PYTHON

--参考廖雪峰网站

环境: Python3.5 、windows

Python库: [https://docs.python.org/3/library/index.html](%20https:/docs.python.org/3/library/index.html)

### 前言

Python安装完成后进入DOS输入python，可见python版本，此时已进入python交互式环境。

例如，在Python交互式环境下，输入：

>>>1+1

2

在命令行模式下执行：

C:\work>python calc.py

### 文本编辑器

关于文本编辑器可选notepad++。

绝对不能用Word和Windows自带的记事本。Word保存的不是纯文本文件，而记事本会自作聪明地在文件开始的地方加上几个特殊字符（UTF-8 BOM），结果会导致程序运行出现莫名其妙的错误。

文件必须要以.py结尾，其他的都不行。此外，文件名只能是英文字母、数字和下划线的组合。

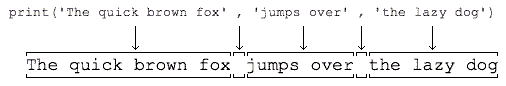
### 输入输出

print()函数也可以接受多个字符串，用逗号“,”隔开，就可以连成一串输出：

>>> print('The quick brown fox', 'jumps over', 'the lazy dog')

The quick brown fox jumps over the lazy dog

print()会依次打印每个字符串，遇到逗号“,”会输出一个空格，因此，输出的字符串是这样拼起来的：



也可打印整数 print(300) 300

>>> print('100 + 200 =', 100 + 200)

100 + 200 = 300

name = input()

当你输入name = input()并按下回车后，Python交互式命令行就在等待你的输入了。这时，你可以输入任意字符，然后按回车后完成输入。

### Python基础

Python的语法比较简单，采用缩进方式：

# print absolute value of an integer:

a = 100

if a >= 0:

print(a)

else:

print(-a)

以#开头的语句是注释，注释是给人看的，可以是任意内容，解释器会忽略掉注释。其他每一行都是一个语句，当语句以冒号:结尾时，缩进的语句视为代码块。请务必注意，Python程序是大小写敏感的，如果写错了大小写，程序会报错。

#### 数据类型和变量

整数(1,10,100,-2)、 浮点数(1.2, 3.4, 1.23x109就是1.23e9)、字符串(字符串是以单引号'或双引号"括起来的任意文本，比如'abc'，"xyz"等等)、 布尔值(True, False)、空值(None)。此外，Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型。

如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，Python允许用'''...'''的格式表示多行内容，(python交互式命令下的执行, 提示符由>>>变为...，提示你可以接着上一行输入)：

>>> print('''line1

... line2

... line3''')

line1

line2

line3

布尔值可以用and、or和not运算。

变量(变量名必须是大小写英文、数字和\_的组合，且不能用数字开头)：

a=1 b=True c=’abc’

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。

常量: 在Python中，通常用全部大写的变量名表示常量：PI=3.1415926。

Python中的两种除法:

>>> 10 / 3

3.3333333333333335

/除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数：

>>> 9 / 3

3.0

还有一种除法是//，称为地板除，两个整数的除法仍然是整数：

>>> 10 // 3

3

#### 字符串和编码



##### **Python的字符串：**

在最新的Python 3版本中，字符串是以Unicode编码的，也就是说，Python的字符串支持多语言，例如：

>>> print('包含中文的str')

包含中文的str

对于单个字符的编码，Python提供了ord()函数获取字符的整数表示，chr()函数把编码转换为对应的字符：

>>> ord('A')

65

>>> chr(25991)

'文'

如果知道字符的整数编码，还可以用十六进制这么写str：

>>> '\u4e2d\u6587'

'中文'

如果要在网络上传输，或者保存到磁盘上，就需要把str变为以字节为单位的bytes。

Python对bytes类型的数据用带b前缀的单引号或双引号表示：

x = b'ABC'

要注意区分'ABC'和b'ABC'，前者是str，后者虽然内容显示得和前者一样，但bytes的每个字符都只占用一个字节。

以Unicode表示的str通过encode()方法可以编码为指定的bytes，例如：

>>> 'ABC'.encode('ascii')

b'ABC'

>>> '中文'.encode('utf-8')

b'\xe4\xb8\xad\xe6\x96\x87'

如果我们从网络或磁盘上读取了字节流，那么读到的数据就是bytes。要把bytes变为str，就需要用decode()方法：

>>> b'ABC'.decode('ascii')

'ABC'

>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')

'中文'

计算str包含多少个字符，可以用len()函数：len('ABC') 3

如果换成bytes，len()函数就计算字节数：

>>> len(b'ABC')

3

>>> len(b'\xe4\xb8\xad\xe6\x96\x87')

6

>>> len('中文'.encode('utf-8'))

6

可见，1个中文字符经过UTF-8编码后通常会占用3个字节，而1个英文字符只占用1个字节。

我们通常在文件开头写上这两行：

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释；

第二行注释是为了告诉Python解释器，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

申明了UTF-8编码并不意味着你的.py文件就是UTF-8编码的，必须并且要确保文本编辑器正在使用UTF-8 without BOM编码。

##### **格式化：**

%运算符就是用来格式化字符串的。在字符串内部，%s表示用字符串替换，%d表示用整数替换，有几个%?占位符，后面就跟几个变量或者值，顺序要对应好。如果只有一个%?，括号可以省略。

常见的占位符有：

|  |  |
| --- | --- |
| %d | 整数 |
| %f | 浮点数 |
| %s | 字符串 |
| %x | 十六进制整数 |

其中，格式化整数和浮点数还可以指定是否补0和整数与小数的位数：

>>> '%2d-%02d' % (3, 1)

' 3-01'

>>> '%.2f' % 3.1415926

'3.14'

如果你不太确定应该用什么，%s永远起作用，它会把任何数据类型转换为字符串：

>>> 'Age: %s. Gender: %s' % (25, True)

'Age: 25. Gender: True'

有些时候，字符串里面的%是一个普通字符怎么办？这个时候就需要转义，用%%来表示一个%：

>>> 'growth rate: %d %%' % 7

'growth rate: 7 %'

### 使用list和tuple

#### list

list是一种有序的集合，可以随时添加和删除其中的元素。

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

>>> classmates

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

用len()函数可以获得list元素的个数：len(classmates) 3

用索引来访问list中每一个位置的元素，记得索引是从0开始，要确保不要越界；还可以用-1做索引，直接获取最后一个元素。

也可以把元素插入到指定的位置，比如索引号为1的位置：

>>> classmates.insert(1, 'Jack')

>>> classmates

['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']

要删除list末尾的元素，用pop()方法：

>>> classmates.pop()

'Adam'

>>> classmates

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

要删除指定位置的元素，用pop(i)方法，其中i是索引位置

要把某个元素替换成别的元素，可以直接赋值给对应的索引位置：

>>> classmates[1] = 'Sarah'

>>> classmates

['Michael', 'Sarah', 'Tracy']

list里面的元素的数据类型也可以不同，比如：

>>> L = ['Apple', 123, True]

list中的复制：a = L 和 a = L[:]，第一种情况a和L指向同一列表，第二种是不同列表

#### tuple

元组，tuple一旦初始化就不能修改。

>>> classmates = ('Michael', 'Bob', 'Tracy')

要定义一个只有1个元素的tuple：t = (2,)，若是t = (3) 定义的式3这个数。

>>> t = ('a', 'b', ['A', 'B'])

>>> t[2][0] = 'X'

>>> t[2][1] = 'Y'

>>> t

('a', 'b', ['X', 'Y'])



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



tuple的元素确实变了，但其实变的不是tuple的元素，而是list的元素。

#### 条件判断

if <条件判断1>:

<执行1>

elif <条件判断2>:

<执行2>

elif <条件判断3>:

<执行3>

else:

<执行4>

注意冒号”:”

#### 循环

Python的循环有两种，一种是for...in循环，依次把list或tuple中的每个元素迭代出来，看例子：

names = ['Michael', 'Bob', 'Tracy']

for name in names:

print(name)

>>> list(range(5))

[0, 1, 2, 3, 4]

range(5)生成的序列是从0开始小于5的整数

第二种循环是while循环，只要条件满足，就不断循环，条件不满足时退出循环。比如我们要计算100以内所有奇数之和，可以用while循环实现：

sum = 0

n = 99

while n > 0:

sum = sum + n

n = n - 2

print(sum)

#### 使用dict和set

##### dict

使用键-值（key-value）存储。

>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}

>>> d['Michael']

95

把数据放入dict的方法，除了初始化时指定外，还可以通过key放入：

>>> d['Adam'] = 67

>>> d['Adam']

67

检查key有两种方法：

一是通过in >>> 'Thomas' in d False

二是通过get方法，如果key不存在，可以返回None，或者自己指定的value：

>>> d.get('Thomas')

>>> d.get('Thomas', -1)

-1

注意：返回None的时候Python的交互式命令行不显示结果。在代码中可通过判断

d.get('Thomas') == None为True or False

要删除一个key，用pop(key)方法，对应的value也会从dict中删除：

>>> d.pop('Bob')

75

>>> d

{'Michael': 95, 'Tracy': 85}

要牢记的一条就是dict的key必须是**不可变对象**。

在Python中，字符串、整数等都是不可变的，因此，可以放心地作为key。而list是可变的，就不能作为key：

>>> key = [1, 2, 3]

>>> d[key] = 'a list'

Traceback (most recent call last):

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

TypeError: unhashable type: 'list'

##### set

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key。

要创建一个set，需要提供一个list作为输入集合：

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

>>> s

{1, 2, 3}

注意，传入的参数[1, 2, 3]是一个list，而显示的{1, 2, 3}只是告诉你这个set内部有1，2，3这3个元素，显示的顺序也不表示set是有序的。

通过add(key)方法可以添加元素到set中，通过remove(key)方法可以删除元素。

注意：此处的是key不是索引，当set中删除不存在的key时会报错。

### 函数

#### 函数的调用和定义

内置函数：<https://docs.python.org/3/library/functions.html>

也可以在交互式命令行通过help(abs)查看abs函数的帮助信息。

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

>>> float('12.34')

12.34

>>> str(1.23)

'1.23'

>>> str(100)

'100'

>>> bool(1)

True

>>> bool('')

False

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：

>>> a = abs # 变量a指向abs函数

>>> a(-1) # 所以也可以通过a调用abs函数

1

Python中函数的定义：

def my\_abs(x):

if x >= 0:

return x

else:

return -x

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。

空函数：

def do\_nothing():

pass

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，Python的函数返回多值其实就是返回一个tuple，但写起来更方便。

**import** math

**def** **move**(x, y, step, angle=0):

nx = x + step \* math.cos(angle)

ny = y - step \* math.sin(angle)

**return** nx, ny

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

>>> print(x, y)

151.96152422706632 70.0

#### 函数的参数

##### 位置参数

def power(x):

return x \* x

对于power(x)函数，参数x就是一个位置参数。当我们调用power函数时，必须传入有且仅有的一个参数x

##### 默认参数

def power(x, n=2):

s = 1

while n > 0:

n = n - 1

s = s \* x

return s

>>> power(5)

25

>>> power(5, 2)

25

>>> power(5, 3)

125

设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

**注意：**

先定义一个函数，传入一个list，添加一个END再返回：

def add\_end(L=[]):

L.append('END')

return L

>>> add\_end()

['end']

>>> add\_end()

['end', 'end']

>>> add\_end()

['end', 'end', 'end']

原因如下：

Python函数在定义的时候，默认参数L的值就被计算出来了，即[]，因为默认参数L也是一个变量，它指向对象[]，每次调用该函数，如果改变了L的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的[]了。

所以，定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用None这个不变对象来实现：

def add\_end(L=None):

if L is None:

L = []

L.append('END')

return L

现在，无论调用多少次，都不会有问题：

>>> add\_end()

['END']

>>> add\_end()

['END']

##### 可变参数

def calc(\*numbers):

sum = 0

for n in numbers:

sum = sum + n \* n

return sum

定义可变参数和定义一个list或tuple参数相比，仅仅在参数前面加了一个\*号。在函数内部，参数numbers接收到的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

>>> calc(1, 2)

5

>>> calc()

0

Python允许你在list或tuple前面加一个\*号，把list或tuple的元素变成可变参数传进去：

>>> nums = [1, 2, 3]

>>> calc(\*nums)

14

##### 关键字参数

关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。请看示例：

def person(name, age, \*\*kw):

print('name:', name, 'age:', age, 'other:', kw)

>>> person('Bob', 35, city='Beijing')

name: Bob age: 35 other: {'city': 'Beijing'}

>>> person('Adam', 45, gender='M', job='Engineer')

name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}

>>> extra = {'city': 'Beijing', 'job': 'Engineer'}

>>> person('Jack', 24, \*\*extra)

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

##### 命名关键字参数

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收city和job作为关键字参数。这种方式定义的函数如下：

def person(name, age, \*, city, job):

print(name, age, city, job)

和关键字参数\*\*kw不同，命名关键字参数需要一个特殊分隔符\*，\*后面的参数被视为命名关键字参数。

调用方式如下：

>>> person('Jack', 24, city='Beijing', job='Engineer')

Jack 24 Beijing Engineer

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符\*了：

def person(name, age, \*args, city, job):

print(name, age, args, city, job)

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

>>> person('Jack', 24, 'Beijing', 'Engineer')

Traceback (most recent call last):

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

TypeError: person() takes 2 positional arguments but 4 were given

由于调用时缺少参数名city和job，Python解释器把这4个参数均视为位置参数，但person()函数仅接受2个位置参数。

命名关键字参数可以有缺省值，从而简化调用：

def person(name, age, \*, city='Beijing', job):

print(name, age, city, job)

由于命名关键字参数city具有默认值，调用时，可不传入city参数：

>>> person('Jack', 24, job='Engineer')

Jack 24 Beijing Engineer

注意：只有在可变参数出现时，后面跟着的命名关键字参数就不再需要一个特殊分隔符\*。

##### 参数组合

参数定义的顺序必须是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数。

比如定义一个函数，包含上述若干种参数：

**def** **f1**(a, b, c=0, \*args, \*\*kw):

print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

**def** **f2**(a, b, c=0, \*, d, \*\*kw):

print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

>>> f1(1, 2)

a = 1 b = 2 c = 0 args = () kw = {}

>>> f1(1, 2, c=3)

a = 1 b = 2 c = 3 args = () kw = {}

>>> f1(1, 2, 3, 'a', 'b')

a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}

>>> f1(1, 2, 3, 'a', 'b', x=99)

a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}

>>> f2(1, 2, d=99, ext=None)

a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}

最神奇的是通过一个tuple和dict，你也可以调用上述函数：

>>> args = (1, 2, 3, 4)

>>> kw = {'d': 99, 'x': '#'}

>>> f1(\*args, \*\*kw)

a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}

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

>>> kw = {'d': 88, 'x': '#'}

>>> f2(\*args, \*\*kw)

a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}

所以，对于任意函数，都可以通过类似func(\*args, \*\*kw)的形式调用它，无论它的参数是如何定义的。

要注意定义可变参数和关键字参数的语法：

\*args是可变参数，args接收的是一个tuple；

\*\*kw是关键字参数，kw接收的是一个dict。

### 高级特性

#### 切片

取前n个元素：L[0:3] or L[:3]；取倒数第一个L[-1]，取后两个L[-2:]

前10个数，每两个取一个：

>>> L[:10:2]

[0, 2, 4, 6, 8]

所有数，每5个取一个：

>>> L[::5]

[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]

甚至什么都不写，只写[:]就可以原样复制一个list：

>>> L[:]

[0, 1, 2, 3, ..., 99]

tuple也是一种list，唯一区别是tuple不可变。因此，tuple也可以用切片操作，只是操作的结果仍是tuple：

>>> (0, 1, 2, 3, 4, 5)[:3]

(0, 1, 2)

字符串'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：

>>> 'ABCDEFG'[:3]

'ABC'

>>> 'ABCDEFG'[::2]

'ACEG'

#### 迭代

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

默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.values()，如果要同时迭代key和value，可以用for k, v in d.items()。

如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：

>>> from collections import Iterable

>>> isinstance('abc', Iterable) # str是否可迭代

True

>>> isinstance([1,2,3], Iterable) # list是否可迭代

True

>>> isinstance(123, Iterable) # 整数是否可迭代

False

如果要对list实现类似Java那样的下标循环怎么办？Python内置的enumerate函数可以把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

>>> for i, value in enumerate(['A', 'B', 'C']):

print(i, value)

0 A

1 B

2 C

#### 列表生成式

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

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

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

[4, 16, 36, 64, 100]

>>> [m + n **for** m **in** 'ABC' **for** n **in** 'XYZ']

['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']

注意：如果list中既包含字符串，又包含整数，由于非字符串类型没有lower()方法，可用isinstance方法来判断。

#### 生成器

由于创建大的列表容量很耗资源，可以采用生成器方法

>>> L = [x \* x **for** x **in** range(10)]

>>> L

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> g = (x \* x **for** x **in** range(10))

>>> g

<generator object <genexpr> at 0x1022ef630>

创建L和g的区别仅在于最外层的[]和()，L是一个list，而g是一个generator。如果要一个一个打印出来，可以通过next()函数获得generator的下一个返回值：

>>> next(g)

0

>>> next(g)

1

…

>>> next(g)

81

>>> next(g)

Traceback (most recent call last):

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

StopIteration

正确的方法是使用for循环，因为generator也是可迭代对象:

>>> for n in g:

... print(n)

...

0

1. ….

def fib(max):

n, a, b = 0, 0, 1

while n < max:

yield b

a, b = b, a + b

n = n + 1

return 'done'

这就是定义generator的另一种方法。如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator：

>>> f = fib(6)

>>> f

<generator object fib at 0x104feaaa0>

#### 迭代器

可以直接作用于for循环的数据类型有以下几种：

一类是集合数据类型，如list、tuple、dict、set、str等；

一类是generator，包括生成器和带yield的generator function。

这些可以直接作用于for循环的对象统称为可迭代对象：Iterable。

可以使用isinstance()判断一个对象是否是Iterable对象：

>>> from collections import Iterable

>>> isinstance([], Iterable)

True

>>> isinstance({}, Iterable)

True

>>> isinstance('abc', Iterable)

True

>>> isinstance((x for x in range(10)), Iterable)

True

>>> isinstance(100, Iterable)

False

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。

可以使用isinstance()判断一个对象是否是Iterator对象：

>>> from collections import Iterator

>>> isinstance((x for x in range(10)), Iterator)

True

>>> isinstance([], Iterator)

False

>>> isinstance({}, Iterator)

False

>>> isinstance('abc', Iterator)

False

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

>>> isinstance(iter([]), Iterator)

True

>>> isinstance(iter('abc'), Iterator)

True

你可能会问，为什么list、dict、str等数据类型不是Iterator？

这是因为Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

### 函数式编程

#### 高阶函数

##### map/reduce

>>> def f(x):

... return x \* x

...

>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> list(r)

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

map()传入的第一个参数是f，即函数对象本身。由于结果r是一个Iterator，Iterator是惰性序列，因此通过list()函数让它把整个序列都计算出来并返回一个list。

所以，map()作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x)=x2，还可以计算任意复杂的函数，比如，把这个list所有数字转为字符串：

>>> list(map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

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

reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算，其效果就是：

reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)

##### filter

def \_odd\_iter():

n = 1

while True:

n = n + 2

yield n

def \_not\_divisible(n):

return lambda x: x % n > 0

def primes():

yield 2

it = \_odd\_iter() # 初始序列

while True:

n = next(it) # 返回序列的第一个数

yield n

it = filter(\_not\_divisible(n), it) # 构造新序列

##### 返回函数

返回的函数并没有立刻执行，而是直到调用了f()才执行。我们来看一个例子：

def count():

fs = []

for i in range(1, 4):

def f():

return i\*i

fs.append(f)

return fs

f1, f2, f3 = count()

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果是

>>> f1()

9

>>> f2()

9

>>> f3()

9

原因就在于返回的函数引用了变量i，但它并非立刻执行。等到3个函数都返回时，它们所引用的变量i已经变成了3，因此最终结果为9。返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

##### 装饰器

本质上，decorator就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的decorator，可以定义如下：

def log(func):

def wrapper(\*args, \*\*kw):

print('call %s():' % func.\_\_name\_\_)

return func(\*args, \*\*kw)

return wrapper

@log

def now():

print('2015-3-25')

调用now()函数，不仅会运行now()函数本身，还会在运行now()函数前打印一行日志：

>>> now()

call now():

2015-3-25

如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂。比如，要自定义log的文本：

def log(text):

def decorator(func):

def wrapper(\*args, \*\*kw):

print('%s %s():' % (text, func.\_\_name\_\_))

return func(\*args, \*\*kw)

return wrapper

return decorator

这个3层嵌套的decorator用法如下：

@log('execute')

def now():

print('2015-3-25')

以上两种decorator的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有\_\_name\_\_等属性，但你去看经过decorator装饰之后的函数，它们的\_\_name\_\_已经从原来的'now'变成了'wrapper'：

>>> now.\_\_name\_\_

'wrapper'

因为返回的那个wrapper()函数名字就是'wrapper'，所以，需要把原始函数的\_\_name\_\_等属性复制到wrapper()函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写wrapper.\_\_name\_\_ = func.\_\_name\_\_这样的代码，Python内置的functools.wraps就是干这个事的，所以，一个完整的decorator的写法如下：

import functools

def log(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kw):

print('call %s():' % func.\_\_name\_\_)

return func(\*args, \*\*kw)

return wrapper

### 面向对象编程

#### 类

**class Student(object):**

**pass**

class后面紧接着是类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

class Student(object):

def \_\_init\_\_(self, name, score):

self.name = name

self.score = score

有了\_\_init\_\_方法，在创建实例的时候，就不能传入空的参数了，必须传入与\_\_init\_\_方法匹配的参数，但self不需要传，Python解释器自己会把实例变量传进去。

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，在Python中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问。

在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用\_\_name\_\_、\_\_score\_\_这样的变量名。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问\_\_name是因为Python解释器对外把\_\_name变量改成了\_Student\_\_name，所以，仍然可以通过\_Student\_\_name来访问\_\_name变量：

>>> bart.\_Student\_\_name

'Bart Simpson'

但是强烈建议你不要这么干，因为不同版本的Python解释器可能会把\_\_name改成不同的变量名。

最后注意下面的这种错误写法：

>>> bart = Student('Bart Simpson', 98)

>>> bart.get\_name()

'Bart Simpson'

>>> bart.\_\_name = 'New Name' # 设置\_\_name变量！

>>> bart.\_\_name

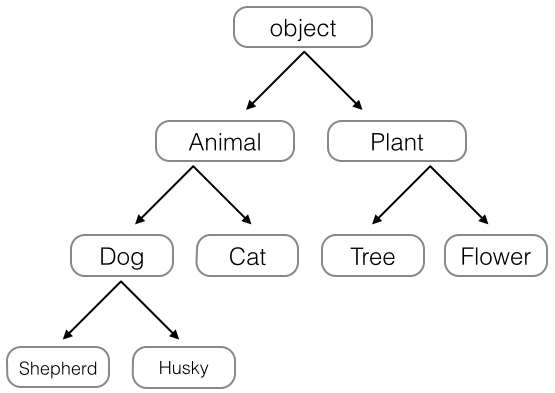
'New Name'

表面上看，外部代码“成功”地设置了\_\_name变量，但实际上这个\_\_name变量和class内部的\_\_name变量*不是*一个变量！内部的\_\_name变量已经被Python解释器自动改成了\_Student\_\_name，而外部代码给bart新增了一个\_\_name变量。不信试试：

>>> bart.get\_name() *# get\_name()内部返回self.\_\_name*

'Bart Simpson'

#### 继承和多态



|  |  |  |
| --- | --- | --- |
| class Animal(object):  def run(self):  print('Animal is running...') | class Dog(Animal):  def run(self):  print('Dog is running...') | class Cat(Animal):  def run(self):  print('Cat is running...') |
|  |  | def run\_twice(animal):  animal.run()  animal.run() |

静态语言 vs 动态语言

对于静态语言（例如Java）来说，如果需要传入Animal类型，则传入的对象必须是Animal类型或者它的子类，否则，将无法调用run()方法。

对于Python这样的动态语言来说，则不一定需要传入Animal类型。我们只需要保证传入的对象有一个run()方法就可以了：

class Timer(object):

def run(self):

print('Start...')

>>> run\_twice(Timer())

Start...

Start...

这就是动态语言的“鸭子类型”，它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。

Python的“file-like object“就是一种鸭子类型。对真正的文件对象，它有一个read()方法，返回其内容。但是，许多对象，只要有read()方法，都被视为“file-like object“。许多函数接收的参数就是“file-like object“，你不一定要传入真正的文件对象，完全可以传入任何实现了read()方法的对象。

#### 实例属性和类属性

给实例绑定属性的方法是通过实例变量，或者通过self变量：

class Student(object):

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

self.name = name

但是，如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归Student类所有：

class Student(object):

name = 'Student'

当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。来测试一下：

>>> class Student(object):

... name = 'Student'

...

>>> s = Student() # 创建实例s

>>> print(s.name) # 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性

Student

>>> print(Student.name) # 打印类的name属性

Student

>>> s.name = 'Michael' # 给实例绑定name属性

>>> print(s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性

Michael

>>> print(Student.name) # 但是类属性并未消失，用Student.name仍然可以访问

Student

>>> del s.name # 如果删除实例的name属性

>>> print(s.name) # 再次调用s.name，由于实例的name属性没有找到，类的name属性就显示出来了

Student

在编写程序的时候，千万不要把实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

#### 使用\_\_slots\_\_

定义一个特殊的\_\_slots\_\_变量，来限制该class实例能添加的属性：

**class Student(object):**

\_\_slots\_\_ = ('name', 'age') *# 用tuple定义允许绑定的属性名称*

使用\_\_slots\_\_要注意，\_\_slots\_\_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的。

Python内置的@property装饰器就是负责把一个方法变成属性调用的：

class Student(object):

@property

def score(self):

return self.\_score

@score.setter

def score(self, value):

if not isinstance(value, int):

raise ValueError('score must be an integer!')

if value < 0 or value > 100:

raise ValueError('score must between 0 ~ 100!')

self.\_score = value

@property的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上@property就可以了，此时，@property本身又创建了另一个装饰器@score.setter，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

>>> s = Student()

>>> s.score = 60 # OK，实际转化为s.set\_score(60)

>>> s.score # OK，实际转化为s.get\_score()

60

>>> s.score = 9999

Traceback (most recent call last):

...

ValueError: score must between 0 ~ 100!

注意到这个神奇的@property，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性

### 错误、调试和测试

#### 错误

try:

print('try...')

r = 10 / 0

print('result:', r)

except ZeroDivisionError as e:

print('except:', e)

finally:

print('finally...')

Python的错误其实也是class，所有的错误类型都继承自BaseException，所以在使用except时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。

如果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用raise语句抛出一个错误的实例：

# err\_raise.py

class FooError(ValueError):

pass

def foo(s):

n = int(s)

if n==0:

raise FooError('invalid value: %s' % s)

return 10 / n

foo('0')

#### 调试

##### Assert

**def** **foo**(s):

n = int(s)

**assert** n != 0, 'n is zero!'

**return** 10 / n

assert的意思是，表达式n != 0应该是True，否则，根据程序运行的逻辑，后面的代码肯定会出错。如果断言失败，assert语句本身就会抛出AssertionError

程序中如果到处充斥着assert，和print()相比也好不到哪去。不过，启动Python解释器时可以用-O参数来关闭assert：

$ python3 -O err.py

Traceback (most recent **call** **last**):

...

ZeroDivisionError: division **by** zero

##### Logging

import logging

logging.basicConfig(level=logging.INFO)

s = '0'

n = int(s)

logging.info('n = %d' % n)

print(10 / n)

允许你指定记录信息的级别，有debug，info，warning，error等几个级别，当我们指定level=INFO时，logging.debug就不起作用了。同理，指定level=WARNING后，debug和info就不起作用。

>>>python -m pdb xxcc.py

以参数-m pdb启动后，pdb定位到下一步要执行的代码,输入命令l来查看代码, 输入命令n可以单步执行代码, 输入命令p 变量名来查看变量, 输入命令q结束调试，退出程序.

##### 单元测试

类Dict：--- mydict.py

class Dict(dict):

def \_\_init\_\_(self, \*\*kw):

super().\_\_init\_\_(\*\*kw)

def \_\_getattr\_\_(self, key):

try:

return self[key]

except KeyError:

raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

def \_\_setattr\_\_(self, key, value):

self[key] = value

单元测试类:--- mydict\_test.py

import unittest

from mydict import Dict

class TestDict(unittest.TestCase):

def setUp(self):

print('setUp...')

def tearDown(self):

print('tearDown...')

def test\_init(self):

d = Dict(a=1, b='test')

self.assertEqual(d.a, 1)

self.assertEqual(d.b, 'test')

self.assertTrue(isinstance(d, dict))

def test\_key(self):

d = Dict()

d['key'] = 'value'

self.assertEqual(d.key, 'value')

def test\_attr(self):

d = Dict()

d.key = 'value'

self.assertTrue('key' in d)

self.assertEqual(d['key'], 'value')

def test\_keyerror(self):

d = Dict()

with self.assertRaises(KeyError):

value = d['empty']

def test\_attrerror(self):

d = Dict()

with self.assertRaises(AttributeError):

value = d.empty

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

##### doctest

Python内置的“文档测试”（doctest）模块可以直接提取注释中的代码并执行测试。doctest严格按照Python交互式命令行的输入和输出来判断测试结果是否正确。只有测试异常的时候，可以用...表示中间一大段烦人的输出。运行程序，什么输出也没有表示测试通过。

if \_\_name\_\_=='\_\_main\_\_':

import doctest

doctest.testmod()

### IO编程

#### 文件读写

以读文件的模式打开一个文件对象，使用Python内置的open()函数，传入文件名和标示符：

>>> f = open('/Users/michael/test.txt', 'r')

由于文件读写时都有可能产生IOError，一旦出错，后面的f.close()就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用try ... finally来实现：

try:

f = open('/path/to/file', 'r')

print(f.read())

finally:

if f:

f.close()

但是每次都这么写实在太繁琐，所以，Python引入了with语句来自动帮我们调用close()方法：

with open('/path/to/file', 'r') as f:

print(f.read())

这和前面的try ... finally是一样的，但是代码更佳简洁，并且不必调用f.close()方法。如果文件很小，read()一次性读取最方便；如果不能确定文件大小，反复调用read(size)比较保险；如果是配置文件，调用readlines()

### 访问数据库

#### Sqlite

#### Mysql

在Windows上，安装时请选择UTF-8编码，以便正确地处理中文；在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在/etc/my.cnf或者/etc/mysql/my.cnf：

**[client]**

default-character-set = utf8

**[mysqld]**

default-storage-engine = INNODB

character-set-server = utf8

collation-server = utf8\_general\_ci



Mysql安装驱动：

$ pip install mysql-connector-python --allow-external mysql-connector-python

或者

$ pip install mysql-connector

在Python中，最有名的ORM框架是SQLAlchemy。

pip install sqlalchemy

### 异步IO

#### 协程(Coroutine)

协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。Python对协程的支持是通过generator实现的。

#### asyncio

asyncio是Python 3.4版本引入的标准库，直接内置了对异步IO的支持。

asyncio的编程模型就是一个消息循环。我们从asyncio模块中直接获取一个EventLoop的引用，然后把需要执行的协程扔到EventLoop中执行，就实现了异步IO。