1. list列表：[ ]

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。

列出班里所有同学的名字，就可以用一个list表示：

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

>>> classmates

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

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。

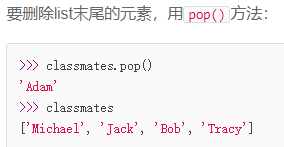
（1）



（2）



（3）



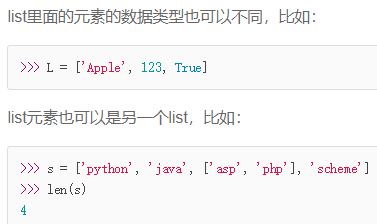
（4）

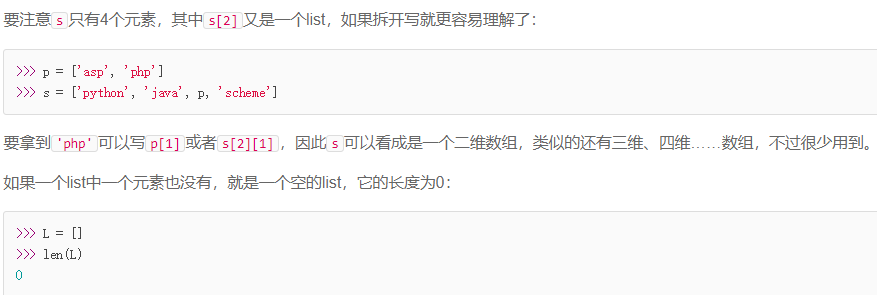


（5）



（6）





1. tuple元组：( )

（1）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：

>>> 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。

（2） 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。 注意：tuple的陷阱：当你去 **定义一个tuple时，在定义的时候，tuple的元素就必须被确定下来**，比如：

>>> t = (1, 2)

>>> t

(1, 2)

（3）如果要定义一个空的tuple，可以写成()：

>>> t = ()

>>> t

()

（3）但是，要定义一个只有1个元素的tuple，如果你这么定义：

>>> t = (1)

>>> t

1

定义的不是tuple，是1这个数！这是因为括号()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，计算结果自然是1。

所以，**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,)

如上图最后一行所示： Python在显示只有1个元素的tuple时，也会加一个逗号, 以免你误解成数学计算意义上的括号。

（4） 最后来看一个“可变的”tuple：

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

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

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

>>> t

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

这个tuple定义的时候有3个元素，分别是'a'，'b'和一个list。不是说tuple一旦定义后就不可变了吗？怎么后来又变了？

表面看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**。tuple一开始指向的list并没有改成别的list，所以，tuple所谓的“不变”是说：

**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。 注意：**list和tuple是Python内置的有序集合**，一个可变，一个不可变。根据需要来选择使用它们。

（5） 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 。

1. dict字典：{ }

（1）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。

假设根据同学的名字查找对应的成绩，用dict实现： 仅需一个 “名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。

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

>>> d['Michael']

95

dict实现方式：给定一个名字，比如'Michael'，dict在内部就可以**直接计算出**Michael对应的存放成绩的“页码”，也就是95这个成绩数字所 **存放的内存地址**，便可直接取出来，所以速度非常快。 此key-value存储方式，在放进去的时候 **dict必须根据key来算出value的存放位置的**，这样，取的时候才能根据key直接拿到value。

（2） 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

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

>>> 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'

（3）要避免key不存在的错误，有两种办法：

一、通过**in**成员运算符：来判断key是否存在：

>>> 'Thomas' in d

False

二、 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的交互环境不显示结果。

（4） 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}

（5）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的增加而变慢；

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内部就会混乱了，这个通过key计算位置的算法称为哈希算法。 注意：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，一定不能作为key**：

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

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

Traceback (most recent call last):

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

TypeError: unhashable type: 'list'

1. set：{ }

（1）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，即无论你准备把任何东西装入set：都需要提供 一个list形式 来作为输入集合，甚至是空集，也得是 [ ] 形式。**

>>> 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。

**注意：重复元素在set中自动被过滤！！！**

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

>>> s

{1, 2, 3}

（2）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

{1, 2, 3, 4}

>>> s.add(4)

>>> s

{1, 2, 3, 4}

（3）通过 **remove(key)**方法可以删除元素：

>>> s.remove(4)

>>> s

{1, 2, 3}

（4） 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

{2, 3}

>>> s1 | s2

{1, 2, 3, 4}

**set和dict的唯一区别：仅在于set没有存储对应的value**， 但是set的原理和dict一样，所以set也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，故也就无法保证set内部“不会有重复元素”。 (即把list放入set会报错)

（5） 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**。 tuple虽然是不变对象，但试试把 (1, 2, 3) 和(1, [2, 3]) 放入dict或set中，并解释结果。

今天我们大多数时候使用的计算机，虽然它们的元器件做工越来越精密，处理能力越来越强大，但究其本质来说仍然属于“冯·诺依曼结构”的计算机。“冯·诺依曼结构”有两个关键点： 一是指出要将存储设备与中央处理器分开、 二是提出了将数据以二进制方式编码。 二进制是一种“逢二进一”的计数法，对于计算机来说，二进制在物理器件上来说是最容易实现的（高电压表示1，低电压表示0），于是在“冯·诺依曼结构”的计算机都使用了二进制。

1. 不可变对象 （如str字符串是不变对象，而list是可变对象）

（1）对于可变对象，比如list，对list进行操作，则list内部的内容是会变化的，如

>>> a = ['c', 'b', 'a']

>>> a.sort()

>>> a

['a', 'b', 'c']

（2）而对于不可变对象，比如str，对str进行操作呢：

>>> a = 'abc'

>>> a.replace('a', 'A')

'Abc'

>>> a

'abc'

虽然字符串有个replace()方法，也确实变出了'Abc'，但变量a最后仍是'abc'，应该怎么理解呢？

我们先把代码改成下面这样：

>>> a = 'abc'

>>> b = a.replace('a', 'A')

>>> b

'Abc'

>>> a

'abc'

要始终牢记的是，a是变量，而'abc'才是字符串对象！ 即a本身是一个变量，它指向的对象的内容才是'abc'：

┌───┐ ┌───────┐

│ a │─────────────────>│ 'abc' │

└───┘ └───────┘

当我们调用a.replace('a', 'A')时，实际上调用方法**replace是作用在字符串对象'abc'上的**， 而这个方法虽然名字叫replace，但却没有改变字符串'abc'的内容。 相反，replace方法创建了一个新字符串'Abc'并返回， 如果我们用变量b指向该新字符串，就容易理解了，变量a仍指向原有的字符串'abc'，但变量b却指向新字符串'Abc'了：

┌───┐ ┌───────┐

│ a │─────────────────>│ 'abc' │

└───┘ └───────┘

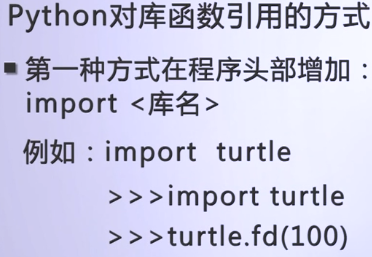
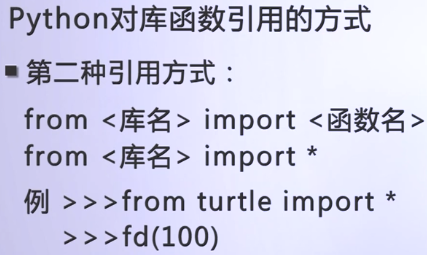
┌───┐ ┌───────┐

│ b │─────────────────>│ 'Abc' │

└───┘ └───────┘

重点！！！ **故对于不变对象来说，调用对象自身的任意方法也不会改变该对象自身的内容，相反，这些方法会创建新的对象并返回**， 这样就保证了不可变对象本身 永远是 不可变的。

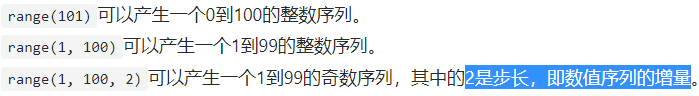
6.

注意：浮点数是用计算机专有的浮点运算单元来计算，相比整数更加耗时； 故为兼顾效率，会把整数、浮点数区分开来。

7. for in 循环





在很多编程语言中，针对字符串提供了很多各种截取函数（例如，substring），其实目的就是对字符串切片。 Python没有针对字符串的截取函数，只需要切片运算符就可以完成，非常简单。

8. **\_\_name\_\_** 是属于 python 中的内置类属性，即其会天生就存在于一个 python 程序中，**代表对应程序名称**。 一段 **程序作为主线运行程序时其内置名称就是 \_\_main\_\_**



9. 下标运算符 、 切片运算符：



str2 = 'abc123456'

（1）从字符串中取出指定位置的字符(下标运算)

print(str2[2]) # c

（2）字符串切片(从指定的开始索引到指定的结束索引)

print(str2[2:5]) # c12

print(str2[2:]) # c123456

print(str2[2::2]) # c246

print(str2[::2]) # ac246

print(str2[::-1]) # 654321cba

print(str2[-3:-1]) # 45

10. 函数 和 模块(来管理函数)

（1）多人协作进行团队开发的时候，团队中可能有多个程序员都定义了名为foo的函数，那么怎么解决这种命名冲突呢？**Python中每个文件就代表了一个模块（module）**，故我们可在不同的模块中有同名的函数，在使用函数的时候我们通过 **import关键字导入指定的模块** 就可以区分到底要使用的是哪个模块中的foo函数，代码如下所示：

(注意：module1、module2是文件名，是不含 .py 的扩展名)

1）module1.py

def foo():

print('hello, world!')

2）module2.py

def foo():

print('goodbye, world!')

3）test.py

**from module1 import foo**

# 输出hello, world!

foo()

**from module2 import foo**

# 输出goodbye, world!

foo()

（2）也可以按照如下所示的方式来区分到底要使用哪一个foo函数。

test.py

**import module1 as m1**

**import module2 as m2**

m1.foo()

m2.foo()

但是如果将代码写成了下面的样子，那么程序中调用的是最后导入的那个foo，因为后导入的foo覆盖了之前导入的foo。

1）test.py

from module1 import foo

from module2 import foo

# 输出goodbye, world!

foo()

2）test.py

from module2 import foo

from module1 import foo

# 输出hello, world!

foo()

### （3）空函数

1）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，那有什么用？ 实际上**pass可以用来作为占位符**，比如现在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。

2） pass还可以用在其他语句里，比如：

if age >= 18:

pass

缺少了pass，代码运行就会有语法错误。

（4） 注意python可以return返回多个值（区别于C语言）：其实返回多个值时，会返回一个tuple。

（5） 函数默认参数： python的函数可以设置默认参数：能降低调用函数的难度。



1) 可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

2) 有多个默认参数时，调用的时候，可以按顺序提供默认参数： 比如调用enroll('Bob', 'M', 7)， 指除了name、gender 两个参数外， 最后1个参数应用在参数age上， 而 city 参数由于没有提供，仍然使用默认值。

3) 也可以不按顺序来提供部分默认参数， **当不按顺序提供部分默认参数时，需要把参数名写上**： 比如调用enroll('Adam', 'M', city='Tianjin')，意思是， city 参数用传进去的值，其他默认参数继续使用默认值。

4） 定义默认参数要牢记一点：**默认参数必须指向不变对象**！！！

为什么要设计**str、None、整数 这样的不变对象**呢？ 因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。 此外，由于对象不变，多任务环境下同时读取对象就可以不需要加锁了，这样同时读一点问题都没有。 故我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

（6） 函数可变参数：

1）我们把函数的参数改为可变参数：

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

sum = 0

for n in numbers:

sum = sum + n \* n

return sum

**定义可变参数时仅仅在参数前面加了一个\*号**，即表示numbers是可变参数。 **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

2） 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

\*nums 表示： 把nums 这个list的所有元素，作为可变参数传进去（即表示nums是一个可变参数）。

（7）关键字参数

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；

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**。 示例如下图：



1） 关键字参数有什么用？ 它可以**扩展函数的功能**。 比如，在person函数里， “我们保证能接收到name和age这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到”。 试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

2） 与可变参数类似，也可先组装出一个dict，再把该dict转换为关键字参数传进去：

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

>>> person('Jack', 24, city=extra['city'], job=extra['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：

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

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

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

**\*\*extra**表示： 把extra这个dict的所有key-value用关键字参数传入到函数的\*\*kw参数，kw将获得一个dict，注意**kw获得的dict是extra的一份拷贝，对kw的改动不会影响到函数外的extra**。

（8） 命名关键字参数：

1） 对于关键字参数， 函数的调用者可以传入任意不受限制的关键字参数。 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

2）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)

3） 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，'Beijing' 和 'Engineer' 缺少写上参数名city 和 job ， 则Python解释器会把这4个参数 均视为位置参数， 但person()函数仅接受2个位置参数。

4） 使用命名关键字参数时，要特别注意，如果没有可变参数，就必须加一个 \* 来作为特殊分隔符。 如果缺少 \* ， 则Python解释器将无法识别位置参数和命名关键字参数：

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

*# 缺少 \*，city和job被视为位置参数*

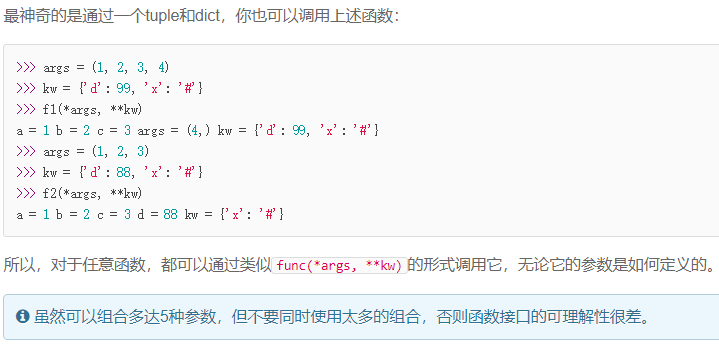
**pass**

6）在Python中定义函数，可以用必选参数、 默认参数、 可变参数、关键字参数 和命名关键字参数，这5种参数都可以组合使用。 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**、 **命名关键字参数** 和 **关键字参数** 。



注意： 1） 函数f2中的‘ \* ’是起分隔符作用， 指后面的都是命名关键字参数；

2） 命名关键字参数： 必须要传入参数名 （如x=99; ext=Nome）。



7） 结论：

默认参数一定要用不可变对象，如果是可变对象，程序运行时会有逻辑错误！！！

\*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；

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 。

命名关键字参数：为限制调用者可传入的参数名，同时可提供默认值。 定义命名关键字参数时，在没有可变参数的情况下不要忘了写分隔符\* ，否则定义的将是位置参数。

11. 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，如果编译器或者解释器对尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，则就不会出现栈溢出的情况。

针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。 但是Python标准的解释器并没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

12.

（1）Python的for循环抽象程度要高于C的for循环， 因为Python的for循环不仅可以用在list或tuple上，还可以作用在其他可迭代对象上。

list这种数据类型虽然有下标，但很多其他数据类型是没有下标的， 但是, 只要是可迭代对象Iterable，无论有无下标，都可以迭代，比如dict就可以迭代：

>>> 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.values()** ； 如果要同时迭代key和value，可以用**for k, v in d.items()** 。

由于字符串也是可迭代对象，因此，也可以作用于for循环：

>>> **for** ch **in** 'ABC':

... print(ch)

...

A

B

C

（2）所以，当我们使用for循环时，只要作用于一个可迭代对象，for循环就可以正常运行，而我们不太关心该对象究竟是list还是其他数据类型。

注意：字符串也是可迭代对象，也可以作用于for循环。

（3） for循环其实可以同时使用两个甚至多个变量， 比如对于dict 的 items()方法，可以同时迭代key和value：

>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }

>>> for k, v in d.items():

... print(k, '=', v)

...

y = B

x = A

z = C

（4）可以直接作用于for循环的数据类型有以下几种：（重点）

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** 。

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

（5） 生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator迭代器。

因为Python的**Iterator对象**表示的是一个数据流，Iterator对象 **可被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**。

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

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

True

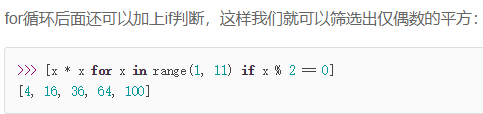
>>> isinstance(iter('abc'), Iterator)

True

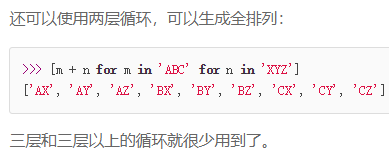
13. 列表生成式： Python内置的可以用来 **创建list的生成式**。

（1）

（2）



（3）



14. 生成器： 通过列表生成式，我们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。 而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。 所以，如果列表元素可以按照某种算法推算出来，那我们是否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**。

（1）要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

>>> 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。**

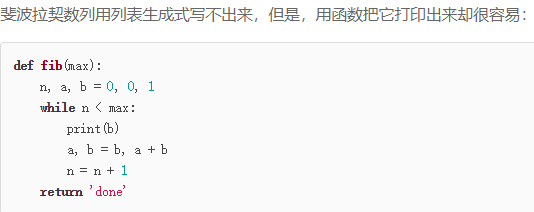
我们可以直接打印出list的每一个元素，但我们怎么才能打印出generator的每一个元素呢？ 如果要一个个打印出来，可通过**next()函数** 获得generator的下一个返回值。 但是上面这种不断调用next(g) 实在是太变态了，正确的方法是使用 **for循环** ，因为generator也是可迭代对象：

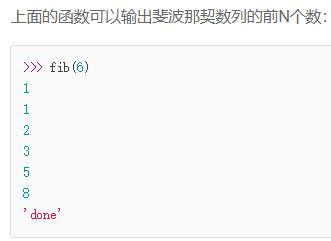
>>> g = (x \* x **for** x **in** range(10))

>>> **for** n **in** g:

... print(n)

（2） yield关键字





可以看出，fib函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。 也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator，只需要把 print(b) 改为yield b 就可以了：

def fib(max):

n, a, b = 0, 0, 1

while n < max:

**yield b**

a, b = b, a + b

n = n + 1

return 'done'

这是定义generator的另一种方法： 若 **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>

（3） 这里，最难理解的就是generator和函数的执行流程不一样。 函数是顺序执行，遇到return 语句 或者 最后一行函数语句 就返回。 而变成 **generator的函数，在每次调用next()时才执行，遇到yield语句时函数返回**， **再次执行时要从上次返回的yield语句处继续执行**。

举个简单的例子，定义一个generator，依次返回数字1，3，5：

def odd():

print('step 1')

yield 1

print('step 2')

yield(3)

print('step 3')

yield(5)

在调用该generator时，首先要生成一个generator对象， 然后用next()函数不断获得下一个返回值：

>>> o = odd()

>>> next(o)

step 1

1

>>> next(o)

step 2

3

>>> next(o)

step 3

5

>>> next(o)

Traceback (most recent call last):

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

StopIteration

可以看到，odd不是普通函数，而是generator，在执行过程中，**generator遇到yield就中断**， 下次又继续执行。执行3次yield后，已经没有yield可以执行了，所以，第4次调用next(o)就报错。

（4） 回到上面 fib 的例子，我们在循环过程中不断调用yield，就会不断地中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代生成器： （注意：少打印了一个done）

>>> for n in fib(6):

... print(n)

...

1

1

2

3

5

8

在用for循环调用generator时，发现拿不到generator的 return 语句的返回值。 如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration 的 value中：

>>> g = fib(6)

>>> while True:

... try:

... x = next(g)

... print('g:', x)

... except StopIteration as e:

... print('Generator return value:', e.value)

... break

...

g: 1

g: 1

g: 2

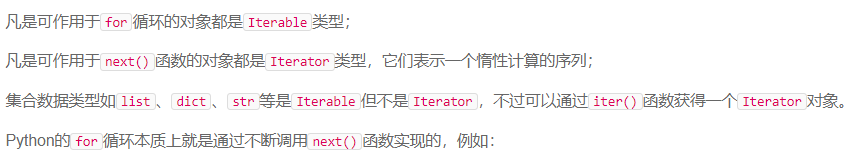
g: 3

g: 5

g: 8

Generator return value: done

15. 迭代器 Iterator



16. 函数式编程：

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。

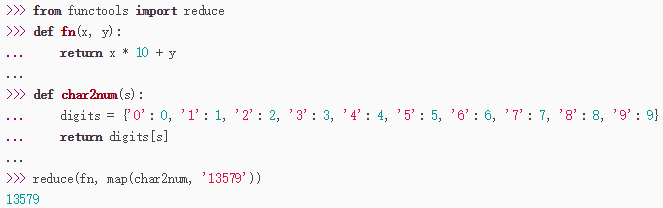
注意：函数名也是变量，是指向函数的变量。 既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可接收另一个函数作为参数，这种函数称为 **高阶函数**。编写高阶函数，就是让函数的参数能够接收别的函数，把函数作为参数传入， 函数式编程即指这种高度抽象的编程范式。

（1）高阶函数map ：接收2个参数，一个是函数、一个是Iterable可迭代对象 , map将传入的函数依次作用到序列中的每一个元素，并把结果作为 新的Iterator 迭代器返回。

（2）高阶函数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)

（3）字符串str也是一个序列，配合map、reduce，把str转换为int函数：





（4） Python内建的filter()函数用于过滤序列。 和map()类似， **filter()也接收一个函数和一个序列**。 和map()不同的是，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。

例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

def is\_odd(n):

return n % 2 == 1

list(filter(is\_odd, [1, 2, 4, 5, 6, 9, 10, 15]))

*# 结果: [1, 5, 9, 15]*

注意filter()函数返回的是 Iterator，也就是一个惰性序列，所以要强迫filter()完成计算结果，需要用 **list()函数把整个惰性序列都计算出来并返回一个list**。

（5）高阶函数sorted()：可以对list进行排序。

此外，sorted()函数也是一个高阶函数，它还可以接收一个key函数来实现自定义的排序，例如按绝对值大小排序：

>>> sorted([36, 5, -12, 9, -21], key=abs)

[5, 9, -12, -21, 36]

key指定的函数将作用于list的每一个元素上，并根据key函数返回的结果进行排序。

17. 匿名函数

当我们在传入函数时，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：

>>> list(map(**lambda x: x \* x**, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

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

通过对比可以看出，匿名函数lambda x: x \* x实际上就是：

def f(x):

return x \* x

**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。 用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。 且 **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：

>>> f = lambda x: x \* x

>>> f

<function <lambda> at 0x101c6ef28>

>>> f(5)

25

同样，也可以把匿名函数作为返回值返回，比如：

def build(x, y):

return lambda: x \* x + y \* y

18. 模块：

（1）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）**。

（2）使用模块有什么好处？

最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。 我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。

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。  自己创建模块时要注意命名，不能和Python自带的模块名称冲突，例如： 系统自带了sys模块，自己的模块就不可命名为sys.py，否则将无法导入系统自带的sys模块。

1. 模块是一组Python代码的集合，可以使用其他模块，也可以被其他模块使用。

创建自己的模块时，要注意：

模块名要遵循Python变量命名规范，不要使用中文、特殊字符；

模块名不要和系统模块名冲突，最好先查看系统是否已存在该模块，检查方法是在Python交互环境执行**import abc** ，若成功则说明系统存在此模块。

（4）Python本身就内置了很多有用的模块，只要安装完毕这些模块就可以立刻使用。

我们以内建的sys模块为例，编写一个hello的模块（hello.py文件）：如下

*#!/usr/bin/env python3*

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

' a test module '

\_\_author\_\_ = 'Michael Liao'

**import** sys

**def** **test**():

args = sys.argv

**if** len(args)==1:

print('Hello, world!')

**elif** len(args)==2:

print('Hello, %s!' % args[1])

**else**:

print('Too many arguments!')

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

test()

第1行和第2行是 **标准注释**： 第1行注释可以让这个hello.py文件直接在Unix/Linux/Mac上运行，第2行注释表示.py文件本身使用标准UTF-8编码；

第4行是一个字符串，表示**模块的文档注释**， 任何模块代码的第一个字符串都被视为模块的文档注释；

第6行使用 **\_\_author\_\_变量** 把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名。

以上就是Python模块的标准文件模板，当然也可以全部删掉不写。

（5）使用sys模块的第一步，就是导入该模块：

import sys

导入sys模块后， 我们就有了 **变量sys指向该模块** ，利用sys这个变量，就可以访问sys模块的所有功能。

**sys模块有一个argv变量** ，用list存储了命令行的所有参数。 argv至少有一个元素，因为第一个参数永远是该 .py文件的名称。

（6）if \_\_name\_\_=='\_\_main\_\_':

test()

当我们在命令行运行hello模块文件时， Python解释器把一个 **特殊变量\_\_name\_\_置为 \_\_main\_\_**， 而如果在其他地方 import导入该hello模块时 ， if判断会失败。

（7） 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， hello模块定义的文档注释也可以用特殊变量 \_\_doc\_\_ 访问，我们自己的变量一般不要用这种变量名；

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。

19. 面向对象编程OOP：

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**。

在面向对象编程的世界中，一切皆为对象，**对象都有属性和行为**，每个对象都是独一无二的，而且对象一定属于某个类（型）。

我们把一大堆拥有共同特征的对象的 **静态特征（属性）** 和 **动态特征（行为）** 都抽取出来后，就可以定义出一个叫做“类”的东西。

我自己对封装的理解是“ **隐藏一切可以隐藏的实现细节，只向外界暴露（提供）简单的编程接口** ”。 我们在类中定义的方法其实就是把数据和对数据的操作封装起来了，在我们创建了对象之后，只需要给对象发送一个消息（调用方法）就可以执行方法中的代码，即我们只需要知道方法的名字和传入的参数，而不需要知道方法内部的实现细节。

20. 类和实例

（1）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(object类) 继承下来的**； 通常若没有合适的继承类，就使用object类， 这是所有类最终都会继承的类。

（2）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 0x10a67a590>

>>> Student

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

可以看到，变量 bart 指向的就是一个Student的实例， 后面的 0x10a67a590 是实例对象的内存地址，每个object的地址都不一样，而Student本身则是一个类。

**可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 = 'Bart Simpson'

>>> bart.name

'Bart Simpson'

（3）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**：

class Student(object):

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

self.name = name

self.score = score

注意： 特殊方法“\_\_init\_\_”前后分别有两个下划线！！！

注意到 **\_\_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不用传**，Python解释器自己会把实例变量传进去：

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

>>> bart.name

'Bart Simpson'

>>> 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**。 除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

（4） 数据封装： 面向对象编程的一个重要特点就是数据封装。在上面的Student类中，每个实例都拥有各自的name和score这些数据。

既然 Student实例 本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接 **在 Student类的内部 来定义访问数据的函数**，这样，就把“数据”给封装起来了。 这些封装数据的函数 是和 Student类本身 是关联起来的， 我们称之为 **类的方法**：

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))

**要定义一个方法，除了第一个参数是self外，其他和普通函数一样**。

要调用一个方法，只需要在实例变量上直接调用，除了self不用传递，其他参数正常传入：

>>> bart.print\_score()

Bart Simpson: 59

这样一来，我们从外部看Student类，就只需要知道： 创建实例需要给出name和 score。 而如何打印，都是在Student类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

21. 访问限制

（1） 外部代码还是可以自由地修改一个实例的name、score属性：

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

>>> bart.score

59

>>> bart.score = 99

>>> bart.score

99

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**， 所以，我们把Student类改一改：

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))

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实例变量.\_\_name 和 实例变量.\_\_score了：

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

>>> bart.\_\_name

Traceback (most recent call last):

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

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

这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

（2） 需要注意的是，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\_\_这样的变量名做私有变量。

有些时候，你会看到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

'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。

22. 继承和多态

（1） 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）、派生类** 或 **衍生类**； 而被继承的class称为 **基类**、**父类** 或 **超类（Base class、Super class）**。

比如，我们已经编写了一个名为Animal的class，有一个run()方法可以直接打印：

class Animal(object):

def run(self):

print('Animal is running...')

当我们需要编写Dog和Cat类时，就可以直接从Animal类继承：

class Dog(Animal):

pass

class Cat(Animal):

pass

对于Dog来说，Animal就是它的父类，对于Animal来说，Dog就是它的子类。 Cat和Dog类似。

**继承**有什么好处？最大的好处是 **子类获得了父类的全部功能**。 由于Animial实现了run()方法，故Dog和Cat作为它的子类，就自动拥有了run()方法：

dog = Dog()

dog.run()

cat = Cat()

cat.run()

运行结果如下：

Animal is running...

Animal is running...

（2） 继承的第二个好处需要我们对代码做一点改进。 你看到了，无论是Dog还是Cat，它们run()的时候，显示的都是Animal is running...，符合逻辑的做法是分别显示Dog is running...和Cat is running...，因此，对Dog和Cat类改进如下：

class Dog(Animal):

def run(self):

print('Dog is running...')

class Cat(Animal):

def run(self):

print('Cat is running...')

再次运行，结果如下：

Dog is running...

Cat is running...

当 **子类和父类都存在相同的run()方法** 时，我们说，子类的run()覆盖了父类的run()， **在代码运行的时候，总是会调用子类的run()**。这样，我们就获得了继承的另一个好处：多态。

（3） 要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个class的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两样：

a = list() # a是list类型

b = Animal() # b是Animal类型

c = Dog() # c是Dog类型

判断一个变量是否是某个类型可以用isinstance()判断：

>>> isinstance(a, list)

True

>>> isinstance(b, Animal)

True

>>> isinstance(c, Dog)

True

看来a、b、c确实对应着list、Animal、Dog这3种类型。

但是等等，试试：

>>> isinstance(c, Animal)

True

看来c不仅仅是 Dog 类型， c 还是 Animal 类型！！

不过仔细想想，这是有道理的，因为Dog是从Animal继承下来的，当我们创建了一个Dog的实例c时， 我们认为c的数据类型是Dog没错， 但c同时也是Animal也没错， Dog本来就是Animal的一种！

所以在继承关系中，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：

>>> b = Animal()

>>> isinstance(b, Dog)

False

Dog可以看成Animal，但Animal不可以看成Dog。

（4） 把一组数据结构和处理它们的方法组成 对象（object）， 把相同行为的对象归纳为 类（class）， 通过 类的封装（encapsulation）隐藏内部细节， 通过继承（inheritance）实现 类的特化（specialization）和 泛化（generalization），通过多态（polymorphism）实现基于对象类型的动态分派。

（5） 子类在继承了父类的方法后，可以对父类已有的方法给出新的实现版本，这个动作称之为 **方法重写（override）**。 通过方法重写我们可以让父类的同一个行为在子类中拥有不同的实现版本，当我们**调 用这个经过 子类重写 的方法时， 不同的子类对象会表现出不同的行为**：这个就是 **多态**。

（6） 要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个Animal类型的变量： （即鸭子类型）

**def** **run\_twice**(animal):

animal.run()

animal.run()

当我们传入Animal类型的实例时， run\_twice()就打印出：

>>> run\_twice(Animal())

Animal **is** running...

Animal **is** running...

当我们传入Dog类型的实例时， run\_twice()就打印出：

>>> run\_twice(Dog())

Dog **is** running...

Dog **is** running...

当我们传入Cat类型的实例时， run\_twice()就打印出：

>>> run\_twice(Cat())

Cat **is** running...

Cat **is** running...

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个Tortoise类型，也从Animal派生：

**class Tortoise(Animal):**

**def** **run**(self):

print('Tortoise is running slowly...')

当我们调用run\_twice()时，传入Tortoise的实例：

>>> run\_twice(Tortoise())

Tortoise **is** running slowly...

Tortoise **is** running slowly...

你会发现，新增的一个Animal的子类，不需要对run\_twice()做任何修改，即任何依赖Animal作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

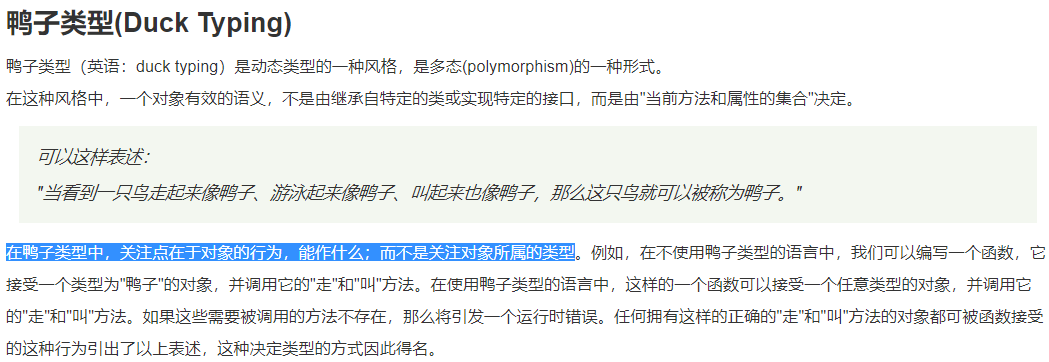
多态的好处就是，当我们需要传入Dog、Cat、Tortoise……时，我们只需要接收Animal类型就可以了，因为Dog、Cat、Tortoise……都是Animal类型，然后，按照Animal类型进行操作即可。 由于Animal类型有run()方法，因此，**传入的任意类型，只要是Animal类 或者 子类，就会自动调用实际类型的run()方法**，这就是多态的意思：

对于一个变量，我们只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal、Dog、Cat还是Tortoise对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种Animal的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。这就是著名的“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。

（7）



### （8）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， 则 **不一定需要传入Animal类型，我们只要保证传入的对象有一个run()方法就可以**了：如下

class Timer(object):

def run(self):

print('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。

（9） 继承可以把父类的所有功能都直接拿过来，这样就不必重零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写。

动态语言的鸭子类型特点决定了继承不像静态语言那样是必须的。

23. 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，比如，获得一个str对象的所有属性和方法：

>>> **dir('ABC')**

['\_\_add\_\_', '\_\_class\_\_',..., '\_\_subclasshook\_\_', 'capitalize', 'casefold',..., 'zfill']

类似 \_\_xxx\_\_的属性和方法 在Python中都有特殊用途，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** ，所以，下面的代码是等价的：

>>> len('ABC')

3

>>> 'ABC'.\_\_len\_\_()

3

注意： 针对于我们自己写的类： 如果也想用len(myObj)的话，就要自己写一个\_\_len\_\_()方法：

>>> class MyDog(object):

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

... return 100

...

>>> dog = MyDog()

>>> len(dog)

100

24. 配合 **getattr()** 、**setattr()** 以及 **hasattr()**，我们可直接操作一个对象的状态:

>>> class MyObject(object):

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

... self.x = 9

... def power(self):

... return self.x \* self.x

...

>>> obj = MyObject()

（1）紧接着，可以测试该对象的**属性**：

>>> **hasattr(obj, 'x')** *# 有属性'x'吗？*

True

>>> obj.x

9

>>> hasattr(obj, 'y') *# 有属性'y'吗？*

False

>>> **setattr(obj, 'y', 19)** *# 设置一个属性'y'*

>>> hasattr(obj, 'y') *# 有属性'y'吗？*

True

>>> **getattr(obj, 'y')** *# 获取属性'y'*

19

>>> obj.y *# 获取属性'y'*

19

（2）如果试图获取不存在的属性，会抛出AttributeError的错误：如下

>>> getattr(obj, 'z') # 获取属性'z'

Traceback (most recent call last):

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

AttributeError: 'MyObject' object has no attribute 'z'

（3）可以传入一个default参数，如果属性不存在，就返回默认值：

>>> getattr(obj, 'z', 404) *# 获取属性'z'，如果不存在，返回默认值404*

404

（4）也可以获得对象的**方法**：

>>> hasattr(obj, 'power') *# 有方法'power'吗？*

True

>>> getattr(obj, 'power') *# 获取obj的方法'power'*

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x10077a6a0>>

>>> fn = getattr(obj, 'power') *# 获取方法'power'并赋值到变量fn*

>>> fn *# fn指向obj.power*

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x10077a6a0>>

>>> fn() *# 调用fn()与调用obj.power()是一样的*

81

注意: 所有的在cmd命令行下对环境变量的修改只对**当前窗口**有效，不是永久性的修改。也就是说当关闭此cmd命令行窗口后，将不再起作用。 永久性修改环境变量的方法有两种：一种是直接修改注册表（此种方法目前没试验过）；另一种是通过我的电脑-〉属性-〉高级，来设置系统的环境变量。

环境变量：是操作系统用来指定运行环境的一些参数。 比如临时文件夹位置和系统文件夹位置等。当你运行某些程序时，除了在当前文件夹中寻找外，还会到这些环境变量中去查找，比如“**Path**”就是一个变量，里面存储了常用的 **命令所存放的 目录路径**。