## Python基础

### 输出: print

用print加上字符串，就可以向屏幕上输出指定的文字。比如输出'hello, world'

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，遇到逗号“,”会输出一个空格

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

The quick brown fox jumps over the lazy dog

### 输入: raw\_input()

raw\_input，可以让用户输入字符串，并存放到一个变量里，并且可以添加输入提示；

>>> name = raw\_input('please enter your name:')

please enter your name:CaiMao

>>> print 'Hello,',name

Hello, CaiMao

### 数据类型和变量

#### 整数

Python可以处理任意大小的整数，在程序中的表示方法和数学上的写法一模一样，例如：1，100，-8080，0，等等。计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用0x前缀和0-9，a-f表示，例如：0xff00，0xa5b4c3d2，等等。

#### 浮点数

浮点数也就是小数，之所以称为浮点数，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，比如，1.23x109和12.3x108是相等的。浮点数可以用数学写法，如1.23，3.14，-9.01，等等。但是对于很大或很小的浮点数，就必须用科学计数法表示，把10用e替代，1.23x109就是1.23e9，或者12.3e8，0.000012可以写成1.2e-5，等等。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（除法难道也是精确的？是的！），而浮点数运算则可能会有四舍五入的误差。

#### 字符串

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。字符串内部既包含'又包含"可以用转义字符\来标识;

转义字符\可以转义很多字符，比如\n表示换行，\t表示制表符，字符\本身也要转义，所以[\\表示的字符就是\](file:///\\表示的字符就是\);

Python还允许用r''表示''内部的字符串默认不转义;

>>> **print** 'I\'m ok.'

I'm ok.

>>> print 'I\'m learning\nPython.'

I'm learning

Python.

>>> print '\\\n\\'

\

\

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, 多行字符串'''...'''还可以在前面加上r使用

**print** '''line1

line2

line3'''

#### 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，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），也可以通过布尔运算计算出来

#### 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。

#### 变量

变量在程序中就是用一个变量名表示了，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; 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。例如Java是静态语言;

#### 常量

所谓常量就是不能变的变量，比如常用的数学常数π就是一个常量。在Python中，通常用全部大写的变量名表示常量；

### 字符串和编码

#### 字符编码

计算机只能处理数字，如果要处理文本，就必须先把文本转换为数字才能处理。最早的计算机在设计时采用8个比特（bit）作为一个字节（byte），所以，一个字节能表示的最大的整数就是255（二进制11111111=十进制255），如果要表示更大的整数，就必须用更多的字节。比如两个字节可以表示的最大整数是65535，4个字节可以表示的最大整数是4294967295。由于计算机是美国人发明的，因此，最早只有127个字母被编码到计算机里，也就是大小写英文字母、数字和一些符号，这个编码表被称为ASCII编码，比如大写字母A的编码是65，小写字母z的编码是122。但是要处理中文显然一个字节是不够的，至少需要两个字节，而且还不能和ASCII编码冲突，所以，中国制定了GB2312编码，用来把中文编进去。Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。Unicode标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。现代操作系统和大多数编程语言都直接支持Unicode。UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用UTF-8编码就能节省空间：

#### Python的字符串

Python提供了ord()和chr()函数，可以把字母和对应的数字相互转换：

>>> ord('A')

65

>>> chr(65)

'A'

Python在后来添加对Unicode的支持，以Unicode表示的字符串用u'...'表示，比如：

>>> **print** u'中文'

中文

>>> u'中'

u'\u4e2d'

把u'xxx'转换为UTF-8编码的'xxx'用encode('utf-8')方法：

>>> u'ABC'.encode('utf-8')

'ABC'

>>> u'中文'.encode('utf-8')

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

UTF-8编码表示的字符串'xxx'转换为Unicode字符串u'xxx'用decode('utf-8')方法：

>>> 'abc'.decode('utf-8')

u'abc'

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

u'\u4e2d\u6587'

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

中文

由于Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

*#!/usr/bin/env python*

*# -\*- 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。

如果你使用Notepad++进行编辑，除了要加上# -\*- coding: utf-8 -\*-外，中文字符串必须是Unicode字符串：

#### 格式化

在Python中，采用的格式化方式和C语言是一致的，用%实现；%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。

%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'

有时候，字符串里面的%是一个普通字符时就需要转义，用%%来表示一个%：

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

'growth rate: 7 %'

### 使用list和tuple

#### 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。L = [e1, e2, e3, …]

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

>>> classmates

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

变量classmates就是一个list。用len()函数可以获得list元素的个数：用索引来访问list中每一个位置的元素，索引是从0开始的：当索引超出了范围时，Python会报一个IndexError错误，所以，要确保索引不要越界，记得最后一个元素的索引是len(classmates) - 1。如果要取最后一个元素，除了计算索引位置外，还可以用-1做索引，直接获取最后一个元素：

>>> classmates[0]

'Michael'

>>> classmates[1]

'Bob'

>>> classmates[3]

Traceback (most recent call last):

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

IndexError: list index out of range

>>> classmates[-1]

'Tracy'

list是一个可变的有序表，所以，可以往list中追加元素到末尾：append()

>>> classmates.append('Adam')

>>> classmates

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

也可以把元素插入到指定的位置，比如索引号为1的位置：insert(index, element)

>>> 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：

>>> classmates.pop(1)

'Jack'

>>> classmates

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

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元素也可以是另一个list，比如：

>>> s = ['python', 'java', ['asp', 'php'], 'scheme']

>>> len(s)

4

#### tuple

另一种有序列表叫元组：tuple。tuple和list非常类似，但是tuple一旦初始化就不能修改，比如同样是列出同学的名字：T = (e1, e2, e3, …)

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

现在，classmates这个tuple不能变了，它也没有append()，insert()这样的方法。其他获取元素的方法和list是一样的，你可以正常地使用classmates[0]，classmates[-1]，但不能赋值成另外的元素。不可变的tuple有什么意义？因为tuple不可变，所以代码更安全。如果可能，能用tuple代替list就尽量用tuple。

要定义一个只有1个元素的tuple，因为括号()既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，只有1个元素的tuple定义时必须加一个逗号,，来消除歧义：

>>> t = (1,)

>>> t

(1,)

tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。即指向'a'，就不能改成指向'b'，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！

理解了“指向不变”后，要创建一个内容也不变的tuple怎么做？那就必须保证tuple的每一个元素本身也不能变。

### 条件判断和循环

#### 条件判断

if语句的完整形式就是：

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>

if语句执行有个特点，它是从上往下判断，如果在某个判断上是True，把该判断对应的语句执行后，就忽略掉剩下的elif和else

if判断条件还可以简写，比如写：

**if** x:

**print** 'True'

只要x是非零数值、非空字符串、非空list等，就判断为True，否则为False。

#### 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：

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

**for** name **in** names:

**print** name

for x in ...循环就是把每个元素代入变量x，然后执行缩进块的语句。

range()函数，可以生成一个整数序列，比如range(5)生成序列是从0开始小于5的整数：

>>> range(5)

[0, 1, 2, 3, 4]

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

在循环内部变量n不断自减，直到变为-1时，不再满足while条件，循环退出。

### 使用dict和set

#### dict

Python内置了字典：dict的支持，dict全称dictionary，在其他语言中也称为map，使用键-值（key-value）存储，具有极快的查找速度。D = {k1:v1, k2:v2, …}

>>> 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

一个key只能对应一个value，多次对一个key放入value，后面的值会覆盖前面的值：

>>> d['Jack'] = 90

>>> d['Jack']

90

>>> d['Jack'] = 88

>>> d['Jack']

88

如果key不存在，dict就会报错：

>>> d['Thomas']

Traceback (most recent call last):

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

KeyError: 'Thomas'

要避免key不存在的错误，有两种办法，一是通过in判断key是否存在：

>>> 'Thomas' **in** d

False

二是通过dict提供的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。

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}

请务必注意，dict内部存放的顺序和key放入的顺序是没有关系的。

和list比较，dict有以下几个特点：

1. 查找和插入的速度极快，不会随着key的增加而增加；
2. 需要占用大量的内存，内存浪费多。

而list相反：

1. 查找和插入的时间随着元素的增加而增加；
2. 占用空间小，浪费内存很少。

所以，dict是用空间来换取时间的一种方法。

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用dict非常重要，需要牢记的第一条就是dict的key必须是**不可变对象**。

这是因为dict根据key来计算value的存储位置，如果每次计算相同的key得出的结果不同，那dict内部就完全混乱了。这个通过key计算位置的算法称为哈希算法（Hash）。

要保证hash的正确性，作为key的对象就不能变。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（重复元素在set中自动被过滤），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

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

注意，传入的参数[1, 2, 3]是一个list，而显示的set([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[]不表示这是一个list。

通过add(key)方法可以添加元素到set中，可以重复添加，但不会有效果：

>>> s.add(4)

>>> s

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

>>> s.add(4)

>>> s

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

通过remove(key)方法可以删除元素：

>>> s.remove(4)

>>> s

set([1, 2, 3])

set可以看成数学意义上的无序和无重复元素的集合，因此，两个set可以做数学意义上的交集、并集等操作：

>>> s1 = **set**([1, 2, 3])

>>> s2 = **set**([2, 3, 4])

>>> s1 & s2

**set**([2, 3])

>>> s1 | s2

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

set和dict的唯一区别仅在于没有存储对应的value，但是，set的原理和dict一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证set内部“不会有重复元素”。

使用key-value存储结构的dict在Python中非常有用，选择不可变对象作为key很重要，最常用的key是字符串。

### 函数

#### 调用函数

Python的官方网站查看文档： <http://docs.python.org/2/library/functions.html#abs> ；也可以在交互式命令行通过help(abs)查看abs函数的帮助信息。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Built-in Functions** |  |  |  |
| [**abs()**](https://docs.python.org/2/library/functions.html#abs) | [**divmod()**](https://docs.python.org/2/library/functions.html#divmod) | [**input()**](https://docs.python.org/2/library/functions.html#input) | [**open()**](https://docs.python.org/2/library/functions.html#open) | [**staticmethod()**](https://docs.python.org/2/library/functions.html#staticmethod) |
| [**all()**](https://docs.python.org/2/library/functions.html#all) | [**enumerate()**](https://docs.python.org/2/library/functions.html#enumerate) | [**int()**](https://docs.python.org/2/library/functions.html#int) | [**ord()**](https://docs.python.org/2/library/functions.html#ord) | [**str()**](https://docs.python.org/2/library/functions.html#str) |
| [**any()**](https://docs.python.org/2/library/functions.html#any) | [**eval()**](https://docs.python.org/2/library/functions.html#eval) | [**isinstance()**](https://docs.python.org/2/library/functions.html#isinstance) | [**pow()**](https://docs.python.org/2/library/functions.html#pow) | [**sum()**](https://docs.python.org/2/library/functions.html#sum) |
| [**basestring()**](https://docs.python.org/2/library/functions.html#basestring) | [**execfile()**](https://docs.python.org/2/library/functions.html#execfile) | [**issubclass()**](https://docs.python.org/2/library/functions.html#issubclass) | [**print()**](https://docs.python.org/2/library/functions.html#print) | [**super()**](https://docs.python.org/2/library/functions.html#super) |
| [**bin()**](https://docs.python.org/2/library/functions.html#bin) | [**file()**](https://docs.python.org/2/library/functions.html#file) | [**iter()**](https://docs.python.org/2/library/functions.html#iter) | [**property()**](https://docs.python.org/2/library/functions.html#property) | [**tuple()**](https://docs.python.org/2/library/functions.html#tuple) |
| [**bool()**](https://docs.python.org/2/library/functions.html#bool) | [**filter()**](https://docs.python.org/2/library/functions.html#filter) | [**len()**](https://docs.python.org/2/library/functions.html#len) | [**range()**](https://docs.python.org/2/library/functions.html#range) | [**type()**](https://docs.python.org/2/library/functions.html#type) |
| [**bytearray()**](https://docs.python.org/2/library/functions.html#bytearray) | [**float()**](https://docs.python.org/2/library/functions.html#float) | [**list()**](https://docs.python.org/2/library/functions.html#func-list) | [**raw\_input()**](https://docs.python.org/2/library/functions.html#raw_input) | [**unichr()**](https://docs.python.org/2/library/functions.html#unichr) |
| [**callable()**](https://docs.python.org/2/library/functions.html#callable) | [**format()**](https://docs.python.org/2/library/functions.html#format) | [**locals()**](https://docs.python.org/2/library/functions.html#locals) | [**reduce()**](https://docs.python.org/2/library/functions.html#reduce) | [**unicode()**](https://docs.python.org/2/library/functions.html#unicode) |
| [**chr()**](https://docs.python.org/2/library/functions.html#chr) | [**frozenset()**](https://docs.python.org/2/library/functions.html#func-frozenset) | [**long()**](https://docs.python.org/2/library/functions.html#long) | [**reload()**](https://docs.python.org/2/library/functions.html#reload) | [**vars()**](https://docs.python.org/2/library/functions.html#vars) |
| [**classmethod()**](https://docs.python.org/2/library/functions.html#classmethod) | [**getattr()**](https://docs.python.org/2/library/functions.html#getattr) | [**map()**](https://docs.python.org/2/library/functions.html#map) | [**repr()**](https://docs.python.org/2/library/functions.html#func-repr) | [**xrange()**](https://docs.python.org/2/library/functions.html#xrange) |
| [**cmp()**](https://docs.python.org/2/library/functions.html#cmp) | [**globals()**](https://docs.python.org/2/library/functions.html#globals) | [**max()**](https://docs.python.org/2/library/functions.html#max) | [**reversed()**](https://docs.python.org/2/library/functions.html#reversed) | [**zip()**](https://docs.python.org/2/library/functions.html#zip) |
| [**compile()**](https://docs.python.org/2/library/functions.html#compile) | [**hasattr()**](https://docs.python.org/2/library/functions.html#hasattr) | [**memoryview()**](https://docs.python.org/2/library/functions.html#func-memoryview) | [**round()**](https://docs.python.org/2/library/functions.html#round) | [**\_\_import\_\_()**](https://docs.python.org/2/library/functions.html#__import__) |
| [**complex()**](https://docs.python.org/2/library/functions.html#complex) | [**hash()**](https://docs.python.org/2/library/functions.html#hash) | [**min()**](https://docs.python.org/2/library/functions.html#min) | [**set()**](https://docs.python.org/2/library/functions.html#func-set) |  |
| [**delattr()**](https://docs.python.org/2/library/functions.html#delattr) | [**help()**](https://docs.python.org/2/library/functions.html#help) | [**next()**](https://docs.python.org/2/library/functions.html#next) | [**setattr()**](https://docs.python.org/2/library/functions.html#setattr) |  |
| [**dict()**](https://docs.python.org/2/library/functions.html#func-dict) | [**hex()**](https://docs.python.org/2/library/functions.html#hex) | [**object()**](https://docs.python.org/2/library/functions.html#object) | [**slice()**](https://docs.python.org/2/library/functions.html#slice) |  |
| [**dir()**](https://docs.python.org/2/library/functions.html#dir) | [**id()**](https://docs.python.org/2/library/functions.html#id) | [**oct()**](https://docs.python.org/2/library/functions.html#oct) | [**sorted()**](https://docs.python.org/2/library/functions.html#sorted) |  |

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

>>> float('3.1415926')

3.1415926

>>> str(1.2345)

'1.2345'

>>> unicode(100)

u'100'

>>> bool(1)

True

>>> bool('')

False

>>> bool(None)

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

>>>

#### 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。

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。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。

##### 空函数

如果想定义一个什么事也不做的空函数，可以用pass语句：

>>> def nop():

... pass

...

pass语句什么都不做，可以用来作为占位符，比如还没想好怎么写函数的代码，就可以先放一个pass，让代码能运行起来。

##### 参数检查

调用函数时，如果参数个数不对，Python解释器会自动检查出来，并抛出TypeError：

##### 返回多个值

Python的函数返回多值其实就是返回一个tuple

#### 函数的参数

Python的函数定义除了正常定义的**必选参数**外，还可以使用**默认参数**、**可变参数**和**关键字参数**，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

##### 默认参数

默认参数可以简化函数的调用。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；

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。

使用默认参数最大的好处是能降低调用函数的难度。

>>> 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

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。

>>> def enroll(name, gender, age = 6, city = 'Chengdu'):

...

print 'name:',name,',gender:',gender,',age:',age,',city:',city

...

>>> enroll('Panda','F')

name: Panda ,gender: F ,age: 6 ,city: Chengdu

>>> enroll('Greatwall','M',10)

name: Greatwall ,gender: M ,age: 10 ,city: Chengdu

>>> enroll('Greatwall','M',city = 'Beijing')

name: Greatwall ,gender: M ,age: 6 ,city: Beijing

>>>

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！

>>> add\_end([1,2,3])

[1, 2, 3, 'END']

>>> add\_end(['x','y','z'])

['x', 'y', 'z', 'END']

>>> add\_end()

['END']

>>> add\_end()

['END', 'END']

>>> add\_end()

['END', 'END', 'END']

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

...

>>> add\_end()

['END']

>>> add\_end()

['END']

>>> add\_end()

['END']

##### 可变参数

可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。

**def** **calc**(\*numbers):

sum = 0

**for** n **in** numbers:

sum = sum + n \* n

**return** sum

定义可变参数和定义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。

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

**print** 'name:', name, 'age:', age, 'other:', kw

函数person除了必选参数name和age外，还接受关键字参数kw。在调用该函数时，可以只传入必选参数：

>>> person('Michael', 30)

name: Michael age: 30 other: {}

也可以传入任意个数的关键字参数：

>>> 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'}

和可变参数类似，也可以先组装出一个dict，然后，把该dict转换为关键字参数传进去：

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

>>> person('Jack', 24, city=kw['city'], job=kw['job'])

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

当然，上面复杂的调用可以用简化的写法：

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

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

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

##### 参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数和关键字参数，这4种参数都可以一起使用，或者只用其中某些，但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数和关键字参数。

默认参数一定要用不可变对象，如果是可变对象，运行会有逻辑错误！

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。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：func(1, 2, 3)，又可以先组装list或tuple，再通过\*args传入：func(\*(1, 2, 3))；

关键字参数既可以直接传入：func(a=1, b=2)，又可以先组装dict，再通过\*\*kw传入：func(\*\*{'a': 1, 'b': 2})。

使用\*args和\*\*kw是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

#### 递归函数

在函数内部，可以调用其他函数。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。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。

解决递归调用栈溢出的方法是通过**尾递归**优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。尾递归是指，在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以也会导致栈溢出。

### 高级特性

#### 切片

Python提供切片（Slice）操作符用于经常取指定索引范围的操作。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。如果第一个索引是0，用L[:3] ；甚至什么都不写，只写[:]就可以原样复制一个list；tuple也可以用切片操作，只是操作的结果仍是tuple

>>> L = ['Tom','Cat','Jerry','Leilei']

>>> L[0:2]

['Tom', 'Cat']

>>> L[:2]

['Tom', 'Cat']

>>> L[-2:]

['Jerry', 'Leilei']

>>> L[:]

['Tom', 'Cat', 'Jerry', 'Leilei']

#### 迭代

如果给定一个list或tuple，可以通过for循环来遍历这个list或tuple，这种遍历称为迭代（Iteration）。只要是可迭代对象，无论有无下标，都可以迭代。

>>> d = {'a':1, 'b':2, 'c':3}

>>> for key in d:

... print key

...

a

c

b

因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.itervalues()，如果要同时迭代key和value，可以用for k, v in d.iteritems()。

通过collections模块的Iterable类型判断一个对象是可迭代对象；

>>> from collections import Iterable

>>> isinstance('abc',Iterable)

True

enumerate函数可以把list变成索引-元素对，可在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

列表生成式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。举个例子，要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]可以用range(1, 11)；但如果要生成[1x1, 2x2, 3x3, ..., 10x10] 时用列表生成式[x \* x for x in range(1, 11)] 代替循环生成list。

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。for循环后面还可以加上if判断。

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

[4, 16, 36, 64, 100]

还可以使用两层循环，可以生成全排列：

>>> [m + n for m in 'ABO' for n in 'abo']

['Aa', 'Ab', 'Ao', 'Ba', 'Bb', 'Bo', 'Oa', 'Ob', 'Oo']

运用列表生成式，可以写出非常简洁的代码。例如，列出当前目录下的所有文件和目录名

>>> import os

>>> [d for d in os.listdir('.')]

['DLLs', 'Doc', 'include', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt', 'python.exe

', 'pythonw.exe', 'README.txt', 'Scripts', 'tcl', 'Tools', 'w9xpopen.exe']

>>>

>>> [isinstance(s, str) and s.lower() or s for s in L]

['hello', 18, 'world', None]

#### 生成器

通过列表生成式可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。如果列表元素可以按照某种算法推算出来，那就可以在循环的过程中不断推算出后续的元素呢，这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器（Generator）。

创建generator，第一种方法是把一个列表生成式的[]改成()，就创建了一个generator：

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

>>> g

<generator object <genexpr> at 0x02750670>

>>> g.next()

0

>>> g.next()

1

>>> g.next()

4

generator保存的是算法，每次调用next()，就计算出下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。创建了一个generator后，基本上永远不会调用next()方法，而是通过for循环来迭代它。

创建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：

斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：1, 1, 2, 3, 5, 8, 13, 21, 34, ...

>>> def fibonacci(max):

... n, a, b = 0, 0, 1

... while n < max:

... yield b

... a, b = b, a + b

... n = n + 1

...

>>> fibonacci(6)

<generator object fibonacci at 0x02750760>

generator的工作原理，它是在for循环的过程中不断计算出下一个元素，并在适当的条件结束for循环。对于函数改成的generator来说，遇到return语句或者执行到函数体最后一行语句，就是结束generator的指令，for循环随之结束。

### 函数式编程

函数是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

#### 高阶函数

高阶函数英文叫Higher-order function。abs(-10)是函数调用，而abs是函数本身。函数本身也可以赋值给变量，即：变量可以指向函数。函数名其实就是指向函数的变量！对于abs()这个函数，完全可以把函数名abs看成变量，它指向一个可以计算绝对值的函数！

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

>>> def add(x, y, f):

... return f(x) + f(y)

...

>>> add(5, -6, abs)

11

##### map

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

>>> def f(x):

... return x \* x

...

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

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

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：

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

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

##### reduce

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;

>>> def char2num(s):

return {'0':0,'1':1,'2':2,'3':3,'4':4,'5':5,'6':6,'7':7,'8':8,'9':9}[s]

...

>>> def str2int(s):

... return reduce(lambda x,y: x\*10+y, map(char2num,s))

...

>>> str2int('9527')

9527

##### filter

Python内建的filter()函数用于过滤序列。filter()也接收一个函数和一个序列。filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

>>> def is\_odd(n):

... return n % 2 == 1

...

>>> filter(is\_odd,[1,2,3,4,5,6,7,8,9])

[1, 3, 5, 7, 9]

##### Sorted

Python内置的sorted()函数就可以对list进行排序; sorted()函数也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。

>>> sorted([36,5,13,9,54])[::-1]

[54, 36, 13, 9, 5]

#### 返回函数

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

>>> def lazy\_sum(\*args):

... def sum():

... ax = 0

... for n in args:

... ax = ax + n

... return ax

... return sum

...

>>> f = lazy\_sum(1,3,5,7,9)

>>> f

<function sum at 0x0271C4F0>

>>> f()

25

调用lazy\_sum()时，返回的并不是求和结果，而是求和函数; 调用函数f时，才真正计算求和的结果。函数lazy\_sum中又定义了函数sum，并且，内部函数sum可以引用外部函数lazy\_sum的参数和局部变量，当lazy\_sum返回函数sum时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。调用lazy\_sum()时，每次调用都会返回一个新的函数，即使传入相同的参数。

>>> f1 = lazy\_sum(1,3,5,7,9)

>>> f2 = lazy\_sum(1,3,5,7,9)

>>> f1 == f2

False

f1()和f2()的调用结果互不影响。

注意到返回的函数在其定义内部引用了局部变量args，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用。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。

>>> count = [lambda x=y: x\*\*2 for y in range(3)]

>>> f1,f2,f3 = count

>>> f1(),f2(),f3()

(0, 1, 4)

#### 匿名函数

在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。在Python中，对匿名函数提供了有限支持。还是以map()函数为例，计算f(x)=x2时，除了定义一个f(x)的函数外，还可以直接传入匿名函数；关键字lambda表示匿名函数，冒号前面的x表示函数参数。匿名函数有个限制，就是只能有一个表达式，不用写return，返回值就是该表达式的结果。

>>> map(lambda x: x \* x, [1, 3, 5, 7, 9])

[1, 9, 25, 49, 81]

匿名函数不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数；同样，也可以把匿名函数作为返回值返回。

#### 装饰器

函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。函数对象有一个\_\_name\_\_属性，可以拿到函数的名字；

>>> def name():

... print 'Hello World!'

...

>>> f = name

>>> f()

Hello World!

>>> name.\_\_name\_\_

'name'

>>> f.\_\_name\_\_

'name'

在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。本质上，decorator就是一个返回函数的高阶函数。

>>> def log(func):

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

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

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

... return wrapper

...

Python的@语法，把decorator置于函数的定义处

>>> @log

... def now():

... print '2017-07-26'

...

>>> now()

call now():

2017-07-26

把@log放到now()函数的定义处，相当于执行了语句now = log(now)

Python内置的functools.wraps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。在定义wrapper()的前面加上@functools.wraps(func)即可。

一个完整的decorator的写法

>>> import functools

>>> def log(func):

... @functools.wraps(func)

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

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

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

... return wrapper

...

>>> f = log(now)

>>> f()

call wrapper():

call now():

2017-07-26

带参数的decorator：

>>> import functools

>>> def log(text):

... def decorator(func):

... @functools.wraps(func)

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

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

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

... return wrapper

... return decorator

...

>>> f = log('parameter')(now)

>>> f()

parameter wrapper():

call now():

2017-07-26

在面向对象（OOP）的设计模式中，decorator被称为装饰模式。OOP的装饰模式需要通过继承和组合来实现，而Python除了能支持OOP的decorator外，直接从语法层次支持decorator。Python的decorator可以用函数实现，也可以用类实现。

#### 偏函数

Python的functools模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。可以降低函数调用的难度。当函数的参数个数太多，需要简化时，使用functools.partial可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。

>>> import functools

>>> int2 = functools.partial(int,base=2)

>>> int2('10')

2

### 模块

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Python中，一个.py文件就称之为一个模块（Module）。使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。Python又引入了按目录来组织模块的方法，称为包（Package）。引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。每一个包目录下面都会有一个\_\_init\_\_.py的文件，这个文件是必须存在的，否则，Python就把这个目录当成普通目录，而不是一个包。\_\_init\_\_.py可以是空文件，也可以有Python代码，因为\_\_init\_\_.py本身就是一个模块。

#### 使用模块

##### 别名

导入模块时，还可以使用别名，这样，可以在运行时根据当前环境选择最合适的模块。比如Python标准库一般会提供StringIO和cStringIO两个库，这两个库的接口和功能是一样的，但是cStringIO是C写的，速度更快，

>>> try:

... import cStringIO as StringIO

... except ImportError:

... import StringIO

...

这样就可以优先导入cStringIO。如果有些平台不提供cStringIO，还可以降级使用StringIO。导入cStringIO时，用import ... as ...指定了别名StringIO，因此，后续代码引用StringIO即可正常工作。由于Python是动态语言，函数签名一致接口就一样，因此，无论导入哪个模块后续代码都能正常工作。

##### 作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过\_前缀来实现的。正常的函数和变量名是公开的（public），可以被直接引用，比如：abc，x123，PI等；类似\_\_xxx\_\_这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的\_\_author\_\_，\_\_name\_\_就是特殊变量，类似\_xxx和\_\_xxx这样的函数或变量就是非公开的（private），不应该被直接引用，比如\_abc，\_\_abc等；外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

##### 安装第三方模块

win7 64位无法安装PIL的同学可以上<http://www.pythonware.com/products/pil/> 这个网址下载安装包到本地C:\Python27\Lib\site-packages下，然后运行应用程序，手动安装就可以啦~

C:\Users\CM>pip install PIL

Collecting PIL

Could not find a version that satisfies the requirement PIL (from versions: )

No matching distribution found for PIL

>>> import Image

>>> im = Image.open('123.jpg')

>>> print im.format, im.size, im.mode

JPEG (3744, 5616) RGB

>>> im.thumbnail((300,500))

>>> im.save('456.jpg','JPEG')

##### \_\_future\_\_模块

Python的每个新版本都会增加一些新的功能，或者对原来的功能作一些改动。有些改动是不兼容旧版本的，也就是在当前版本运行正常的代码，到下一个版本运行就可能不正常了。从Python 2.7到Python 3.x就有不兼容的一些改动，比如2.x里的字符串用'xxx'表示str，Unicode字符串用u'xxx'表示unicode，而在3.x中，所有字符串都被视为unicode，因此，写u'xxx'和'xxx'是完全一致的，而在2.x中以'xxx'表示的str就必须写成b'xxx'，以此表示“二进制字符串”。 要直接把代码升级到3.x是比较冒进的，因为有大量的改动需要测试。相反，可以在2.7版本中先在一部分代码中测试一些3.x的特性，如果没有问题，再移植到3.x不迟。Python提供了\_\_future\_\_模块，把下一个新版本的特性导入到当前版本，于是我们就可以在当前版本中测试一些新版本的特性。

为了适应Python 3.x的新的字符串的表示方法，在2.7版本的代码中，可以通过unicode\_literals来使用Python 3.x的新的语法：

>>> from \_\_future\_\_ import unicode\_literals

>>> print '\'xxx\' is unicode?',isinstance('xxx', unicode)

'xxx' is unicode? True

>>> print 'u\'xxx\' is unicode?', isinstance(u'xxx', unicode)

u'xxx' is unicode? True

>>> print '\'xxx\' is str?', isinstance('xxx', str)

'xxx' is str? False

>>> print 'b\'xxx\' is str?', isinstance(b'xxx', str)

b'xxx' is str? True

上面的代码仍然在Python 2.7下运行，但结果显示去掉前缀u的'a string'仍是一个unicode，而加上前缀b的b'a string'才变成了str；

在Python 2.x中，对于除法有两种情况，如果是整数相除，结果仍是整数，余数会被扔掉，这种除法叫“地板除”； 要做精确除法，必须把其中一个数变成浮点数；而在Python 3.x中，所有的除法都是精确除法，地板除用//表示；在Python 2.7的代码中直接使用Python 3.x的除法，可以通过\_\_future\_\_模块的division实现；

>>> 10 / 3

3

>>> from \_\_future\_\_ import division

>>> 10 / 3

3.3333333333333335

>>> 10 // 3

3

### 面向对象编程

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

>>> std1 = {'name':'Tom','score':90}

>>> std2 = {'name':'Jerry','score':99}

>>> def print\_score(std):

... print '%s: %s' % (std['name'],std['score'])

...

>>> print\_score(std1)

Tom: 90

>>> print\_score(std2)

Jerry: 99

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。在Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

>>> class Student(object):

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

... self.name = name

... self.score = score

... def print\_score(self):

... print '%s: %s' % (self.name, self.score)

...

>>> bart = Student('Bart Simpson', 59)

>>> lisa = Student('Lisa Simpson', 95)

>>> bart.print\_score()

>>> lisa.print\_score()

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义的Class——Student，是指学生这个概念，而实例（Instance）则是一个个具体的Student，比如，Bart Simpson和Lisa Simpson是两个具体的Student：所以，面向对象的设计思想是抽象出Class，根据Class创建Instance。面向对象的抽象程度又比函数要高，因为一个Class既包含数据，又包含操作数据的方法。

#### 类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，比如Student类，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。在Python中，定义类是通过class关键字：

>>> 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(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)。定义好了Student类，就可以根据Student类创建出Student的实例，创建实例是通过类名+()实现的：

>>> bart = Student()

>>> bart

<\_\_main\_\_.Student object at 0x02820510>

>>> Student

<class '\_\_main\_\_.Student'>

变量bart指向的就是一个Student的object，后面的0x10a67a590是内存地址，每个object的地址都不一样，而Student本身则是一个类。可以自由地给一个实例变量绑定属性，比如，给实例bart绑定一个name属性：

>>> bart.name = 'Tom'

>>> bart.score = 99

>>> bart.name

u'Tom'

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的\_\_init\_\_方法，在创建实例的时候，就把name，score等属性绑上去。\_\_init\_\_方法的第一个参数永远是self，表示创建的实例本身，因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。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。

>>> class Student(object):

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

... self.name = name

... self.score = score

...

>>> bart = Student('Tom', 59)

>>> bart

<\_\_main\_\_.Student object at 0x028297F0>

>>> bart.name

u'Tom'

>>> bart.score

59

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量self，并且，调用时，不用传递该参数。

数据封装

面向对象编程的一个重要特点就是数据封装。

#### 访问限制

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，如果需要访问需要加get方法，要修改的话需要set方法；

>>> class Student(object):

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

... self.\_\_name = name

... self.\_\_score = score

... def print\_score(self):

... print '%s: %s' % (self.\_\_name, self.\_\_score)

... def get\_name(self):

... return self.\_\_name

... def set\_score(self):

... self.\_\_score = score

...

>>> bart = Student('Jerry',99)

>>> bart.\_\_name

Traceback (most recent call last):

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

AttributeError: 'Student' object has no attribute '\_\_name'

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。

以一个下划线开头的实例变量名，比如\_name，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。 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

u'Jerry'

>>>

#### 继承和多态

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

>>> class Animal(object):

... def run(self):

... print 'Animal is running!'

...

>>> class Dog(Animal):

... def run(self):

... print 'Dog is running!'

...

>>> dog = Dog()

>>> dog.run()

Dog is running!

当子类和父类都存在相同的方法时，子类的方法覆盖了父类的方法，在代码运行的时候，总是会调用子类的方法。这就是继承的另一个好处：多态。

在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行。

“开闭”原则：对扩展开放：允许新增Animal子类；对修改封闭：不需要修改依赖Animal类型的run\_twice()等函数。

#### 获取对象信息

拿到一个对象的引用时，使用type()函数判断对象类型；函数如果一个变量指向函数或者类，也可以用type()判断：

>>> type(dog)

<class '\_\_main\_\_.Dog'>

>>> type('123')

<type 'unicode'>

Python把每种type类型都定义好了常量，放在types模块里，使用之前，需要先导入：

>>> import types

>>> types.StringType

<type 'str'>

>>> types.TypeType

<type 'type'>

>>> type(str)

<type 'type'>

有一种类型就叫TypeType，所有类型本身的类型就是TypeType

判断class的类型，可以使用isinstance()函数。isinstance()判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list

>>> dir(dog)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribut

e\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_module\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_e

x\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_

\_weakref\_\_', 'run']

>>>

类似\_\_xxx\_\_的属性和方法在Python中都是有特殊用途的，比如\_\_len\_\_方法返回长度。在Python中，如果你调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的\_\_len\_\_()方法；

配合getattr()、setattr()以及hasattr()，可以直接操作一个对象的状态：

### 面向对象高级编程

数据封装、继承和多态只是面向对象程序设计中最基础的3个概念。在Python中，面向对象还有很多高级特性，允许我们写出非常强大的功能。

#### 使用\_\_slots\_\_

Python允许在定义class的时候，定义一个特殊的\_\_slots\_\_变量，来限制该class能添加的属性：

>>> class Coder(object):

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

...

>>> c = Coder()

>>> c.subject = 'Java'

>>> c.age = 21

>>> c.salary = 30000

Traceback (most recent call last):

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

AttributeError: 'Coder' object has no attribute 'salary'

使用\_\_slots\_\_要注意，\_\_slots\_\_定义的属性仅对当前类起作用，对继承的子类是不起作用的；除非在子类中也定义\_\_slots\_\_，这样，子类允许定义的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_。

#### 使用@property

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

#### 多重继承

Mixin的目的就是给一个类增加多个功能，这样，在设计类的时候，我们优先考虑通过多重继承来组合多个Mixin的功能，而不是设计多层次的复杂的继承关系。由于Python允许使用多重继承，因此，Mixin就是一种常见的设计。先继承第一个类，如果有相应的方法则执行，否则向上查找，直到object类；如果还是没有则查找第二个类，如果有相应的方法则执行，否则向上查找，直到object类；

python 支持多继承，但对与经典类和新式类来说，多继承查找的顺序是不一样的。Python 2.x中默认都是经典类，只有显式继承了object才是新式类。Python 3.x中默认都是新式类，不必显式的继承object。

经典类的搜索方式是按照“**从左至右，深度优先**”的方式去查找属性。

>>> class P1:

... def foo(self):

... print 'p1-foo'

...

>>> class P2:

... def foo(self):

... print 'p2-foo'

... def bar(self):

... print 'p2-bar'

...

>>> class C1(P1,P2):

... pass

...

>>> class C2(P1,P2):

... def bar(self):

... print 'C2-bar'

...

>>> class D(C1,C2):

... pass

...

>>> d = D()

>>> d.foo()

p1-foo

>>> d.bar()

p2-bar

>>>

实例d调用foo()时，搜索顺序是 D => C1 => P1

实例d调用bar()时，搜索顺序是 D => C1 => P1 => P2

d先查找自身是否有foo方法，没有则查找最近的父类C1里是否有该方法，如果没有则继续向上查找，直到在P1中找到该方法，查找结束。

新式类的搜索方式是采用“**广度优先**”的方式去查找属性：

>>> class P1(object):

... def foo(self):

... print 'p1-foo'

...

>>> class P2(object):

... def foo(self):

... print 'p2-foo'

... def bar(self):

... print 'p2-bar'

...

>>> class C1(P1,P2):

... pass

...

>>> class C2(P1,P2):

... def bar(self):

... print 'C2-bar'

...

>>> class D1(C1,C2):

... pass

...

>>> class D2(C2,C1):

... pass

...

>>> d1 = D1()

>>> d2 = D2()

>>>

>>> d1.foo()

p1-foo

>>> d2.foo()

p1-foo

>>> d1.bar()

C2-bar

>>> d2.bar()

C2-bar

实例d1调用foo()时，搜索顺序是 D1 => C1 => C2 => P1

实例d1调用bar()时，搜索顺序是 D1 => C1 => C2

#### 定制类

##### \_\_str\_\_

类中定义\_\_str\_\_函数可以用来打印信息；直接显示变量调用的不是\_\_str\_\_()，而是\_\_repr\_\_()，两者的区别是\_\_str\_\_()返回用户看到的字符串，而\_\_repr\_\_()返回程序开发者看到的字符串，也就是说，\_\_repr\_\_()是为调试服务的。

>>> class Subject(object):

... def \_\_init\_\_(self, subject):

... self.subject = subject

... def \_\_str\_\_(self):

... return 'The most pop language is %s' % self.subject

... \_\_repr\_\_ = \_\_str\_\_

...

>>> s = Subject('Java')

>>> s

The most pop language is Java

##### \_\_iter\_\_

如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的next()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

>>> class Fib(object):

... def \_\_init\_\_(self):

... self.a, self.b = 0, 1

... def \_\_iter\_\_(self):

... return self

... def next(self):

... self.a, self.b = self.b, self.a + self.b

... if self.a > 100000:

... raise StopIteration;

... return self.a

...

>>> for n in Fib():

... if n < 20:

... print n

...

1

1

2

3

5

8

13

##### \_\_getitem\_\_

要表现得像list那样按照下标取出元素，需要实现\_\_getitem\_\_()方法：

\_\_getitem\_\_>>> class Fib(object):

... def \_\_getitem\_\_(self, n):

... a, b = 1, 1

... for x in range(n):

... a, b = b, a + b

... return a

...

>>> f = Fib()

>>> f[0]

1

>>> f[3]

3

>>> f[6]

13

与之对应的是\_\_setitem\_\_()方法，把对象视作list或dict来对集合赋值。最后，还有一个\_\_delitem\_\_()方法，用于删除某个元素。

##### \_\_getattr\_\_

只有在没有找到属性的情况下，才调用\_\_getattr\_\_，已有的属性，比如name，不会在\_\_getattr\_\_中查找。

##### \_\_call\_\_

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用instance.method()来调用。任何类，只需要定义一个\_\_call\_\_()方法，就可以直接对实例进行调用。判断一个对象是否能被调用，能被调用的对象就是一个Callable对象，通过callable()函数，我们就可以判断一个对象是否是“可调用”对象。

#### 使用元类

##### type()

type()函数可以查看一个类型或变量的类型，Hello是一个class，它的类型就是type，而h是一个实例，它的类型就是class Hello。我们说class的定义是运行时动态创建的，而创建class的方法就是使用type()函数。

**class Hello(object):**

**def** **hello**(self, name='world'):

print('Hello, %s.' % name)

>>> from hello import Hello

>>> h = Hello()

>>> h.hello()

Hello, world.

>>> print(type(Hello))

<type 'type'>

>>> print(type(h))

<class 'hello.Hello'>

### 错误、调试和测试

##### 错误

代码可能会出错时用try来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即except语句块，执行完except后，如果有finally语句块，则执行finally语句块。

**try**:

**print** 'try...'

r = 10 / 0

**print** 'result:', r

**except** ZeroDivisionError, e:

**print** 'except:', e

**finally**:

**print** 'finally...'

**print** 'END'

**try**...

**except**: integer division **or** modulo by zero

**finally**...

END

Python所有的错误都是从BaseException类派生的，常见的错误类型和继承关系看这里：

<https://docs.python.org/2/library/exceptions.html#exception-hierarchy>

使用try...except捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数main()调用foo()，foo()调用bar()，结果bar()出错了，这时，只要main()捕获到了，就可以处理：也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写try...except...finally的麻烦。

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。

>>> def foo(s):

... return 10 / int(s)

...

>>> def bar(s):

... return foo(s) \* 2

...

>>> def main():

... bar('0')

...

>>> main()

Traceback (most recent call last):

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

File "<stdin>", line 2, in main

File "<stdin>", line 2, in bar

File "<stdin>", line 2, in foo

ZeroDivisionError: integer division or modulo by zero

因为错误是class，捕获一个错误就是捕获到该class的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，自己编写的函数也可以抛出错误。

#### 调试

程序能一次写完并正常运行的概率很小，基本不超过1%。总会有各种各样的bug需要修正。因此，需要一整套调试程序的手段来修复bug。

第一种方法就是用print把可能有问题的变量打印出来看看；用print最大的坏处是将来还得删掉它，想想程序里到处都是print，运行结果也会包含很多垃圾信息。

第二种方法是使用断言（assert），如果断言失败，assert语句本身就会抛出AssertionError；启动Python解释器时可以用-O参数来关闭assert，关闭后，你可以把所有的assert语句当成pass来看。

第三种方式是logging，logging不会抛出错误，而且可以输出到文件；允许指定记录信息的级别，有debug，info，warning，error等几个级别，当指定level=INFO时，logging.debug就不起作用了。同理指定level=WARNING后，debug和info就不起作用了。这样可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

import logging

logging.basicConfig(level=logging.INFO)

第四种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。

#### 单元测试

#### 文档测试

### 文件读写

#### 读文件

Python内置的open()函数可以以读文件的模式打开一个文件对象

#标示符'r'表示读

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

#read()方法可一次读取文件全部内容，Python把内容读到内存，用一个str对象表示

>>> f.read()

'this is a python demo!'

#close()方法关闭文件。文件使用完后必须关闭，因为文件对象会占用操作系统资源，并且操作系统同一时间能打开的文件数量也是有限的

>>> f.close()

#如果文件不存在，open()函数会抛出一个IOError的错误,且指出不存在的文件位置;

>>> f = open('/Users/cm/python/test2.txt', 'r')

Traceback (most recent call last):

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

IOError: [Errno 2] No such file or directory: '/Users/cm/python/test2.txt'

>>>

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，可以使用try ... finally来实现：

>>> try:

... f = open('/Users/cm/python/test.txt', 'r')

... print f.read()

... finally:

... if f:

... f.close()

...

也可以使用with语句来自动调用close()方法：

>>> with open('/Users/cm/python/test.txt', 'r') as f:

... print f.read()

...

this is a python demo!

调用read()会一次性读取文件的全部内容，如果文件有10G，内存就溢出了，所以保险起见，可以反复调用read(size)方法，每次最多读取size个字节的内容。另外，调用readline()可以每次读取一行内容，调用readlines()一次读取所有内容并按行返回list。因此，要根据需要决定怎么调用。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()最方便。

像open()函数返回的这种有个read()方法的对象，在Python中统称为file-like Object。除了file外，还可以是内存的字节流，网络流，自定义流等等。file-like Object不要求从特定类继承，只要写个read()方法就行。StringIO就是在内存中创建的file-like Object，常用作临时缓冲。

要读取二进制文件，比如图片、视频等等，用'rb'模式打开文件即可

>>> p = open('/Users/cm/python/123.jpg', 'rb')

>>> p.read()

#### 写文件

写文件和读文件是一样的，唯一区别是调用open()函数时，传入标识符'w'或者'wb'表示写文本文件或写二进制文件：