开场白

什么是python？

1. 一种解释型的，面向对象的，带有动态语义的高级程序设计语言（官方）。
2. 一种想让你在编程时实现自己想法时感觉不那么碍手碍脚的程序设计语言。你可以花较少的代价实现想要的功能，并且编写的程序清晰易懂。
3. 基础知识

一、数字和表达式

幂（乘方）运算符：\*\*，并且该运算符的优先级比取反（一元减运算符）的优先级要高。

二、语句

1. 表达式是某件事情，而语句是做某件事情（即告诉计算机做什么）。
2. 交互式解释器总是把所有表达式的值打印出来！

三、获取用户输入

input()或input(“\*\*\*\*\*\*\*\*\*\*\*”)，后者更为友好。

四、函数

1.pow()：返回乘方值ab

有两种：math模块的math.pow()和内置的pow()

注意：pow() 通过内置的方法直接调用，内置方法会把参数作为整型，而 math 模块则会把参数转换为 float。

2.abs(number)：返回数字的绝对值

3.float(number)：将字符串和数字转换为浮点数

4.math.ceil(number)：返回数的上入整数，返回值的类型为整数

5. math.floor(number)：返回数的下舍整数，返回值的类型为整数

6.math.sqrt(number)：返回平方根，不适用于负数

7.str()：将值转换为字符串

8.round(number) ：四舍五入（慎用，有bug）

五、模块

1. 用import导入模块，然后按照“模块.函数”的格式使用这个模块的函数。

e.g:import math

math.floor(32.9)

🡺32

1. 使用“from模块import函数”这种形式的import命令后，就可以直接使用函数，而不需要模块名作为前缀。

e.g:from math import sqrt

sqrt(9)

🡺3.0

1. 第二种import形式可能会带来函数使用上的冲突，例如math里的sqrt和cmath里的sqrt。所以除非真的需要from这个形式的模块导入语句，否则应该坚持使用普通的import。

六、执行程序

在代码的最后加上input(“Press <enter>”)可以防止在windows上双击运行时一闪而过。

七、注释

#\*\*\*\*\*\*\*\*（行注释）

八、字符串

1. 拼接字符串

（1）一个接着另一个，可以实现自动拼接。

（2）使用“+”。

1. 长字符串

如果需要写一个非常长的字符串，它需要跨多行，可以使用三个引号代表普通引号。

e.g:print(’’’abc

def

ghi’’’)

🡺abc

def

ghi

1. 普通字符串也可以跨行。如果一行之中最后一个字符是反斜线，那么，换行符本身就“转义”了，也就是被忽略了。例如：

print(“hello,\

world!”)

🡺helloworld!

1. 原始字符串：r’\*\*\*\*\*\*’
2. 列表和元组

数据结构是通过某种方式组织在一起的数据元素的集合，这些数据元素可以是数字或者字符，甚至可以是其他数据结构。在python中，最基本的数据结构是序列。

一、序列概览

最常用的两种类型：列表（list）和元组（tuple）。

主要区别在于，列表可以修改，元组则不能。也就是说如果要根据要求来添加元素，那么列表可能会更好用；而出于某些原因，序列不能修改时，使用元组则更为合适。使用后者的原因通常是技术性的，它与python内部的运作方式有关。

二、通用序列操作

索引，分片，加，乘，成员资格

1. 索引

序列中的所有元素都是有编号的——从0开始递增。

字符串就是一个由字符组成的序列。

使用负数索引时，python会从右边开始计数。最后1个元素的位置编号为-1.

1. 分片

使用分片操作来访问一定范围内的元素。

简而言之，分片操作的实现需要提供两个索引作为边界，第1个索引的元素是包含在分片内的，而第2个则不包含在分片内。

e.g:numbers = [1,2,3,4,5,6,7,8,9,10]

numbers[3:6]

🡺[4,5,6]

1. 优雅的捷径

为了让分片部分能够包含列表的最后一个元素，必须提供最后一个元素的下一个元素所对应的索引作为边界。

e.g:numbers[7:10]

🡺[8,9,10]

实际上，只要分片中最左边的索引比它右边的晚出现在序列中，结果就是一个空的序列（默认步长为1的情况下）。幸好，可以使用一个捷径：如果分片所得部分包括序列结尾的元素，那么，只需空置最后一个索引即可：

e.g:numbers[-3:]

🡺[8,9,10]

这种方法同样适用于序列开始的元素：

e.g: numbers[:3]

🡺[1,2,3]

实际上，如果需要复制整个序列，可以将两个索引都置空：

e.g:numbers[:]

🡺[1,2,3,4,5,6,7,8,9,10]

1. 更长的步长

在普通的分片中，步长默认是1。

e.g:numbers[0:10:1]

🡺[1,2,3,4,5,6,7,8,9,10]

numbers[0:10:2]

🡺[1,3,5,7,9]

当然，步长不能为0，但可以为负数，此时分片从右到左提取元素：

e.g:numbers[0:10:-1]

🡺 [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

1. 序列相加

通过加运算符可以进行序列的连接操作，但仅限于两种相同类型的序列。

1. 乘法

用数字x乘以一个序列会生成新的序列，而在新的序列中，原来的序列将被重复x次。

None是一个python的内建值，它的确切含义是“这里什么也没有”。

1. 成员资格

in运算符，返回布尔值。

格式：字符或字符串 in 序列

1. 长度、最小值和最大值

len：返回序列中所包含的元素的数量。

min：返回序列中最小的元素。

max：返回序列中最大的元素。

三、列表

1. list函数

可以实现将所有类型的序列转换为list！

‘’.join(somelist)可以实现将一个由字符组成的列表转换为字符串。

1. 基本的列表操作

（1）改变列表：元素赋值

注意：不能为一个位置不存在的元素进行赋值。

（2）删除元素：del语句

a = ['1', '2', '3']

del a[0]

print(a)

🡺['2', '3']

1. 分片赋值（骚操作）
2. 列表方法

方法是一个与某些对象有紧密联系的函数。一般来说，方法可以这样进行调用：对象.方法（参数）

1. append：用于在列表末尾追加新的对象。
2. count：统计某个元素在列表中出现的次数。
3. extend：可以在列表的末尾一次性追加另一个序列中的多个值。换句话说，可以用新列表扩展原有的列表：

a = [1,2,3]

b = [4,5,6]

1. extend(b)

a

🡺[1,2,3,4,5,6]

这个操作与连接操作最主要的区别在于：extend方法修改了被扩展的序列，而原始连接操作则不然，它会返回一个全新的列表。

1. index：用于从列表中找出某个值第一个匹配项的索引位置。
2. insert：用于将对象插入到列表中。insert(位置，对象)
3. pop：移除列表中的一个元素（默认是最后一个），并且返回该元素的值。pop（位置）

pop方法是唯一一个既能修改列表又返回元素值的列表方法。

1. remove：用于移除列表中某个值的第一个匹配项。（没有返回值）
2. reverse：将列表中的元素反向存放。（没有返回值）

reversed函数：作用同上，但返回的是一个迭代器（iterator）

1. sort：用于在原位置对列表进行排序（默认升序），不返回。

sorted函数：作用同上，返回一个列表。

（10）高级排序

sort(key = 函数,reverse = 布尔值)

四、元组

元组与列表唯一的不同是元组不能修改。实际上，如果你用逗号分隔了一些值，那么你就自动创建了元组。

元组大部分时候是通过圆括号括起来的。

空元组可以用没有包含内容的两个圆括号来表示。

实现包括一个值的元组必须加个逗号，即使只有一个值

1. tuple函数：以一个序列作为参数并把它转换为元组。
2. 那么，意义何在？
3. 元组可以在映射（和集合的成员）中当作键使用——而列表则不行。
4. 元组作为很多内建函数和方法的返回值存在，也就是说你必须对元组进行处理。

一般来说，列表可能更能满足对序列的所有需求。

五、小结

序列是一种数据结构，它包含的元素都进行了编号。典型的序列包括列表、元组和字符串。其中，列表是可变的，而元组和字符串都是不可变的（一旦创建了就是固定的）。通过分片可以完成访问序列等骚操作。

第三章 使用字符串

一、字符串格式化：完整版

如果右操作数是元组的话，则其中的每一个元素都会被单独格式化，每个值都需要一个对应的转换说明符。

%字符：标记转换说明符的开始。

转换标志（可选）：-表示左对齐；+表示在转换值之前要加上正负号（在需要对齐正负数时会很有用）；空格表示正数之前保留空格（在需要对其正负数时会很有用）；0表示转换值若位数不够则用0填充。

最小字段宽度（可选）：转换后的字符串至少应该具有该值指定的宽度。如果是\*，则宽度的值会从元组中读出。

点（.）后跟精度值（可选）：如果转换的是实数，精度值就表示出现在小数点后的位数。如果转换的是字符串，那么该数字就表示最大字段宽度。如果是\*，则宽度的值会从元组中读出。

转换类型：

|  |  |
| --- | --- |
| 转换类型 | 含义 |
| d,i | 十进制 |
| f,F | 十进制浮点数 |
| o | 八进制 |
| x,X | 十六进制（小写，大写） |
| e,E | 科学计数法表示的浮点数（小写，大写） |
| c | 单字符（接受整数或单字符） |
| s | 字符串 |

1. 字符串方法

（1）find：在一个较长的字符串中查找子串，返回子串所在位置的最左端索引，如果没有找到则返回-1。这个方法还可以接收可选的起始点和结束点参数。（按照惯例，不包括第二个索引）

（2）join：用来连接序列中的元素，但被连接的序列元素都必须是字符串。格式：连接字符（串）.join(序列)

（3）lower：返回字符串的小写字母版。

（4）replace：返回某字符串的所有匹配项均被替换之后得到的字符串。

（5）split（join的逆方法）：用来将字符串分割成序列。注意，如果不提供任何分隔符，程序会把所有空格作为分隔符。格式：字符串.split(分隔字符串)

（6）strip：返回去除两侧（不包括内部）空格的字符串。可以指定需要去除的字符，将它们列为参数即可，但只会去除两侧的字符。

（7）translate：和替换类似，但只处理单个字符。它的优势在于可以同时进行多个替换，有时候比replace效率更高。在使用该方法之前，需要先完成一张转换表。转换表中是以某字符替换某字符的对应关系。因为这个表有多达256个项目，使用string模块里的maketrans函数会比较方便。maketrans接收两个等长的字符串，表示第一个字符串中的每个字符都用第二个字符串中相同位置的字符替换。略（python3用法改变）

第四章 字典

通过名字来引用值的数据结构——映射（mapping）。字典是python中唯一内建的映射类型。字典中的值并没有特殊的顺序，但是都储存在一个特定的键（key）下。键可以是数字，字符串甚至是元组。

一、创建和使用字典

phonebook = {'Peter':'13333333333','Tom':'14444444444'}

字典由多个键与其对应的值构成的键值对组成（键值对也称为项）。每个键和它的值之间用冒号（:）隔开，项之间用逗号（,）隔开，而整个字典是用一对大括号（{}）括起来。空字典由两个大括号组成，像这样{}。

注意：字典中的键是唯一的，而值并不唯一。

1. dict函数：通过其他映射或者键值对的序列建立字典。

items = [('Name','Tom'),('Age','42')]

d = dict(items)

🡺 d = {'Name': 'Tom', 'Age': '42'}

也可以通过关键字参数来创建字典。

d = dict(name='Tom',age = 15)

2. 基本字典操作

字典的基本行为在很多方面与序列类似：

len(d)：返回键值对的数量

d[key]：返回关联到键key上的值

d[key] = v：将值v关联到键上

del d[key]：删除键为key的项

key in d：检查d中是否有含有键为key的项

键类型：字典的键不一定为整型数据，键可以是任意的不可变类型，例如浮点数，字符串或元组。

自动添加：即使键起初在字典中并不存在，也可以为它赋值，这样字典就会建立新的项，而列表并不能这样。

成员资格：表达式 key in dict查找的是键，而不是值。表达式v in list则用来查找值，而不是索引。

提示：在字典中检查键的成员资格比在列表中检查值的成员资格更加高效，数据结构的规模越大，两者的效率差距越明显。

3. 字典的格式化字符串

在每个转换说明符中过得%字符后面，可以加上键（用圆括号括起来），后面再跟上其他说明要素。只要所有给出的键都能在字典中找到，就可以使用任意数量的转换说明符。

4. 字典方法

（1）clear：清除字典中所有的项，无返回值。

（2）copy：返回一个具有相同键值对的新字典（浅复制）

>>> x = {'username':'admin','machines':['bar','foo','baz']}

>>> y = x.copy()

>>> y['username']

'admin'

>>> y['username'] = 'Tom'

>>> y

{'username': 'Tom', 'machines': ['bar', 'foo', 'baz']}

>>> x

{'username': 'admin', 'machines': ['bar', 'foo', 'baz']}

>>> y['machines'].pop(1)

'foo'

>>> y

{'username': 'Tom', 'machines': ['bar', 'baz']}

>>> x

{'username': 'admin', 'machines': ['bar', 'baz']}

可以看到，当在副本中替换值的时候，原始字典不受影响，但是，如果修改了某个值（原地修改，而不是替换），原始的字典也会改变，因为同样的值也储存在原字典中。

避免这种问题的一个方法是采用深复制。

>>> from copy import deepcopy

>>> z = deepcopy(x)

>>> z

{'username': 'admin', 'machines': ['bar', 'baz']}

>>> x['machines'].append('dsds')

>>> x

{'username': 'admin', 'machines': ['bar', 'baz', 'dsds']}

>>> z

{'username': 'admin', 'machines': ['bar', 'baz']}

（3）fromkeys：使用给定的键建立新的字典，每个键都对应一个默认的值None。

>>> {}.fromkeys(['name','age'])

{'name': None, 'age': None}

也可以自己提供默认值。

>>> {}.fromkeys(['name','age'],'unknow')

{'name': 'unknow', 'age': 'unknow'}

（4）get：这是一个更宽松的访问字典项的方法。一般来说，如果试图访问字典中不存在的项是会出错，而get就不会，当get访问一个不存在的键时，没有任何异常，而得到了None值。还可以自己定义“默认值”，替换None。

d = {}

print(d.get('name','N/A'))

get方法使得程序在用户输入我们并未准备的值时也能做出合理的反应。

（5）items：

items方法将字典所有的项以列表方式返回，列表中的每一项都表示为（键，值）对的形式。但是项在返回时并没有遵循特定的次序？

（6）keys：将字典中的键以迭代器方式返回。

（7）pop：用来获得对应于给定健的值，然后将这个键值对从字典中移除。

（8）popitem：popitem弹出随机的项，因为字典并没有“最后的元素”或者其他有关顺序的概念。若想一个接着一个地移除并处理项，这个方法就很骚了！

（10）setdefault：在某种程度上类似于get方法，能够获得与给定键相关联的值，除此之外，setdefault还能在字典中不含有给定键的情况下设定相应的键值。当键不存在的时候，setdefault返回默认值并且相应地更新字典。如果键存在，那么就返回与其对应的值，但不改变字典。默认值是可选的，这点和get不一样。如果不设定，会默认使用None。

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

>>> x.setdefault("a")

1

>>> x.setdefault('d')

>>> x

{'a': 1, 'b': 2, 'c': 3, 'd': None}

>>> x.setdefault('a','sdsdssdsds')

1

>>> x.setdefault('e','sdsdssdsds')

'sdsdssdsds'

>>> x

{'a': 1, 'b': 2, 'c': 3, 'd': None, 'e': 'sdsdssdsds'}

（11）update：利用一个字典项更新另一个字典。提供的字典中的项会被添加到旧的字典中，若有相同的键则会进行覆盖。

>>> x = {'a':1,'y':2,'z':3}

>>> y = {'d':4}

>>> x.update(y)

>>> x

{'a': 1, 'y': 2, 'z': 3, 'd': 4}

>>> z = {'a':5}

>>> x.update(z)

>>> x

{'a': 5, 'y': 2, 'z': 3, 'd': 4}

（12）values和itervalues：以列表的形式返回字典中的值。

第五章 条件、循环和其他语句

一、print和import的更多信息

1.使用逗号输出：打印多个表达式时用逗号将每个表达式隔开，可以看到，每个参数之间都插入了一个空格符。

print('age',123456,'number')

🡺age 123456 number

2.把某个事作为另一件事导入

从模块导入函数的时候，通常可以使用

import somemodule

from somemodule import somefunction

from somemodule import somefunction,anotherfunction

from somemodule import \*

如果有同名函数，只需使用第一种方式导入即可，或者可以在语句末尾增加一个as语句，在该句子后给出想要使用的别名。例如为整个模块提供别名：import math as foobar；为函数提供别名：from math import sqrt as foobar。

二、赋值魔法

1.序列解包

多个赋值操作可以同时进行：

>>> x,y,z = 1,2,3

>>> print(x,y,z)

1 2 3

用它交换两个变量也是没问题的。

>>> x,y = y,x

>>> x

2

>>> y

1

这里所做的事情叫做序列解包或递归解包。当函数或者方法返回元组时，这个特性尤为有用。

arr = {'name':'Peter','age':'26'}

key,value = arr.popitem()

print(key,value)

🡺age 26

它允许函数返回一个以上的值并且打包成元组，然后通过一个赋值语句很容易进行访问。所解包的序列中的元素数量必须和放置在赋值符号=左边的变量数量完全一致，否则会发生异常。

2.增量赋值

+=，-=，\*=，/=

三、语句块

语句块是在条件为真（条件语句）时执行或者多次执行（循环语句）的一组语句。在代码前放置空格来缩进语句即可创建语句块。

块中的每行都应该缩进同样的量。

冒号（:）用来标识语句块的开始，当回退到和已经闭合的块一样的缩进量时，就表示当前块已经结束了。

四、条件和条件语句

1.这就是布尔变量的作用

下面的值在作为布尔表达式的时候，会被解释器看作假（False）：

False , None , 0 , “” , () , [] , {}

也就是标准值Flase和None、所有类型的数字0、空序列和空字典都为假，其他的一切都解释为真。事实上，True和False只不过是1和0的一种“华丽”的说法而已：

>>> True == 1

True

>>> False == 0

True

布尔值True和False属于布尔类型，bool函数可以用来转换其他值，但因为所有值都可以用作布尔值，所以几乎不需要对它们进行显示转换。

注意：尽管[]和””都是假值，它们本身却并不相等（也就是说[] != “”）

2.条件执行和if语句

如果条件判定为真，那么后面的语句块就会被执行。如果条件为假，语句块就不会被执行。

x = input()

if int(x) > 5:

print("good number!")

3.else子句（之所以叫做子句是因为它不是独立的语句，而只能作为if语句的一部分）。

x = input()

if int(x) > 5:

print("good number!")

else:

print("bad number!")

4.elif子句

如果需要检查多个条件，就可以使用elif，它是else if的简写。

5.嵌套代码块（略）

6.更复杂的条件

（1）比较运算符：

|  |  |
| --- | --- |
| 表达式 | 描述 |
| x is y | x和y是同一个对象 |
| x is not y | x和y是不同的对象 |
| x in y | x是y容器的成员 |
| x not in y | x不是y容器的成员 |

（2）相等运算符（==）

（3）is：同一性运算符

x = y = [1,2,3]

z = [1,2,3]

print(x==y)

print(x==z)

print(x is y)

print(x is z)

True

True

True

False

总结一下：使用==运算符来判定两个对象是否相等，使用is判定两者是否等同（同一个对象）

（4）in：成员资格运算符

（5）字符串和序列比较

（6）布尔运算符

事实上，所有值都可以解释为布尔值，所有的表达式也都返回布尔值。如果想要检查一个以上的条件，有时会用到使用神奇的连接比较（1<number<10，和用我们的数学常识来理解是同一个意思）。

布尔运算符有个有趣的特性：只有在需要求值时才进行求值。举例来说，表达式x and y需要两个变量都为真时才为真，所以如果x为假，表达式就会立刻返回False，而不管y的值。实际上，如果x为假，表达式会返回x的值——否则它就返回y的值。这种行为被称为短路逻辑或惰性求值。这意味着在布尔运算符之后的所有代码都不会执行。

它主要是避免了无用地执行代码，可以作为一种技巧使用，可以用来实现c和java中所谓的三元运算符。

（7）断言

设计哲学：与其让程序在晚些时候崩溃，不如在错误条件出现时直接让它崩溃。一般说来，你可以要求某些条件必须为真。语句中使用的关键字是assert。

age = 10

assert age < 0

PS F:\python project> python .\test.py

Traceback (most recent call last):

File ".\test.py", line 2, in <module>

assert age < 0

AssertionError

如果需要确保程序中的某个条件一定为真时才能让程序正常工作的话，assert语句就有用了，它可以在程序中置入检查点。条件后可以添加字符串，用来解释断言。

age = 10

assert age < 0,'This error is mother fucker!'

五、循环

1.while循环

x = 1

while x <= 100:

print(x)

x += 1

2.for循环

如果要为一个集合（序列和其他可迭代对象）的每个元素都执行一个代码块。

number = [0,1,2,3,4,5]

for x in number:

print(x)

因为迭代某范围的数字是很常见的，所以有个内建的范围函数供使用：

range(a,b)。它包含下限，但不包含上限。但如果希望下限为0，可以只提供上限。

提示：如果能使用for循环，就尽量不用while循环。

3.循环遍历字典元素

d = {'x':1,'y':2,'z':3}

for key in d:

print(key+':'+str(d[key]))

如果只需要值，可以使用d.values()

d = {'x':1,'y':2,'z':3}

for value in d.values():

print(value)

d.items()方法会将键值对作为元组返回，for循环的一大好处就是可以循环中使用序列解包。

d = {'x':1,'y':2,'z':3}

for key,value in d.items():

print(key,value)

4.一些迭代工具

（1）并行迭代

zip函数：返回迭代器

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

number = [1,2,3]

for a,b in zip(name,number):

print(a,b)

zip函数可以作用于任意多的序列，重要的是它可以处理不等长的序列，当最短的序列“用完”的时候就会停止。

a = range(0,5)

b = range(0,10)

print(list(zip(a,b)))

（2）按索引迭代

for index,string in enumerate(strings):

if 'xxx' in string:

string[index] = '[censored]'

（3）翻转和排序迭代

reversed和sorted：不是原地修改对象，而是返回一个翻转或排序后的版本。sorted方法返回列表，reversed方法返回可迭代对象。

5.跳出循环

（1）break：结束（跳出）循环。

（2）continue：跳过剩余的循环体，但是不结束循环。

（3）while True/break习语（略）

6.循环中的else子句

在 python 中，for … else 表示这样的意思，for 中的语句和普通的没有区别，else 中的语句会在循环正常执行完（即 for 不是通过 break 跳出而中断的）的情况下执行，while … else 也是一样。

六、列表推导式——轻量级循环

列表推导式是利用其他列表创建新列表的一种方法。它的工作方式类似于for循环，也很简单。

print([x\*x for x in range(0,10)])

这个语句可以通过增加一个if部分添加到列表推导式中。

也可以增加更多for语句的部分：

print([(x,y) for x in range(0,10) for y in range(0,10)])

七、三人行

1.什么都没法发生

pass：作为一个占位符

2.使用del删除

一般说来，python会删除那些不再使用的对象，特别是当没有任何名字绑定到它上面，没有办法获取和使用它，所以python解释器会直接删除这个对象（这种行为被称为垃圾收集）。

另外一种方法是使用del语句，它不仅会移除一个对象的引用，也会移除那个名字本身，但无法删除那个值。

3.使用exec和eval执行和求值字符串

用于动态地创造python代码，然后将其作为语句执行或作为表达式计算，这可能近似于“黑暗魔法”，因此在使用之前，一定要慎之又慎，仔细考虑。

（1）exec：执行一系列python语句

exec("print('hello,world!')")

from math import sqrt

scope = {}

exec('sqrt = 1',scope )

print(sqrt(4))

print(scope['sqrt'])

潜在的破坏性代码并不会覆盖sqrt函数，原来的函数能正常工作，而通过exec赋值的变量sqrt只在它的作用域内有效。

（2）eval：计算python表达式，并返回结果值。（exec并不返回任何对象，因为它本身就是语句）

print(eval(input()))

第六章 抽象

一、创建函数

def语句

文档化函数：在函数的开头写下字符串，它就会作为函数的一部分进行存储，这称为文档字符串。

def hello(number):

'This is a test!'

return number\*\*2

文档字符串可以按如下方式访问：

print(hello.\_\_doc\_\_)

help函数可以得到内建函数的帮助信息。

二、参数魔法

在函数内为参数赋予新值不会改变外部任何变量的值：

def try\_to\_change(n):

n = 'Peter'

name = 'Tom'

try\_to\_change(name)

print(name)

🡺Tom

具体的工作方式类似于下面这样：

name = 'Tom'

n = name

n = 'Peter'

print(name)

如果是将可变的数据结构用作参数，情况便会有所不同：

def change(n):

n[0] = 'Peter'

name = ['Tom','Green']

change(name)

print(name)

🡺['Peter', 'Green']

因为这就相当于

name = ['Tom','Green']

n = name

n[0] = 'Peter'

print(name)

当两个变量同时引用一个列表的时候，它们的确是同时引用一个列表！要避免这种情况的发生，可以考虑复制一个列表的副本（切片）。

1.为什么要修改参数？

使用函数改变数据结构（比如列表或字典）是一种将程序抽象化的好方法。

2.如果我的参数不可变呢？

这个时候应该从函数中返回你所有你需要的值或者将值放入列表中。

3.关键字参数和默认值

位置参数（略）

这类使用参数名提供的参数叫做关键字参数。它的主要作用在于可以明确每个参数的作用，也就避免了奇怪的函数调用。

def hello(a,b):

print('%s,%s' % (a,b) )

hello(a='hello',b='world!')

它最厉害的地方在于可以在函数定义中给参数提供默认值：

def hello(a = 'hello',b = 'world!'):

print('%s,%s' % (a,b) )

hello()

但也可以提供一些或所有的参数。

def hello(a = 'hello',b = 'world!'):

print('%s,%s' % (a,b) )

hello('Peter','wtf?')

注意：除非完全清楚程序的功能和参数的意义，否则应该避免混合使用位置参数和关键字参数。

4.收集参数

def print\_params(\*params):

print(params)

print\_params('Peter')

🡺('Peter',)

结果作为元组打印出来。

def print\_params(title,\*params):

print(title)

print(params)

print\_params('Peter',1,2,3)

🡺Peter

(1, 2, 3)

星号的意思是“收集其余的位置参数”，如果不提供任何供收集的元素，parms就是个空元组。

def print\_params(title,\*params):

print(title)

print(params)

print\_params('Peter')

🡺Peter

()

处理关键字参数的“收集”操作——“\*\*”

def print\_params(\*\*params):

print(params)

print\_params(a = 'Peter' , b = 'Tom')

🡺{'a': 'Peter', 'b': 'Tom'}

返回的是字典！

放一起看看！

def print\_param(x,y,z=3,\*pospar,\*\*keypar):

print(x)

print(y)

print(z)

print(pospar)

print(keypar)

print\_param(1,2,3,5,6,7,foo = 1,bar = 2)

🡺1

2

3

(5, 6, 7)

{'foo': 1, 'bar': 2}

5.参数收集的逆过程

def add(x,y):

return x + y

params = (1,2)

print(add(\*params))

🡺3

def hello(a = 'hello',b = 'world!'):

print('%s,%s' % (a,b))

params = {'a':"123",'b':"456"}

hello(\*\*params)

🡺123,456

三、作用域

在执行x = 1赋值语句后，名称x引用到值1，就像字典一样，键引用值，当然，变量和所对应的值用的是个“不可见”字典。

vars()：返回全局变量的字典

locals()：返回局部变量的字典

x = 1

scope = vars()

print(scope['x'])

🡺1

这类“不可见字典”叫做命名空间或者作用域。

除了全局作用域外，每个函数调用都会创建一个新的作用域：

def foo():

x = 42

x = 1

foo()

print(x)

这里的foo函数改变了（重绑定）了变量x，但是在最后的时候，x并没有变。这是因为当调用foo的时候，新的命名空间就被创建了，它作用于foo内的代码块。赋值语句x = 42只在内部作用域（局部命名空间）起作用，所以它并不影响外部（全局）作用域中的x。函数内的变量被称为局部变量。参数的工作原理类似于局部变量，所以用全局变量的名字作为参数名并没有问题。

def output(x):

print(x)

x = 1

y = 2

output(y)

🡺2

屏蔽引发的问题：

如果局部变量或者参数的名字和想要访问的全局变量名不同的话，一般来说直接访问是没有问题的；但如果局部变量或者参数的名字和想要访问的全局变量名相同的话，就不能直接访问了，全局变量会被局部变量屏蔽。这时候就需要globals()函数了！

def combine(parameter):

print(parameter + globals()['parameter'])

parameter = 'herry'

combine('Tom')

如果在函数内部将值赋予一个变量，它会自动成为局部变量——除非告知python将其声明为全局变量，这样便可引用事先定义好的全局变量：

x = 1

def change():

global x

x += 1

change()

print(x)

四、函数式编程

1.map()函数接收两个参数，一个是函数，一个是Iterable，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Iterator返回。

def f(x):

return x \* x

print(list(map(f,[1,2,3,4])))

🡺 [1, 4, 9, 16]

2.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)

from functools import reduce 🡨

def add(x,y):

return x + y

print(reduce(add,[1,2,3,4]))

3. 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(x):

return x % 2 == 1

print(list(filter(is\_odd,[1,2,3,4,5,6,7])))

4.匿名函数

当我们传入函数时，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。

print(list(map(lambda x: x\*x ,[1,2,3,4,5])))

lambda x: x \* x实际上就是：

def f(x):

return x \* x

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。此外，匿名函数也是一个对象，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

print(f(5))

第七章 更加抽象

创建自己的对象，这将是本章要介绍的内容。创建自己的对象是python的核心概念！

一、对象的魔力

对象，基本上可以看做数据（特性）以及由一系列可以存取、操作这些数据的方法所组成的集合。对象最重要的优点包括：

多态：意味着可以对不同类的对象使用同样的操作

封装：对外部世界隐藏对象的工作细节

继承：以通用的类为基础建立专门的类对象

1.多态

多态意味着就算不知道变量所引用的对象类型是什么，还是能对它进行操作，而它也会根据对象（或类）类型的不同的而表现出不同的行为。

（1）多态和方法

注：标准库random中的choice函数，可以从序列中随机选出元素，给变量赋值。

from random import choice

x = choice(['hello,world!',[1,2,3,4]])

print(x)

（2）多态的多种形式

任何不知道对象到底是什么类型，但是又要对对象“做点儿什么”的时候，都会用到多态。

注：repr()函数将对象转化为供解释器读取的形式。

print(repr('hello,world!'))

print('hello,world!')

🡺'hello,world!'

hello,world!

如果可能的话，应该尽力避免使用显式检查类型的函数，这样会毁掉多态！

2.封装

封装是指向程序中的其他部分隐藏对象的具体实现细节的原则。封装不等同于多态，多态可以让用户对于不知道是什么类的对象进行方法调用，而封装是可以不用关心对象是如何构建的而直接进行使用。

3.继承（略）

二、类和类型

1.类到底是什么？

所有的对象都属于某一个类，称为类的实例（instance）。当一个对象所属的类是另外一个对象所属类的子集时，前者就被称为后者的子类（subclass），相反，后者被称为前者的超类（superclass）。

2.创建自己的类

class Person:

def setName(self,name):

self.name = name

def getName(self):

return self.name

def greet(self):

print("Hello,world! I am %s" % self.name)

foo = Person()

bar = Person()

foo.setName('Peter')

bar.setName("Tom")

foo.greet()

bar.greet()

print(foo.name)

🡺Hello,world! I am Peter

Hello,world! I am Tom

Peter

在调用foo的setName和greet函数时，foo自动将自己作为第一个参数传入函数中——因此形象地命名为self。没有它的话，成员方法就没法访问他们要对其特性进行操作的对象本身了。

3.特性、函数和方法

self参数事实上正是方法和函数的区别。方法将他们的第一个参数绑定到所属的实例上，因此您无需显式提供该参数。

class Bird:

song = 'Squaawk'

def sing(self):

print(self.song)

bird = Bird()

bird.sing()

birdsong = bird.sing()

birdsong

🡺Squaawk

Squaawk

4.再论私有化

class Class:

def setName(self,name):

self.name = name

def getName(self):

print(self.name)

def greet(self):

print("hello,"+self.name)

c = Class()

c.name = 'Peter'

c.getName()

有些程序员觉得这样做是可以的，但是有些人觉得这样就破坏了封装的原则。他们认为对象的状态对于外部应该是完全隐藏（不可访问）的。关键在于其他程序员可能不知道（可能也不应该知道）你的对象内部的具体操作，因此，使用外部对象无法访问，而访问器（setName和getName等）能够访问的特性就显得更有原则。

PYTHON并不直接支持私有方式，而要靠程序员自己把握在外部进行特性修改的时机。毕竟在使用对象前应该知道如何使用，但是，可以用一些小技巧达到私有特性的效果。为了将方法或者特性变为私有（从外部无法访问），只要在它的名字前面加上双下划线即可：

class Secretive:

def \_\_inaccessible(self):

print("You can't see me!")

def accessible(self):

print("The secret is ")

self.\_\_inaccessible()

s = Secretive()

s.\_\_inaccessible()

s.accessible()

现在，\_\_inaccessible从外界是无法访问的，而在类内部还能使用（比如从accessible）访问：

🡺Traceback (most recent call last):

File ".\test.py", line 9, in <module>

s.\_\_inaccessible()

AttributeError: 'Secretive' object has no attribute '\_\_inaccessible'

The secret is

You can't see me!

实际上，类的内部定义中。所有以双下划线开始的名字都被“翻译”成前面加上单下划线的类名的形式，所以，实际上还是能在类外访问这些私有方法！

s.\_Secretive\_\_inaccessible()

🡺You can't see me!

简而言之，确保其他人不会访问对象的方法和特性是不可能的，但是这类“名称变化术”就是他们不应该访问这些函数或特性的强有力信号。

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”。

5.类的命名空间

所有位于class语句中的代码都在特殊的命名空间中执行——类命名空间。这个命名空间可由类内所有成员访问。类的定义其实就是执行代码块，这一点非常有用。

class MemberCounter():

members = 0

def init(self):

MemberCounter.members += 1

m1 = MemberCounter()

m1.init()

print(MemberCounter.members)

m2 = MemberCounter()

m2.init()

print(MemberCounter.members)

print(m1.members)

print(m2.members)

m1.members = 'two'

print(m1.members)

print(m2.members)

🡺1

2

2

2

two

2

6.指定超类

子类可以拓展超类的定义，将其他类名写在class语句后的圆括号内可以指定超类：

class Filter:

def init(self):

self.blocked = []

def filter(self,sequence):

return [x for x in sequence if x not in self.blocked]

class SpamFilter(Filter):

def init(self):

self.blocked = ['spam']

Filter是个用于过滤序列的通用类，事实上它不能过滤任何东西。

Filter类的用处在于它可以作为其他类的超类，比如SpamFilter类，可以将序列中的’spam’过滤出去。

重点：

（1）用提供新定义的方法重写了Filter的init定义

（2）filter方法是从Filter类继承的，所以不需要重写它的定义。

7.检查继承

如果想要查看一个类是否是另一个的子类，使用内建函数issubclass函数

print(issubclass(SpamFilter,Filter))

🡺True

如果想要知道已知类的超类们，可以直接使用它的特殊特性\_\_bases\_\_：

print(SpamFilter.\_\_bases\_\_)

print(Filter.\_\_bases\_\_)

🡺(<class '\_\_main\_\_.Filter'>,)

(<class 'object'>,)

同时，还能使用isinstance方法检查一个对象是否是一个类的实例：

s = SpamFilter()

print(isinstance(s,SpamFilter))

print(isinstance(s,Filter))

print(isinstance(s,str))

🡺True

True

False

如果只想知道一个对象属于哪个类，可以使用\_\_class\_\_特性或者type（对象）：

s = SpamFilter()

print(s.\_\_class\_\_)

print(type(s))

🡺<class '\_\_main\_\_.SpamFilter'>

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

8.多个超类

class Calculator:

def calculate(self,expression):

self.value = eval(expression)

class Talker:

def talk(self):

print('Hi,my value is ',self.value)

class TalkingCalculator(Calculator,Talker):

pass

子类（TalkingCalculator）自己不做任何事，它从自己的超类继承所有的行为。

tc = TalkingCalculator()

tc.calculate('1+2+3')

tc.talk()

🡺Hi,my value is 6

这种行为称为多重继承，是个非常有用的工具，但除非特别熟悉，否则应该尽量避免使用，因为有些时候会出现不可预见的麻烦。

当使用多重继承时，有个特别需要注意的地方。如果一个方法从多个超类继承（也就是说你有两个具有相同名字的不同方法），那么必须要注意一下超类的顺序（在class语句中）：先继承的类中的方法会重写后继承的类中的方法。

9.接口和内省

在python中，不用显式地指定对象必须包含哪些方法才能作为参数接收，可以在使用对象时假定它可以实现你所要求的行为，如果它不能实现的话，程序就会失败。

检查所需方法是否存在时使用hasattr（对象，方法名的字符串形式）

print(hasattr(tc,'talk'))

🡺True

检查所需方法是否存在，而且可在特性不存在时提供一个默认返回值时使用getattr（对象，方法名的字符串形式，默认值）

print(getattr(tc,'talking',404))

🡺404

setattr可以用来设置对象的特性：

setattr(tc,'name','Peter')

print(tc.name)

🡺Peter

第八章 异常

一、什么是异常

python用异常对象（exception object）来表示异常情况。遇到错误后，会引发异常。如果异常对象并未被处理或捕捉，程序就会用所谓的回溯（traceback，一种错误信息）终止执行。

二、按自己的方式出错

1.raise语句

为了引发异常，可以使用一个类（应该是Exception的子类）或者实例参数调用raise语句。使用类时，程序会自动创建类的一个实例。下面的例子们使用了内建的Exception异常类：

raise Exception

raise Exception('Hello,world!')

🡺Traceback (most recent call last):

File ".\test.py", line 1, in <module>

raise Exception

Exception

Traceback (most recent call last):

File ".\test.py", line 2, in <module>

raise Exception('Hello,world!')

Exception: Hello,world!

第一个例子引发了一个没有任何有关错误信息的普通异常。后一个例子中，则添加了错误信息hello,world!

一些内建异常

|  |  |
| --- | --- |
| **类名** | **描述** |
| Exception | 所有异常的基类 |
| AttributeError | 特性引用或赋值失败时引发 |
| IOError | 试图打开不存在文件（包括其他情况）时引发 |
| IndexError | 序列中没有此索引(index) |
| KeyError | 映射中没有这个键 |
| NameError | 未声明/初始化对象 (没有属性) |
| SyntaxError | Python 语法错误 |
| TypeError | 对类型无效的操作 |
| ValueError | 传入无效的参数 |
| ZeroDivisionError | 除(或取模)零 (所有数据类型) |

2.自定义异常类

class SomeCustomException(Exception)：

pass

只是要确保从Exception类继承（直接，间接都行）

三、捕捉异常

try / except

try:

x = input()

y = input()

print(int(x)/int(y))

except ZeroDivisionError:

print("The second number can't be zero!")

🡺The second number can't be zero!

如果捕捉到了异常，但是又想重新引发它（也就是传递异常，不进行处理），那么可以调用不带参数的raise。考虑一下一个能“屏蔽”ZeroDivsionError的计算器类。如果这个行为被激活，那么计算器就会打印错误信息，而不是让异常传播。如果是在与用户进行交互的过程中使用，那么这就有用了，但是如果是在程序内部使用，引发异常会更好些。因此“屏蔽”机制就可以关掉了：

class MuffledCalculator:

muffled = False

def calc(self,expr):

try:

print( eval(expr))

except ZeroDivisionError:

if self.muffled:

print('Division by zero is illegal!')

else:

raise

calculator = MuffledCalculator()

calculator.calc('10/2')

calculator.calc('10/0')

calculator.muffled = True

calculator.calc('10/0')

🡺5.0

Traceback (most recent call last):

File ".\test.py", line 14, in <module>

calculator.calc('10/0')

File ".\test.py", line 5, in calc

print( eval(expr))

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

ZeroDivisionError: division by zero

四、不止一个except子句

class MuffledCalculator:

muffled = False

def calc(self,expr):

try:

print( eval(expr))

except ZeroDivisionError:

print("The second number can't be zero!")

except TypeError:

print("That wasn't a number,was is?")

这样就能检查类型异常了！

五、用一个块捕捉两个异常

class MuffledCalculator:

muffled = False

def calc(self,expr):

try:

print( eval(expr))

except (ZeroDivisionError,TypeError,NameError):

print("The second input is wrong!")

但这样只能打印一个错误信息。。。

六、捕捉对象

如果希望在except子句中访问异常对象本身，可以使用两个参数。下面的示例会打印异常，但是程序会继续运行：

class MuffledCalculator:

def calc(self,expr):

try:

print( eval(expr))

except (ZeroDivisionError,TypeError) as e:

print(e)

calua = MuffledCalculator()

calua.calc('10/0')

🡺division by zero

七、真正的全捕捉

class MuffledCalculator:

def calc(self,expr):

try:

print( eval(expr))

except:

print("wtf???!!!")

calua = MuffledCalculator()

calua.calc('10/2')

这样可以在except子句中忽略所有的异常类！像这样捕捉所有异常是危险的，因为它会隐藏所有程序员未想到并且未做好准备处理的错误。

八、万事大吉

有些情况，没有坏事发生时执行一段代码是很有用的，可以像对条件和循环语句那样，给try/except语句加个else子句：

try:

print('A simple task')

except:

print("What?Something went wrong?")

else:

print('It was ok!')

使用else子句可以实现循环：

while True:

try:

x = input()

y = input()

value = int(x) / int(y)

print(value)

except:

print("What?Something went wrong?")

else:

break

九、最后……

finally可以用来在可能的异常后进行清理。不管try子句中是否发生异常，finally子句肯定会被执行。

十、异常和函数

如果异常在函数内引发而不被处理，它就会传播至函数调用的地方，如果在那里也没有处理异常，它就会继续传播，，一直到达主程序（全局作用域），如果在那里也没有异常处理程序，程序就会带着栈跟踪中止。

第九章 魔法方法、属性和迭代器

一、构造方法

构造方法和其他普通方法的不同的地方在于：当一个对象被创建后，会立即调用构造方法。

class FooBar:

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

self.somevar = 42

f = FooBar()

print(f.somevar)

🡺42

当使用参数时，

class FooBar:

def \_\_init\_\_(self,value=42):

self.somevar = value

f = FooBar('Hello,world!')

print(f.somevar)

🡺Hello,world!

1.重写一般方法和特殊的构造方法

每个类都可能拥有一个或多个超类，它们从超类那里继承行为方式。如果一个方法在B类的一个实例中被调用，但在B类中没有找到该方法，那么就会去它的超类A里面找。

class A:

def hello(self):

print('Hello,I am A!')

class B(A):

pass

a = A()

b = B()

a.hello()

b.hello()

🡺Hello,I am A!

Hello,I am A!

在子类中增加功能的最基本的方法就是增加方法，但是也可以重写一些超类的方法来自定义继承的行为：

class A:

def hello(self):

print('Hello,I am A!')

class B(A):

def hello(self):

print("Hello,I am B!")

a = A()

b = B()

a.hello()

b.hello()

🡺Hello,I am A!

Hello,I am B!

当处理构造方法比重写普通方法时，更可能遇到特别的问题：如果一个类的构造方法被重写，那么就需要调用超类的构造方法，否则对象可能不会被正确地初始化：

class Bird:

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

self.hungry =True

def eat(self):

if self.hungry:

print('Aaaah...')

self.hungry = False

else:

print("No,thanks!")

b = Bird()

b.eat()

b.eat()

🡺Aaaah...

No,thanks!

class Bird:

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

self.hungry =True

def eat(self):

if self.hungry:

print('Aaaah...')

self.hungry = False

else:

print("No,thanks!")

class SongBird(Bird):

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

self.sound = "Squawk"

def sing(self):

print(self.sound)

sb = SongBird()

sb.eat()

🡺Traceback (most recent call last):

File ".\test.py", line 18, in <module>

sb.eat()

File ".\test.py", line 5, in eat

if self.hungry:

AttributeError: 'SongBird' object has no attribute 'hungry'

错误在于：SongBird没有hungry特性。原因在于：在SongBird中，构造方法被重写，但新的构造方法没有任何关于初始化hungry特性的代码。为了达到预期的效果，SongBird的构造方法必须调用其超类Bird的构造方法来确保进行基本的初始化。有两种方法能达到这个目的：调用超类构造方法的未绑定版本或者使用super函数。

3.调用未绑定的超类构造方法

虽然目前版本的python中，使用super函数会更为简单明了，但是很多遗留代码还会使用这种方法，所以还是有必要了解一下。

具体做法：

class SongBird(Bird):

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

Bird.\_\_init\_\_(self)🡨--------

self.sound = "Squawk"

def sing(self):

print(self.sound)

道理在于：在调用一个实例的方法时，该方法的self参数会被自动绑定到

实例上（这叫做绑定方法），但如果直接调用类的方法，那么就没有实例会被绑定，这样就可以自由地提供需要的self参数（这叫做未绑定方法）。

3.使用super函数

当前的类和对象可以作为super函数的参数使用，调用函数返回的对象的任何方法都是调用超类的方法，而不是当前类的方法。

class SongBird(Bird):

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

super(SongBird,self).\_\_init\_\_()🡨------

self.sound = "Squawk"

def sing(self):

print(self.sound)

原理暂时不必深究（反正也搞不懂）！

三、成员访问

1.基本的序列和映射规则

序列和映射是对象的集合，为了实现它们基本的行为（规则），如果对象是不可变的，那么就需要使用两个魔法方法，如果是可变的则需要使用4个。

\_\_len\_\_(self)：这个方法应该返回集合中所含项目的数量。对于序列来说，这就是元素的个数；对于映射来说，则是键值对的数量。如果\_\_len\_\_返回0（并且没有实现重写该行为的\_\_nonzero\_\_），对象会被当作一个布尔变量中的假值进行处理。

\_\_getitem\_\_(self,key)：这个方法返回与所给键对于的值。对于一个序列，键应该是一个0~n-1的整数，n是序列的长度；对于映射来说，可以使用任意种类的键。

\_\_setitem\_\_(self,key,value)：这个方法应该按一定的方式存储和key相关的value，该值随后可以使用\_\_getitem\_\_来获取。当然，只能为可以修改的对象定义这个方法。

\_\_delitem\_\_(self,key)：这个方法在对一部分对象使用del语句时被调用，同时必须删除和键相关的键。这个方法也是为可修改的对象定义的。

对于这些方法的附加要求如下：

（1）对于一个序列来说，如果键是负数，那么要从末尾开始计数。换句话说就是x[-n]和x[len(x)-n]是一样的。

（2）如果键是不合适的类型（例如对序列使用字符串作为键），会引发一个TypeError异常

（3）如果序列的索引是正确的类型，但超出了范围，应该引发一个IndexError异常。

算术序列实现代码：

def checkIndex(key):

if not isinstance(key,int):raise TypeError

if key < 0:raise IndexError

class ArithmeticSequence:

def \_\_init\_\_(self,start=0,step=1):

self.start = start

self.step = step

self.changed = {}

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

checkIndex(key)

try:

return self.changed[key]

except KeyError:

return self.start + key\*self.step

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

checkIndex(key)

self.changed[key] = value

2.子类化列表，字典和字符串

如果类的行为和默认的行为很接近，子类化内建类型就很有用。

class CounterList(list):

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

super(CounterList,self).\_\_init\_\_(\*args)

self.couter = 0

def \_\_getitem\_\_(self,index):

self.counter += 1

return super(CounterList,self).\_\_getitem\_\_(index)

CounterList在很多方面和列表都一样，但它在每次列表元素被访问时，它都会自增！

四、属性

class Rectangle:

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

self.width = 0

self.height = 0

def setSize(self,size):

self.width,self.height = size

def getSize(self):

return self.width,self.height

r = Rectangle()

r.width = 10

r.height = 5

print(r.getSize())

r.setSize((150,100))

print(r.width)

🡺(10, 5)

150

getSize和setSize方法就是一个名为size的假想特性的访问器方法。通过访问器定义的特性被称为属性。

1.property函数

class Rectangle:

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

self.width = 0

self.height = 0

def setSize(self,size):

self.width,self.height = size

def getSize(self):

return self.width,self.height

size = property(getSize,setSize)

r = Rectangle()

r.width = 10

r.height = 5

print(r.size)

r.size = (150,100)

print(r.width)

🡺(10, 5)

150

很明显，size特性仍然取决于getSize和setSize中的计算，但它们看起来就像普通的属性一样。实际上，property函数还可以用0,1,3或4个参数来调用。如果没有参数，产生的属性既不可读，也不可写。如果只使用一个参数调用（一个取值方法），产生的属性是只读的。第3个参数是一个用于删除特性的方法。第4个参数是一个文档字符串。property的4个参数分别被叫做fget，fset，fdel和doc，如果想要一个属性是只写的，并且有一个文档字符串，可以使用关键字参数。

2.静态方法和类成员方法

静态方法和类成员方法分别在创建时分别被装入staticmethod类型和classmethod类型的对象中。静态方法的定义没有self参数，且能够被类自身直接调用。类方法在定义时需要名为cls的类似于self的参数，类成员方法可以直接用类的具体对象调用，但cls参数是自动被绑定到类上的：

class MyClass:

def smeth():

print('This is a static method')

smeth = staticmethod(smeth)

def cmeth(cls):

print('This is a class method of',cls)

cmeth = classmethod(cmeth)

装饰器：它能对任何可调用的对象进行包装，既能够用于方法也能用于函数。使用@操作符，在方法（或函数）的上方将装饰器列出，从而指定一个或更多的装饰器（多个装饰器在应用时的顺序与指定顺序相反）。

class MyClass:

@staticmethod

def smeth():

print('This is a static method')

@classmethod

def cmeth(cls):

print('This is a class method of',cls)

定义了这些方法后，就可以这样使用（例子中没有实例化）：

MyClass.smeth()

MyClass.cmeth()

🡺This is a static method

This is a class method of <class '\_\_main\_\_.MyClass'>

3.\_\_getattr\_\_、\_\_setattr\_\_和它的朋友们

拦截对象的所有特性访问是可能的，这样可以用旧式类实现属性。

\_\_getattribute\_\_(self,name)：当特性name被访问时自动被调用（只能在新式类中使用）

\_\_getattr\_\_(self,name)：当特性name被访问且对象没有相应的特性时被自动调用。

\_\_setattr\_\_(self,name,value)：当试图给特性name赋值时会被自动调用。

\_\_delattr\_\_(self,name)：当试图删除特性name时被自动调用。

尽管和使用property函数相比有点复杂，但这些特殊方法是很强大的！

class Rectangle:

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

self.width = 0

self.height = 0

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

if name =='size':

self.width,self.height = value

else:

self.\_\_dict\_\_[name] = value

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

if name == 'size':

return self.width,self.height

else:

raise AttributeError

注：python的面对对象编程还是去看专门的书吧。。。

五、迭代器

\_\_iter\_\_这个方法是迭代器规则的基础

1.迭代器规则

实际上，只要对象实现了\_\_iter\_\_方法，就能对其进行迭代。\_\_iter\_\_方法会返回一个迭代器(iterator)，所谓的迭代器就是具有next方法（这个方法在调用时不需要任何参数）的对象。在调用next方法时，迭代器会返回它的下一个值。如果next方法被调用，但迭代器没有值可以返回，就会引发一个StopIteration异常。

正式的说法是，一个实现了\_\_iter\_\_方法的对象是可迭代的，一个实现了next方法的对象是迭代器。

class Fibs:

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

self.a = 0

self.b = 1

def \_\_next\_\_(self):🡸python3中得这么写了！

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

return self.a

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

return self

fibs = Fibs()

for f in fibs:

if f > 1000:

print(f)

break

内建函数iter可以从可迭代的对象中获得迭代器

it = iter([1,2,3])

print(next(it))

print(next(it))

print(next(it))

print(next(it))

🡺1

2

3

Traceback (most recent call last):

File ".\test.py", line 5, in <module>

print(next(it))

StopIteration

2.从迭代器得到序列

list（iterator）

六、生成器（略）

我觉得书上的这一节讲的很不好。。。