访问的信息

"""

class Foo(object):

def \_\_init\_\_(self,var):

super(Foo,self).\_\_init\_\_()

self.\_var=var

@property

def var(self):

return self.\_var

@var.setter

def var(self,var):

self.\_var=var

f=Foo('var1')

print(f.var)

f.var='var2'

print(f.var)

"""

注意，对于Python新式类（new-style class），如果将上面的 “@var.setter” 装饰器所装饰的成员函数去掉，

则Foo.var 属性为只读属性，使用 “foo.var = ‘var 2′” 进行赋值时会抛出异常。

但是，对于Python classic class，所声明的属性不是 read-only的，所以即使去掉”@var.setter”装饰器也不会报错。

"""

[复制代码](javascript:void(0);)

**对字符串切片**

字符串 'xxx'和 Unicode字符串 u'xxx'也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：str**[x:y]表示，从索引x开始取，直到索引y为止，但不包括索引y。**切片是指对操作的对象截取其中一部分的操作。字符串、列表、元组都支持切片操作。

切片的语法：[起始:结束:步长]

注意：选取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)。

下标索引

所谓“下标”，就是编号，就好比超市中的存储柜的编号，通过这个编号就能找到相应的存储空间。

如果想取出部分字符，那么可以通过下标的方法，（注意python中下标从 0 开始）

>>> 'ABCDEFG'[:3]

'ABC'

>>> 'ABCDEFG'[-3:]

'EFG'

>>> 'ABCDEFG'[::2]

'ACEG'

在很多编程语言中，针对字符串提供了很多各种截取函数，其实目的就是对字符串切片。知识点：Python没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

**编程任务**

字符串有个方法 upper() 可以把字符变成大写字母：

>>> 'abc'.upper()

'ABC'

但它会把所有字母都变成大写。请设计一个函数，它接受一个字符串，然后返回一个仅首字母变成大写的字符串。

提示：利用切片操作简化字符串操作。

实现代码:

def firstCharUpper(s):

return s[0].upper() + s[1:]

print firstCharUpper('hello')

print firstCharUpper('sunday')

print firstCharUpper('september')

运行结果：

Hello

Sunday

September

**各种迭代方式**

**什么是迭代**

在Python中，如果给定一个list或tuple，我们可以通过for循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。

在Python中，迭代是通过 for ... in 来完成的，而很多语言比如C或者Java，迭代list是通过下标完成的，比如Java代码：

for (i=0; i<list.length; i++) {

n = list[i];

}

可以看出，Python的for循环抽象程度要高于Java的for循环。

因为 Python 的 for循环不仅可以用在list或tuple上，还可以作用在其他任何可迭代对象上。

因此，迭代操作就是对于一个集合，**无论该集合是有序还是无序**，我们用 for 循环总是可以依次取出集合的每一个元素。

注意: **集合**是指包含一组元素的数据结构，我们已经介绍的包括：

1. 有序集合：list，tuple，知识点: str和**unicode**；
2. 无序集合：set
3. 无序集合并且具有 key-value 对：dict

而迭代是一个动词，它指的是一种操作，在Python中，就是 for 循环。

迭代与按下标访问数组最大的不同是，后者是一种具体的迭代实现方式，而前者只关心迭代结果，根本不关心迭代内部是如何实现的。

**编程任务**

请用for循环迭代数列 1-100 并打印出7的倍数。

实现代码:

for i in range(1, 101):

if i % 7 == 0:

print i

运行结果:

7

14

21

28

35

42

49

56

63

70

77

84

91

98

**索引迭代**

知识点：Python中，迭代永远是取出元素本身，而非元素的索引。

对于有序集合，元素确实是有索引的。有的时候，我们确实想在 for 循环中拿到索引，怎么办？

方法是使用 enumerate()函数：

>>> L = ['Adam', 'Lisa', 'Bart', 'Paul']

>>> for index, name in enumerate(L):

... print index, '-', name

...

0 - Adam

1 - Lisa

2 - Bart

3 - Paul

使用 enumerate()函数，我们可以在for循环中同时绑定索引index和元素name。但是，这不是 enumerate() 的特殊语法。实际上，enumerate() 函数把：

['Adam', 'Lisa', 'Bart', 'Paul']

变成了类似：

[(0, 'Adam'), (1, 'Lisa'), (2, 'Bart'), (3, 'Paul')]

因此，迭代的每一个元素实际上是一个tuple：

for t in enumerate(L):

index = t[0]

name = t[1]

print index, '-', name

如果我们知道每个tuple元素都包含两个元素，for循环又可以进一步简写为：

for index, name in enumerate(L):

print index, '-', name

这样不但代码更简单，而且还少了两条赋值语句。

可见，知识点: **索引迭代也不是真的按索引访问，而是由 enumerate() 函数自动把每个元素变成 (index, element) 这样的tuple，再迭代，就同时获得了索引和元素本身。**

**编程任务(天涯)**

zip()函数可以把两个 list 变成一个 list：

>>> zip([10, 20, 30], ['A', 'B', 'C'])

[(10, 'A'), (20, 'B'), (30, 'C')]

在迭代 ['Adam', 'Lisa', 'Bart', 'Paul']时，如果我们想打印出名次 - 名字（名次从1开始)，请考虑如何在迭代中打印出来。

提示：考虑使用zip()函数和range()函数

实现代码:

L = ['Adam', 'Lisa', 'Bart', 'Paul']

for index, name in zip(range(1, len(L)+1), L):

print index, '-', name

运行结果:

1 - Adam

2 - Lisa

3 - Bart

4 - Paul

**迭代dict的value**

迭代dict的value  
我们已经了解了dict对象本身就是可迭代对象，用 **for 循环直接迭代 dict，可以每次拿到dict的一个key。**

如果我们希望迭代 dict 对象的value，应该怎么做？

知识点：values()把dict转换成一个包含所有value的list  
dict 对象有一个 values() 方法，这个方法把dict转换成一个包含所有value的list，这样，我们迭代的就是 dict的每一个 value：

d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }

print d.values()

# [85, 95, 59]

for v in d.values():

print v

# 85

# 95

# 59

如果仔细阅读Python的文档，还可以发现，dict除了values()方法外，还有一个 itervalues() 方法，**用 itervalues() 方法替代 values() 方法，迭代效果完全一样：**

d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }

print d.itervalues()

# <dictionary-valueiterator object at 0x106adbb50>

for v in d.itervalues():

print v

# 85

# 95

# 59

那这两个方法有何不同之处呢？

1. values() 方法实际上把一个 dict 转换成了包含 value 的list。
2. 但是 itervalues() 方法**不会转换**，它会在迭代过程中依次从 dict 中取出 value，所以 itervalues() 方法比 values() 方法**节省了生成 list 所需的内存。**
3. 打印 itervalues() 发现它返回一个 <dictionary-valueiterator> 对象，这说明在Python中，for 循环可作用的迭代对象远不止 list，tuple，str，unicode，dict等，知识点: **任何可迭代对象都可以作用于for循环，**而内部如何迭代我们通常并不用关心。

如果一个对象说自己可迭代，那我们就直接用 for 循环去迭代它，知识点: 可见，**迭代是一种抽象的数据操作，它不对迭代对象内部的数据有任何要求。**

**编程任务**

给定一个dict：d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59, 'Paul': 74 }

请计算所有同学的平均分。

实现代码:

d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59, 'Paul': 74 }

sum = 0.0

for v in d.itervalues():

sum = sum + v

print sum / len(d)

运行结果:

78.25

**迭代dict的key和value**

我们了解了如何迭代 dict 的key和value，那么，在一个 for 循环中，**能否同时迭代 key和value？**答案是肯定的。

首先，我们看看 dict 对象的 items()方法返回的值：

>>> d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }

>>> print d.items()

[('Lisa', 85), ('Adam', 95), ('Bart', 59)]

可以看到，items() 方法把dict对象转换成了包含tuple的list，我们对这个list进行迭代，可以同时获得key和value：

>>> for key, value in d.items():

... print key, ':', value

...

Lisa : 85

Adam : 95

Bart : 59

和 values()有一个 itervalues() 类似，items() 也有一个对应的 iteritems()，知识点： **iteritems() 不把dict转换成list，而是在迭代过程中不断给出 tuple，所以， iteritems() 不占用额外的内存。**

**编程任务**

请根据dict：d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59, 'Paul': 74 }

打印出 name : score，最后再打印出平均分 average : score。

实现代码：

d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59, 'Paul': 74 }

sum = 0.0

for k, v in d.iteritems():

sum = sum + v

print k, ':', v

print 'average', ':', sum / len(d)

运行结果：

Lisa : 85

Paul : 74

Adam : 95

Bart : 59

average : 78.25

**列表生成式:快速生成列表**

生成列表  
要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]，我们可以用range(1, 11)：

>>> range(1, 11)

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

但如果要生成[1x1, 2x2, 3x3, ..., 10x10]怎么做？

* 方法一是循环：

>>> L = []

>>> for x in range(1, 11):

... L.append(x \* x)

...

>>> L

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

但是循环太繁琐，而**列表生成式**则可以用一行语句代替循环生成上面的list：

>>> [x \* x for x in range(1, 11)]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

这种写法就是Python特有的列表生成式。利用列表生成式，可以以非常简洁的代码生成 list。

知识点: 写列表生成式时，把要生成的元素 x \* x放到前面，后面跟 for 循环，就可以把list创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

**编程任务**

请利用列表生成式生成列表 [1x2, 3x4, 5x6, 7x8, ..., 99x100]

提示：range(1, 100, 2)可以生成list [1, 3, 5, 7, 9,...]

实现代码：

print [x \* (x + 1) for x in range(1, 100, 2)]

运行结果：

[2, 12, 30, 56, 90, 132, 182, 240, 306, 380, 462, 552, 650, 756, 870, 992, 1122, 1260, 1406, 1560, 1722, 1892, 2070, 2256, 2450, 2652, 2862, 3080, 3306, 3540, 3782, 4032, 4290, 4556, 4830, 5112, 5402, 5700, 6006, 6320, 6642, 6972, 7310, 7656, 8010, 8372, 8742, 9120, 9506, 9900]

**复杂表达式**

使用for循环的迭代不仅可以迭代普通的list，还可以迭代dict。

假设有如下的dict：

d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }

完全可以通过一个复杂的列表生成式把它变成一个 HTML 表格：

tds = ['<tr><td>%s</td><td>%s</td></tr>' % (name, score) for name, score in d.iteritems()]

print '<table>'

print '<tr><th>Name</th><th>Score</th><tr>'

print '\n'.join(tds)

print '</table>'

个人：<tr><td>%s</td><td>%s</td></tr> 中：

* 第一个%s是name的填充位置。
* 第二个%s为score的填充位置。

有多少个name和score，会通过循环生成多少个。  
<tr><th>Name</th><th>Score</th><tr>设置表格头  
print '\n'.join(tds)。列表里的项通过\n连接成字符串。

注：字符串可以通过%进行格式化，用指定的参数替代 %s。字符串的join()方法可以把一个 list拼接成一个字符串。

把打印出来的结果保存为一个html文件，就可以在浏览器中看到效果了：

<table border="1">

<tr><th>Name</th><th>Score</th><tr>

<tr><td>Lisa</td><td>85</td></tr>

<tr><td>Adam</td><td>95</td></tr>

<tr><td>Bart</td><td>59</td></tr>

</table>

**编程任务(天涯)**

在生成的表格中，对于没有及格的同学，请把分数标记为红色。

提示：红色可以用 <td style="color:red"> 实现。

实现代码:

d = { 'Adam': 95, 'Lisa': 85, 'Bart': 59 }

def generate\_tr(name, score):

if score < 60:

return '<tr><td>%s</td><td style="color:red">%s</td></tr>' % (name, score)

return '<tr><td>%s</td><td>%s</td></tr>' % (name, score)

tds = [generate\_tr(name, score) for name, score in d.iteritems()]

print '<table border="1">'

print '<tr><th>Name</th><th>Score</th><tr>'

print '\n'.join(tds)

print '</table>'

运行结果:

[图片上传失败...(image-837d01-1514981535184)]

**条件过滤**

列表生成式的 for 循环后面还可以加上 if 判断。例如：

>>> [x \* x for x in range(1, 11)]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

如果我们只想要偶数的平方，不改动 range()的情况下，可以加上 if 来筛选：

>>> [x \* x for x in range(1, 11) if x % 2 == 0]

[4, 16, 36, 64, 100]

有了 if 条件，只有 if 判断为 True 的时候，才把循环的当前元素添加到列表中。

**编程任务**

请编写一个函数，它接受一个 list，然后把list中的所有字符串变成大写后返回，非字符串元素将被忽略。

提示：

1. isinstance(x, str) 可以判断变量 x 是否是字符串；
2. 字符串的 upper() 方法可以返回大写的字母。

实现代码:

def toUppers(L):

return [x.upper() for x in L if isinstance(x, str)]

print toUppers(['Hello', 'world', 101])

运行结果:

['HELLO', 'WORLD']

**多层表达式(知识点)**

for循环可以嵌套，知识点：因此，在列表生成式中，也可以用多层 for 循环来生成列表。

对于字符串 'ABC' 和 '123'，可以使用两层循环，生成全排列：

>>> [m + n for m in 'ABC' for n in '123']

['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']

翻译成循环代码就像下面这样：

L = []

for m in 'ABC':

for n in '123':

L.append(m + n)

**编程任务(天涯)**

利用 3 层for循环的列表生成式，找出对称的 3 位数。例如，121 就是对称数，因为从右到左倒过来还是 121。

实现代码:

print [100 \* n1 + 10 \* n2 + n3 for n1 in range(1, 10) for n2 in range(10) for n3 in range(10) if n1==n3]

运行结果：

101, 111, 121, 131, 141, 151, 161, 171, 181, 191, 202, 212, 222, 232, 242, 252, 262, 272, 282, 292, 303, 313, 323, 333, 343, 353, 363, 373, 383, 393, 404, 414, 424, 434, 444, 454, 464, 474, 484, 494, 505, 515, 525, 535, 545, 555, 565, 575, 585, 595, 606, 616, 626, 636, 646, 656, 666, 676, 686, 696, 707, 717, 727, 737, 747, 757, 767, 777, 787, 797, 808, 818, 828, 838, 848, 858, 868, 878, 888, 898, 909, 919, 929, 939, 949, 959, 969, 979, 989, 999]

# python 格式化输出%的用法

1. 整数的输出 %d —— dec 十进制

‘%3d’%(10)---- 空10

2、浮点数输出 %f ——保留小数点后面六位有效数字

　　 %.3f保留3位小数位

3、字符串输出

%s

%10s——右对齐，占位符10位

%-10s——左对齐，占位符10位

%.2s——截取2位字符串

%10.2s——10位占位符，截取两位字符串

>>> print('%s' % 'hello world') # 字符串输出

hello world

>>> print('%20s' % 'hello world') # 右对齐，取20位，不够则补位

hello world

>>> print('%-20s' % 'hello world') # 左对齐，取20位，不够则补位

hello world

>>> print('%.2s' % 'hello world') # 取2位

he

>>> print('%10.2s' % 'hello world') # 右对齐，取2位

he

>>> print('%-10.2s' % 'hello world') # 左对齐，取2位

he

个人学习笔记，内容来自慕课网课程。请勿用作商业用途。

作者：天涯明月笙  
链接：https://www.jianshu.com/p/1297ebda1f65  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

|  |  |
| --- | --- |
| 表 1 time模块常用函数 | |
| **time常用函数** | **功能描述** |
| time.asctime([t]) | 将时间元组或 struct\_time 转换为时间字符串。如果不指定参数 t，则默认转换当前时间。 |
| time.ctime([secs]) | 将以秒数代表的时间（格林威治时间）转换为时间字符串。 |
| time.gmtime([secs]) | 将以秒数代表的时间转换为 struct\_time 对象。如果不传入参数，则使用当前时间。 |
| time.localtime([secs]) | 将以秒数代表的时间转换为代表当前时间的 struct\_time 对象。如果不传入参数，则使用当前时间。 |
| time.mktime(t) | 它是 localtime 的反转函数，用于将 struct\_time 对象或元组代表的时间转换为从 1970 年 1 月 1 日 0 点整到现在过了多少秒。 |
| time.perf\_counter() | 返回性能计数器的值。以秒为单位。 |
| time.process\_time() | 返回当前进程使用 CPU 的时间，以秒为单位。 |
| time.sleep(secs) | 暂停 secs 秒，什么都不干。 |
| time.strftime(format[, t]) | 将时间元组或 struct\_time 对象格式化为指定格式的时间字符串。如果不指定参数 t，则默认转换当前时间。 |
| time.strptime(string[, format]) | 将字符串格式的时间解析成 struct\_time 对象。 |
| time.time() | 返回从 1970 年 1 月 1 日 0 点整到现在过了多少秒。 |
| time.timezone | 返回本地时区的时间偏移，以秒为单位。 |
| time.tzname | 返回本地时区的名字。 |

Python 可以用从 1970 年 1 月 1 日 0 点整到现在所经过的秒数来代表当前时间（又称格林威治时间），比如我们写 30 秒，那么意味着时间是 1970 年 1 月 1 日 0 点 0 分 30 秒。但需要注意的是，在实际输出时可能会受到时区的影响，比如中国处于东八区，因此实际上会输出 1970 年 1 月 1 日 8 点 0 分 30 秒。

下面程序示范了 time 棋块的功能函数：

1. import time
2. # 将当前时间转换为时间字符串
3. **print**(time.asctime())
4. # 将指定时间转换时间字符串，时间元组的后面3个元素没有设置
5. **print**(time.asctime((2018, 2, 4, 11, 8, 23, 0, 0 ,0))) # Mon Feb 4 11:08:23 2018
6. # 将以秒数为代表的时间转换为时间字符串
7. **print**(time.ctime(30)) # Thu Jan 1 08:00:30 1970
8. # 将以秒数为代表的时间转换为struct\_time对象。
9. **print**(time.gmtime(30))
10. # 将当前时间转换为struct\_time对象。
11. **print**(time.gmtime())
12. # 将以秒数为代表的时间转换为代表当前时间的struct\_time对象
13. **print**(time.localtime(30))
14. # 将元组格式的时间转换为秒数代表的时间
15. **print**(time.mktime((2018, 2, 4, 11, 8, 23, 0, 0 ,0))) # 1517713703.0
16. # 返回性能计数器的值
17. **print**(time.perf\_counter())
18. # 返回当前进程使用CPU的时间
19. **print**(time.process\_time())
20. #time.sleep(10)
21. # 将当前时间转换为指定格式的字符串
22. **print**(time.strftime('%Y-%m-%d %H:%M:%S'))
23. st = '2018年3月20日'
24. # 将指定时间字符串恢复成struct\_time对象。
25. **print**(time.strptime(st, '%Y年%m月%d日'))
26. # 返回从1970年1970年1月1日0点整到现在过了多少秒。
27. **print**(time.time())
28. # 返回本地时区的时间偏移，以秒为单位
29. **print**(time.timezone) # 在国内东八区输出-28800

运行上面程序，可以看到如下输出结果：

Fri Feb 22 11:28:39 2019  
Mon Feb  4 11:08:23 2018  
Thu Jan  1 08:00:30 1970  
time.struct\_time(tm\_year=1970, tm\_mon=1, tm\_mday=1, tm\_hour=0, tm\_min=0, tm\_sec=30, tm\_wday=3, tm\_yday=1, tm\_isdst=0)  
time.struct\_time(tm\_year=2019, tm\_mon=2, tm\_mday=22, tm\_hour=3, tm\_min=28, tm\_sec=39, tm\_wday=4, tm\_yday=53, tm\_isdst=0)  
time.struct\_time(tm\_year=1970, tm\_mon=1, tm\_mday=1, tm\_hour=8, tm\_min=0, tm\_sec=30, tm\_wday=3, tm\_yday=1, tm\_isdst=0)  
1517713703.0  
0.0  
0.140625  
2019-02-22 11:28:39  
time.struct\_time(tm\_year=2018, tm\_mon=3, tm\_mday=20, tm\_hour=0, tm\_min=0, tm\_sec=0, tm\_wday=1, tm\_yday=79, tm\_isdst=-1)  
1550806119.4960592  
-28800

time 模块中的 strftime() 和 strptime() 两个函数互为逆函数，其中 strftime() 用于将 struct\_time 对象或时间元组转换为时间字符串；而 strptime() 函数用于将时间字符串转换为 struct\_time 对象。这两个函数都涉及编写格式模板，比如上面程序中使用 %Y 代表年、%m 代表月、%d 代表日、%H 代表时、%M 代表分、%S 代表秒。这两个函数所需要的时间格式字符串支持的指令如表 2 所示：

|  |  |
| --- | --- |
| 表 2 Python 时间格式字符串所支持的指令 | |
| **指令** | **含义** |
| %a | 本地化的星期几的缩写名，比如 Sun 代表星期天 |
| %A | 本地化的星期几的完整名 |
| %b | 本地化的月份的缩写名，比如 Jan 代表一月 |
| %B | 本地化的月份的完整名 |
| %c | 本地化的日期和时间的表示形式 |
| %d | 代表一个月中第几天的数值，范固： 01~31 |
| %H | 代表 24 小时制的小时，范围：00~23 |
| %I | 代表 12 小时制的小时，范围：01~12 |
| %j | 一年中第几天，范围：001~366 |
| %m | 代表月份的数值，范围：01~12 |
| %M | 代表分钟的数值，范围：00~59 |
| %p | 上午或下午的本地化方式。当使用 strptime() 函数并使用 %I 指令解析小时时，%p 只影响小时字段 |
| %S | 代表分钟的数值，范围：00~61。该范围确实是 00~61，60 在表示闰秒的时间戳时有效，而 61 则是由于一些历史原因造成的 |
| %U | 代表一年中表示第几周，以星期天为每周的第一天，范围：00~53。在这种方式下，一年中第一个星期天被认为处于第一周。当使用 strptime() 函数解析时间字符串时，只有同时指定了星期几和年份该指令才会有效 |
| %w | 代表星期几的数值，范围：0~6，其中 0 代表周日 |
| %W | 代表一年小第几周，以星期一为每周的第一天，范围：00~53。在这种方式下，一年中第一个星期一被认为处于第一周。当使用 strptime() 函数解析时间字符串时，只有同时指定了星期几和年份该指令才会有效 |
| %x | 本地化的日期的表示形式 |
| %X | 本地化的时间的表示形式 |
| %y | 年份的缩写，范围：00~99，比如 2018 年就简写成 18 |
| %Y | 年份的完整形式。如 2018 |
| %z | 显示时区偏移 |
| %Z | 时区名（如果时区不行在，则显示为空） |
| %% | 用于代表%符号 |

< [Python random](http://c.biancheng.net/view/2411.html)[Python json](http://c.biancheng.net/view/2423.html) >

当前路径目录

import sys

>>> sys.argv

['F:\\360MoveData\\Users\\Administrator\\Desktop\\1-3-1.py']

python操作excel主要用到xlrd和xlwt这两个库，即xlrd是读excel，xlwt是写excel的库。

常用单元格中的数据类型

1. empty（空的）,1 string（text）, 2 number, 3 date, 4 boolean, 5 error， 6 blank（空白表格）

  1）获取book中一个工作表

table = data.sheets()[0] #通过索引顺序获取

table = data.sheet\_by\_index(sheet\_indx)) #通过索引顺序获取

table = data.sheet\_by\_name(sheet\_name)#通过名称获取

以上三个函数都会返回一个xlrd.sheet.Sheet()对象

names = data.sheet\_names() #返回book中所有工作表的名字

data.sheet\_loaded(sheet\_name or indx) # 检查某个sheet是否导入完毕

2）行的操作

nrows = table.nrows #获取该sheet中的有效行数

table.row(rowx) #返回由该行中所有的单元格对象组成的列表

table.row\_slice(rowx) #返回由该列中所有的单元格对象组成的列表

table.row\_types(rowx, start\_colx=0, end\_colx=None) #返回由该行中所有单元格的数据类型组成的列表

table.row\_values(rowx, start\_colx=0, end\_colx=None) #返回由该行中所有单元格的数据组成的列表

table.row\_len(rowx) #返回该列的有效单元格长度

3）列(colnum)的操作

ncols = table.ncols #获取列表的有效列数

table.col(colx, start\_rowx=0, end\_rowx=None) #返回由该列中所有的单元格对象组成的列表

table.col\_slice(colx, start\_rowx=0, end\_rowx=None) #返回由该列中所有的单元格对象组成的列表

table.col\_types(colx, start\_rowx=0, end\_rowx=None) #返回由该列中所有单元格的数据类型组成的列表

table.col\_values(colx, start\_rowx=0, end\_rowx=None) #返回由该列中所有单元格的数据组成的列表

   4）单元格的操作

table.cell(rowx,colx) #返回单元格对象

table.cell\_type(rowx,colx) #返回单元格中的数据类型

table.cell\_value(rowx,colx) #返回单元格中的数据

table.cell\_xf\_index(rowx, colx) # 暂时还没有搞懂

 Turtle库是Python语言中一个很流行的绘制图像的函数库，想象一个小乌龟，在一个横轴为x、纵轴为y的坐标系原点，(0,0)位置开始，它根据一组函数指令的控制，在这个平面坐标系中移动，从而在它爬行的路径上绘制了图形。

# turtle绘图的基础知识：

## 1. 画布(canvas)

        画布就是turtle为我们展开用于绘图区域，我们可以设置它的大小和初始位置。

        设置画布大小

         turtle.screensize(canvwidth=None, canvheight=None, bg=None)，参数分别为画布的宽(单位像素), 高, 背景颜色。

        如：turtle.screensize(800,600, "green")

               turtle.screensize() #返回默认大小(400, 300)

        turtle.setup(width=0.5, height=0.75, startx=None, starty=None)，参数：width, height: 输入宽和高为整数时, 表示像素; 为小数时, 表示占据电脑屏幕的比例，(startx, starty): 这一坐标表示矩形窗口左上角顶点的位置, 如果为空,则窗口位于屏幕中心。

        如：turtle.setup(width=0.6,height=0.6)

               turtle.setup(width=800,height=800, startx=100, starty=100)

## 2. 画笔

**2.1 画笔的状态**

        在画布上，默认有一个坐标原点为画布中心的坐标轴，坐标原点上有一只面朝x轴正方向小乌龟。这里我们描述小乌龟时使用了两个词语：坐标原点(位置)，面朝x轴正方向(方向)， turtle绘图中，就是使用位置方向描述小乌龟(画笔)的状态。

**2.2 画笔的属性**

        画笔(画笔的属性，颜色、画线的宽度等)

        1) turtle.pensize()：设置画笔的宽度；

        2) turtle.pencolor()：没有参数传入，返回当前画笔颜色，传入参数设置画笔颜色，可以是字符串如"green", "red",也可以是RGB 3元组。

        3) turtle.speed(speed)：设置画笔移动速度，画笔绘制的速度范围[0,10]整数，数字越大越快。

**2.3 绘图命令**

         操纵海龟绘图有着许多的命令，这些命令可以划分为3种：一种为运动命令，一种为画笔控制命令，还有一种是全局控制命令。

(1)    画笔运动命令

|  |  |
| --- | --- |
| 命令 | 说明 |
| turtle.forward(distance) | 向当前画笔方向移动distance像素长度 |
| turtle.backward(distance) | 向当前画笔相反方向移动distance像素长度 |
| turtle.right(degree) | 顺时针移动degree° |
| turtle.left(degree) | 逆时针移动degree° |
| turtle.pendown() | 移动时绘制图形，缺省时也为绘制 |
| turtle.goto(x,y) | 将画笔移动到坐标为x,y的位置 |
| turtle.penup() | 提起笔移动，不绘制图形，用于另起一个地方绘制 |
| turtle.circle() | 画圆，半径为正(负)，表示圆心在画笔的左边(右边)画圆 |
| setx( ) | 将当前x轴移动到指定位置 |
| sety( ) | 将当前y轴移动到指定位置 |
| setheading(angle) | 设置当前朝向为angle角度 |
| home() | 设置当前画笔位置为原点，朝向东。 |
| dot(r) | 绘制一个指定直径和颜色的圆点 |

(2)     画笔控制命令

|  |  |
| --- | --- |
| 命令 | 说明 |
| turtle.fillcolor(colorstring) | 绘制图形的填充颜色 |
| turtle.color(color1, color2) | 同时设置pencolor=color1, fillcolor=color2 |
| turtle.filling() | 返回当前是否在填充状态 |
| turtle.begin\_fill() | 准备开始填充图形 |
| turtle.end\_fill() | 填充完成 |
| turtle.hideturtle() | 隐藏画笔的turtle形状 |
| turtle.showturtle() | 显示画笔的turtle形状 |

(3)    全局控制命令

|  |  |
| --- | --- |
| 命令 | 说明 |
| turtle.clear() | 清空turtle窗口，但是turtle的位置和状态不会改变 |
| turtle.reset() | 清空窗口，重置turtle状态为起始状态 |
| turtle.undo() | 撤销上一个turtle动作 |
| turtle.isvisible() | 返回当前turtle是否可见 |
| stamp() | 复制当前图形 |
| turtle.write(s [,font=("font-name",font\_size,"font\_type")]) | 写文本，s为文本内容，font是字体的参数，分别为字体名称，大小和类型；font为可选项，font参数也是可选项 |

(4)    其他命令

|  |  |
| --- | --- |
| 命令 | 说明 |
| turtle.mainloop()或turtle.done() | 启动事件循环 -调用Tkinter的mainloop函数。  必须是乌龟图形程序中的最后一个语句。 |
| turtle.mode(mode=None) | 设置乌龟模式（“standard”，“logo”或“world”）并执行重置。如果没有给出模式，则返回当前模式。   |  |  |  | | --- | --- | --- | | 模式 | 初始龟标题 | 正角度 | | standard | 向右（东） | 逆时针 | | logo | 向上（北） | 顺时针 | |
| turtle.delay(delay=None) | 设置或返回以毫秒为单位的绘图延迟。 |
| turtle.begin\_poly() | 开始记录多边形的顶点。当前的乌龟位置是多边形的第一个顶点。 |
| turtle.end\_poly() | 停止记录多边形的顶点。当前的乌龟位置是多边形的最后一个顶点。将与第一个顶点相连。 |
| turtle.get\_poly() | 返回最后记录的多边形。 |

random.randint(1,10) ) # 产生 1 到 10 的一个整数型随机数