Python编程规范

整理：李文卓

时间：2021/11/1

参考资料：

第一章 引论

1. 注释

python有三种形式的注释：块注释、行注释以及文档注释

1）块注释

对于复杂的操作，应该在其操作开始前写上若干行注释，解释复杂的操作、算法，难以理解的技巧。

注：为了提高可读性，注释应该至少离开代码2个空格

2） 文档注释（docstring)

为函数和方法添加文档注释。注释内容：方法的功能、参数、返回值、以及可能的异常进行说明

一个文档字符串应该这样组织：首先一行是以句号，问号或者惊叹号结尾的概述；接着是一个空行，接着是文档字符串剩下的部分，应该与文档字符串的第一行的第一个引号对齐。

3）代码类型注释

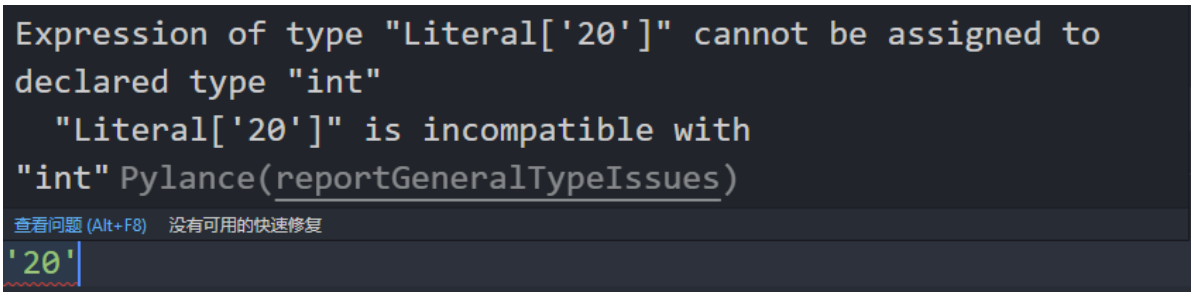
python 是动态语言，在声明变量时，不需要显式声明变量的类型。

但是如果代码中某些变量的类型有错，编辑器等工具无法在早期替你纠错，只能在程序运行阶段才暴露问题。因此，python3.5之后引入了类型注释，作用就是明确的声明变量的类型。

a.变量注释：

在 VS Code 中安装插件 Pylance，开启Pylance设置中的类型检查开关即可。

```
age: int = 20
age = '20'
```



b.函数注释

注释函数参数和返回值的类型

```
def say_hi(name: str) -> str:
    return 'hello '+name
```

可以清楚的看出，这个函数应该接收一个**字符串**参数 name，并且返回值也应该是**字符串**

c.容器类型

列表，字典，元组等包含元素的复合类型，用简单的 list，dic，tuple不能明确说明内部元素的具体类型。

因此要用到typing模块提供的**复合注释**功能。（python3.9以上版本不需要,内置的容器类型就支持复合注解）

```
def min(scores: list[int],ages: dict[str,int]) -> tuple[int,int]:
    return (0,0)
```

4）TODO注释

TODO注释应该在所有开头处包含“TODO”字符串，紧跟着是用括号括起来你的名字，email地址或者其他标识符，然后是一个可选的冒号，接着必须有一行注释，解释要做什么。

```
# TODO(kl@gmail.com): Use a "*" here for string repetition.
# TODO(Zeke) Change this to use relations.
```

2. 空行

在一组代码表示完一个完整的思路之后，应该用空白行进行间隔。

做法：在函数定义或者类定义之间空两行；类定义与第一个方法之间、或需要进行语义分割之间空一行。

2.1 避免过长的代码行

每行最好不要超过80字符，以每屏能够显示完整代码而不需要拖动滚动条为佳。

做法：超过的部分用圆括号、方括号、花括号进行行连接，并且保持行连接的元素垂直对齐。

```
x1=('this is a very long string,'
    'it is used for testing line limited characters')
>>> this is a very long string,it is used for testing line limited characters
```

2.2 空格的警示作用

二元运算符、布尔运算的两边应该有空格

二元运算符：赋值（=），比较（==,<,>,!=,<>,<=,>=,in,not in ,is,is not)

布尔运算：and,or,not

3.编写函数的原则

3.1 函数设计尽量短小，嵌套层次不宜过深。

尽量避免过长的函数，不需要来回翻动屏幕寻找某个变量或者某条逻辑判断等，涉及到的循环，条件判断最好控制在3层以内。

3.2 函数申明做到合理、简单、易于使用。

函数名应该正确反映大致功能，参数的设计也应该简单明了，参数个数不应该过多。

3.3 函数参数的设计应该考虑向下兼容。

通过加入默认参数避免函数调用的接口发生改变。

3.4 一个函数只做一件事，保证函数语句粒度的一致性。

粒度？（待补）

第二章 编程惯用法

1.assert语句

assert（断言）语句为调试程序而服务，能够快速方便的检查程序中的异常或者不恰当的输入。

2. 转换为浮点数再做除法

3.enumerate()

解决循环中中获取序列及其对应值的问题，

函数用法为：enumerate(sequence,start=0),其中sequence可以是list、set

```
li = ['a','b','c']
for i,e in enumerate(li):
    print('index:',i,'element:',e)
>>>
index: 0 elements: a
index: 1 elements: b
index: 2 elements: c
```

4.分清 == 和 is

例子：

```
a = 'hi'
b = 'hi'
print(a is b)
print(a == b)
>>>
True
True
```

```
x = 'i am using long string for testing.'
y = 'i am using long string for testing.'
print(x is y)
print(x == y)
>>>
False
True
```

现象：从上注意到对于短字符串而言，is和==的输出都是True；而对于长字符串而言，is的输出为False，==的输出为True,为什么？

解释：通过id()函数查看变量在内存中具体的存储空间。

```
print(id(a))
print(id(b))
>>>
2626237865136
2626237865136
# 指向的id地址相同
```

```
print(id(x))
print(id(y))
>>>
2626313276080
2626313273776
# 指向的id地址不同
```

注：**字符串驻留机制**：对于较小的字符串，为了系统性能会保留该值的一个副本，当创建新的字符串时，直接指向该副本即可，因此第一个例子中'hi'在内存中只有一个副本，所以a和b的内存值一样。而 x 和 y 是长字符串，并不会**驻留**，python在内存中创建了两个对象。

结论：is 是对象标识符（object identity），== 的意思是相等，两者的区别在于：is 的作用是比较两个对象在内存中是否拥有同一块内存空间，并不适用判断两个字符串是否相等；== 则是检验两个对象的值是否相等。一般情况下，如果 x is y 为True，则 x == y 的值也为True，反之则不然。

5.True/False的求值

尽可能使用隐式False

所有的"空值"都被认为是false，例如：0，None，[]，{}，""

```
# Yes
if foo:
    ...
# No
if foo != []:
    ...
```

6.Unicode编码

背景：

1）最早的ACSII编码用一个字节表示128个字符（英文大小写，数字，其他符号），但是显然128个字符数远远不够

2）于是出现了不同的字符编码系统，比如汉字编码GBK，但是不同的编码系统之间存在冲突。比如：在两种不同的编码系统中，相同的编码可能代表不同的意义或不同的编码代表相同的字符，因此不同语言的文本无法进行很好的转换。

3）为了解决以上冲突，必须为不同的文字分配统一编码，Unicode（Universal Multiple-Octet Code Character Set)由此产生，被称为："万国码"，为每种语言提供了唯一的二进制编码表示方式，提供从数字代码到不同语言字符集之间的映射，可以满足跨平台、跨语言之间的文本处理要求。

层次：

Unicode编码系统分为编码方式和实现方式。

1）编码方式：分为 UCS-2 和 UCS-4 两种方式。

2）实现方式：Unicode转换格式（Unicode Transformation Format），简称 UTF，包括UTF-7,UTF-16,UTF-32,UTF-8，其中 UTF-8 最为常见。

例子：

文件test.txt用UTF-8形式保存，但是windows系统中默认编码为CP936，在windows系统中被映射为GBK编码，两种编码方式并不兼容，出现了错误。解决：首先对读入的字符用UTF-8进行解码，然后用GBK进行编码。

解决措施：

```
with open('test.txt',encoding='utf-8') as f:
    print(f.read())
>>>python中文测试
```

注：

python默认编码方式：'utf-8'，通过sys.getdefaultencoding()查看。

7.Shebang

Shebang 是由一个**井号**和**叹号**构成的字符串串行，出现在文本文件的第一行的前两个字符。

#! 先用于帮助内核找到Python解释器，但是在导入模块时，将会被忽略，因为只有被直接执行的文件中才有必要加入#!

因此程序的main文件应该以 #!/usr/bin/python3开始

源文件编码声明：

```
#! /usr/bin/python
# -*- coding: <encoding name>
```

8.全局变量

全局变量为在模块级的变量，尽量避免全局变量，因为导入模块时会对模块级变量赋值。

但是鼓励使用全局常量，命名全部大写，用 _分隔，例如：

```
MAX_COUNT = 3
```

第三章 基础语法

1.with 自动关闭资源

例子：

```
f = open('test1.txt','w')
f.write('test')
```

此时 test1.txt 文件中是空，没有写入任何字符串。所以对文件进行操作后应该立即关闭他们。

```
with open('test.txt','w') as f:
    f.write('test')
```

2.理解None

以下数据会当作**空**来处理：常量None，常量False，任何形式的数值类型为0，空的序列、空的字典

其中常量None的特殊性体现在它既不是0、False、也不是空字符串，就是一个空值对象，其中数据类型为NoneType，与其他 非None 的对象的比较结果都是False

3.字符串的基本用法

1）多行字符串

python特性：遇到未闭合的小括号会自动将多行代码拼为一行和把相邻的两个字符串拼接在一起

```
s = ('select * '
    'from atable '
    'where afield="value"')
s
>>>select * from atable where afield="value"
```

2）判断字符串类型

```
a = 'hi' # u:unicode
isinstance(a,str)
>>>True
```

3）性质判定

常用方法有：isalnum( )、isalpha( )、isdigit( )、islower( )、isupper( )、issapce( )、istitle( )、startswith( )、endswith( )

```
'Hello world'.istitle() # 判定字符串中的每个单词是否都有且只有第一个字母是大写的
>>>False
```

```
# str.startswith(str, beg,end) beg:检测的起始位置，end:检测的结束位置
str = "this is string example"
str.startswith('is',2,4)
>>> True
```

4）查找替换

count( )、find( )、index( )、sub( )、rfind( )、rindex( )

count( )能够查找子串sub在字符串中出现的次数

find( ) 函数族找不到时返回-1，index( ) 函数族抛出ValueError异常，但是对于判定是否包含子串推荐使用 in 和 not in 操作符

```
str1 = 'test if a string contains some special substrings'
if str1.find("some") != -1:
    print('yes,it contains')
>>>yes,it contains
```

replace(old，new，max) ,参数max代表最多替换max次

```
str1 = "this is string example,this is really string"
str1.replace("is", "was")
>>>'thwas was string example,thwas was really string'
```

5）分切与连接

partition( ) 用来根据指定的分隔符将字符串进行分割。

如果字符串包含指定的分隔符，则返回一个3元的元组，第一个为分隔符左边的子串，第二个为分隔符本身，第三个为分隔符右边的子串。

语法：str.partition(str)，参数str为指定的分隔符

```
str1 = 'www.runoob.com'
str1.partition('.')
>>>('www', '.', 'runoob.com') # 为什么runoob和com之间没有分隔开
```

split( )

语法：

str.split(sep, num) ,参数sep为分隔符，默认为所有的空字符（空格，换行，制表符）； num为分割次数，默认为-1，分割所有。

需要注意的陷阱：

```
print('  hello   world  '.split())
print('  hello   world  '.split(' '))
>>>
['hello', 'world']
['', '', 'hello', '', '', 'world', '', '']
```

产生以上差异的原因：1.当忽略seo参数或者sep参数为None时，与明确给sep赋予字符串值，split( )将采用两种不同的算法。 2. 对于前者，split( )首先去除字符串两端的空白符，然后以任意长度的空白符为sep切分字符串。 3.对于后者认为两个连续的sep之间存在一个空字符串。

6）变形

lower( )、upper( )、capitalize( )、swapcase( )、title( )

注意：

title( ) 将每一个单词的首字母大写，并将单词中**非首字母**转换为**小写**

```
print('hello wORLD'.title())
>>>Hello World
```

swapcase() 将字符串中的英文字母大小写互换，并返回修改后的字符串。

```
old = 'Python'
new = old.swapcase()
new
>>>'pYTHON'
```

7）删减与填充

删减就是把字符串**掐头去尾**

strip() 、lstrip() 、rstrip()

strip() 方法用于移除字符串头尾指定的字符（默认为空格或换行符）或字符序列。

```
str1 = "00000003210Runoob01230000000"
str2 = 'hello\n'
print(str1.strip('0'))
print(str2.strip())
>>>
3210Runoob0123
hello
```

填充常用于字符串的输入，借助他们排出漂亮的版面。

center( )、ljust( )、rjust( ) 、zfill( ) 、expandtabs( )

str.center(width, fillchar) 指定一个长度值，将原字符串在该长度内居中。可以指定字符为填充字符，默认情况下为空格。

```
demo = 'python'
demo.center(15,'-')
>>>'-----python----'
```

4.连接字符串的 join 方法 和 +

例子：

```
string1 = 'this '
string2 = 'is'
string3 = ' a test'
print(string1 + string2 + string3)
print(''.join([string1,string2,string3])) #先转成列表
>>>
this is a test
this is a test
```

思考：以上两种字符串连接的方式除了形式上的区别，在性能上是否有差异呢?

结论：join()方法的效率明显比 + 操作符要高，特别是当字符串规模比较大的时候，join( ) 的优势更为明显。

用+字符串连接 s1+s2+s3+... 时，每执行一次 +操作符 便会在内存中申请一块新的内存空间，将上一步的结果和这一步的数字复制到新的内存空间中，以此类推。

不同的是，join()方法连接字符串会首先计算需要申请的总的内存空间，然后一次性申请所需内存并将字符串序列中的每一个元素复制到内存中去，因此时间复杂度为O(n)

5.格式化字符串的 .format 方法

语法：

{<参数序号>：<填充><对齐><宽度>,<.精度><类别>}

填充符：除了'{'和'}'之外的任意符号

对齐方式：

<	**左对齐（观察箭头的方向）**
>	右对齐（数值默认的对齐方式）
=	仅对数值类型有效，如果有符号的话，在符号后数值前进行填充
^	居中对齐，使用空格进行填充

```
# 1.使用位置符号
'the number of {0:,} in hex is:{0:#x},the number of {1} in oct is
{1:#o}'.format(4746,45)
>>>'the number of 4,746 in hex is:0x128a,the number of 45 in oct is 0o55'
```

```
# 2.使用名称
'the max number is {max},the min number is {min},the average number is
{average:0.3f}'.format(max=189,min=12.6,average=23.5)
>>>'the max number is 189,the min number is 12.6,the average number is 23.500'
```

6.集合set

python中集合是通过Hash算法实现的无需不重复的元素集，创建集合通过set( )方法实现。

```
set('hello')
>>>{'e', 'h', 'l', 'o'}
```

```
# 将列表转换为set
a = [1,2,'34',(5,6)]
set(a)
>>>{(5, 6), 1, 2, '34'}
```

常见操作：

1.m.union(n) 集合m和n的并集

```
m = set(['a','b','c'])
n = set(['a','d','e'])
m.union(n)
>>>{'a', 'b', 'c', 'd', 'e'}
```

2. m.intersection(n) 集合m和n的交集

```
m.intersection(n)
>>>{'a'}
```

3. m.difference(n) 集合m和n的差集，在m中存在但是在n中不存在的元素组成的集合。

```
m.difference(n)
>>>{'b', 'c'}
```

4.m.symmetric_difference(n)

```
m.symmetric_difference(n)
>>>{'b', 'c', 'd', 'e'}
```

优势：

set在某些操作上明显比list更高效，特别在union，intersection，difference等操作要比list的迭代更快。因此list涉及到求交集，并集，或者差集的问题可以转换为set来操作

7.分清可变对象和不可变对象

划分依据：根据值是否可以修改分为**可变对象**和**不可变对象**

可变对象：字典、列表、字节数组

不可变对象：数字、字符串、元组

```
list1 = ['a','b','c']
list2 = list1
list1.append('d')
list2
>>>
['a', 'b', 'c', 'd'] # list2 的值也发生了变化
```

```
list1 = ['a','b','c']
list3 = list1[:] # 切片操作相等于浅拷贝
list3.remove('b')
list1
>>>['a', 'b', 'c'] # list1 的值没有发生变化
```

```
print(id(list1))
print(id(list2))
print(id(list3))
>>>
2421839538368
2421839538368
2421839533312
```

结论：对 list1 的切片操作相等于浅拷贝，会重新生成一个对象，对 list3 的操作不会影响 list1 和 list2

对于不可变对象：

```
a = 1
print(id(a))
a += 2
print(id(a))
print(id(3))
>>>
140709922604704
140709922604768
140709922604768
```

虽然 a 是数值类型，是不可变对象，但是python中变量 a 存放的是数值1在内存中的地址，而数值1本身才是不可变对象。

所以上面的过程改变的是a所指向的对象的地址，数值1并没有发生改变。

8.列表解析（推导式&生成式）

例子：遍历列表的每个元素后，将每个单词包含的空格去掉，首字母大写的元素生成新的列表

```
words = [' Are',' abandon','Passion','Business','fruit','quit']
newlist = []
for i in words:
    if i.strip().istitle():
        newlist.append(i)
print(newlist)
>>>[' Are', 'Passion', 'Business']
```

更好的实现方式：**列表解析**

语法：[expr for iter_item in iterable if cond_expr]，迭代 iterable 的每一个元素，当条件满足的时候根据表达式expr计算的内容生成一个元素并放入新的列表中，以此类推，最终返回整个列表。

注意：1）如果没有条件语句，就直接将 expr 计算出的元素加入 list 中；2）适用于简单情况，禁止多重for语句或者过滤器表达式，复杂情况下还是应该使用循环。

```
# Yes:
result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))
# No：禁止多重for语句
result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]
```

实现方式：

```
newlist2 = [i for i in words if i.strip().istitle()]
newlist2
>>>[' Are', 'Passion', 'Business']
```

更为灵活的功能：

1）多重嵌套

在二位列表中将每个单词全部转换为大写

```
nested_list = [['Hello','World'],['Goodbye','Word']]
nested_list = [[s.upper() for s in xs] for xs in nested_list]
nested_list
>>>[['HELLO', 'WORLD'], ['GOODBYE', 'WORD']]
```

2）多重迭代

```
print([(a,b) for a in ['a','1',1,2] for b in ['1',3,4,'b'] if a != b ],end='')
>>>[('a', '1'), ('a', 3), ('a', 4), ('a', 'b'), ('1', 3), ('1', 4), ('1', 'b'),
(1, '1'), (1, 3), (1, 4), (1, 'b'), (2, '1'), (2, 3), (2, 4), (2, 'b')]
```

3）列表解析中的表达式既可以是简单表达式、复杂表达式、甚至是函数。

```
def f(v):
    if v%2 == 0:
        v = v**2
    else:
        v=v+1
    return v
[f(v) for v in [2,3,4,-1] if v > 0] # expr:f(v)
>>>[4, 4, 16]
```

4）iterable 可以是任意的可迭代对象，比如文件。

```
f = open('test.txt','r',encoding='utf-8')
result = [i for i in f if 'abc' in i]
print(result)
```

优势：

使用列表解析更为直观清晰，代码更为简洁。

列表解析的效率更高

9.list( ) 和 tolist( )

1.tolist()将numpy数组或者矩阵转换为列表

```
import numpy as np

x_tolist = x.tolist()
print(x_tolist)
>>>[[1, 2], [3, 4]]
```

2.list()方法用于将元组转化为列表

```
atuple = (1,2,3,'a')
alist = list(atuple)
>>>alist
[1, 2, 3, 'a']
```

10.条件表达式（三元运算符）

对于 if 语句的一种更为简短的语法规则

```
x = 1 if condition else 2
```

11.默认迭代器和操作符

果类型支持，使用默认迭代器和操作符。

默认操作符和迭代器简单高效，直接表达了操作，没有额外的方法调用。

```
# Yes:
for key in adict:
    ...
for line in afile:
    ...
#No:
for key in adict.keys():
    ...
for line in afile.readlines():
    ...
```

12.lambda函数

lambda在一个表达式中定义匿名函数，常用于map()和filter()。

缺点：比起本地函数更难阅读和调试，并且lambda函数只包括一个表达式，所以表达能力有限。

```
>>> list(filter(lambda x:x % 2 == 1,[1,2,3,4,5,6]))
[1, 3, 5]
```

13. 区分sort( )和sorted( )

用法：

1.list.sort(reverse=True|False, key=myFunc)

2.sorted(iterable, cmp=None, key=None, reverse=False)

3.key是带一个参数的函数，用来为每个元素提取值，默认为None

区别：

1.sorted( ) 的使用范围更广，sorted( ) 作用于任意可迭代的对象，sort( ) 作用于列表。

2.返回类型不同。sorted( ) 会返回一个排序之后的列表，原有的列表保持不变；sort( ) 函数会直接修改原有列表，函数返回值为None。因此实际过程中如果需要保留原有列表，使用sorted( )函数。

3.两个函数中，传入参数key比传入参数cmp效率要高很多。key针对每个元素仅做一次处理。

14.警惕默认参数

在函数调用没有指定与形参对应的实参时就会自动使用默认参数，默认参数给函数的使用带来了更大的灵活性。

默认参数只在模块加载时求值一次，如果参数是列表或者字典之类的可变类型，这可能会导致问题。因此，不要在函数或者方法定义中使用“可变对象”作为默认值。

例子：

```
def append_test(item,lista=[]):
    print(id(lista))
    lista.append(item)
    print(id(lista))
    print(lista)
append_test('a')
append_test(1)
>>>['a', 1]
```

连续两次调用 append_test(1) 和 append_test('a') ，期望函数的返回值应该是[1]和['a']，但是实际情况却输出了[1]和[1,'a']

原因：在解释器执行 def 时，默认参数也会被计算，首次调用 append_test('a') 时，[ ] 变为 ['a'],但是再次调用时，默认参数不会被重新计算，于是在 ['a'] 的基础上变为了 ['a',1]

解决方法：在函数定义时使用None作为占位符。

```
def append_test(item,lista=None):
    if lista == None:
        lista = []
    print(id(lista))
    lista.append(item)
    print(id(lista))
    print(lista)
```

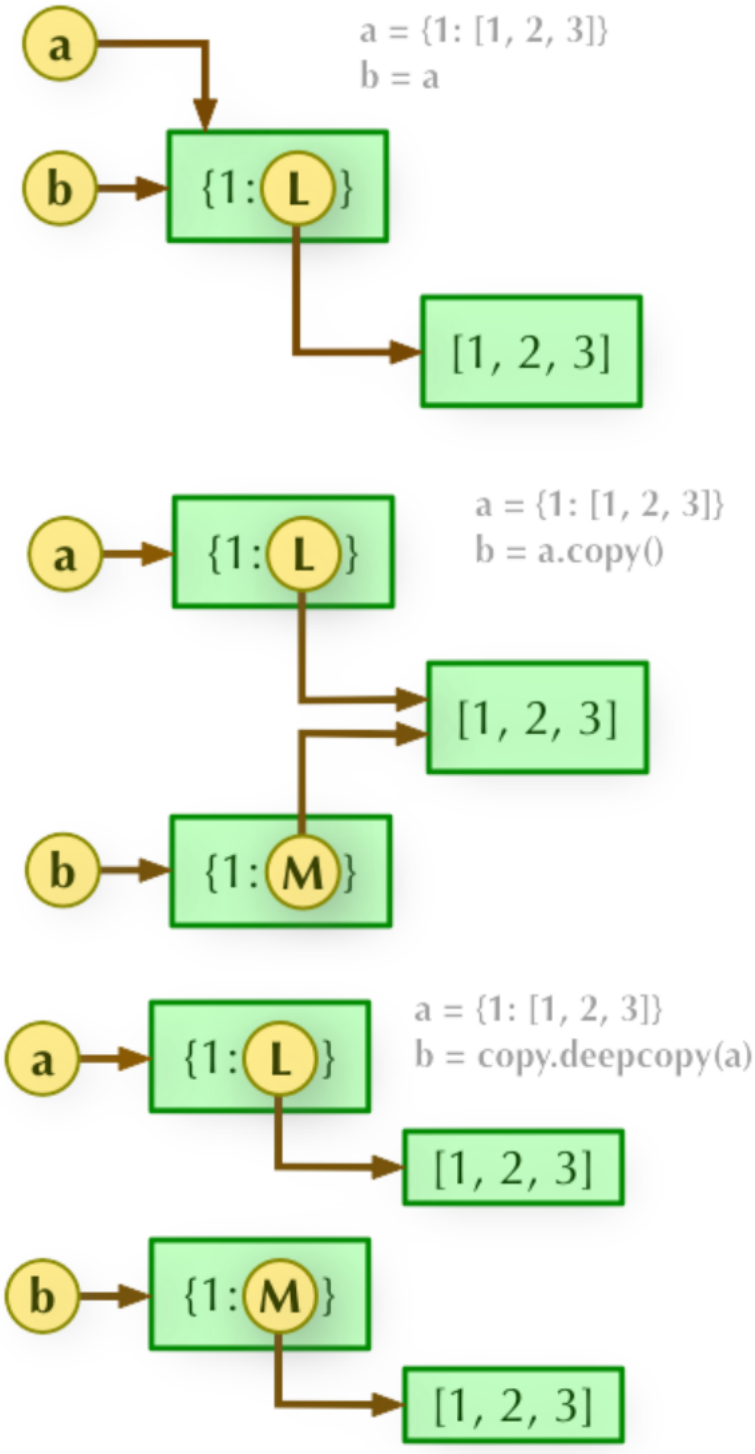
15.区分直接赋值、浅拷贝、深拷贝

1）直接赋值：就是对象的引用（别名）；

2）浅拷贝：拷贝父对象，但是不会拷贝对象内部的子对象；

3）深拷贝：copy模块的deepcopy方法，完全拷贝了父对象和子对象。

图解：



直接赋值是引用对象；浅拷贝：深拷贝父对象（一级目录），但是引用子对象（二级目录）。

```
# 直接赋值与浅拷贝
dict1 = {'user':'runoob','num':[1,2,3]}

dict2 = dict1
dict3 = copy.copy(dict1)

dict1['user'] = 'root'
>>>print(dict1)
>>>print(dict2)
>>>print(dict3)
{'user': 'root', 'num': [1, 2, 3]}
{'user': 'root', 'num': [1, 2, 3]}
{'user': 'runoob', 'num': [1, 2, 3]}     #浅拷贝的父对象（一级目录）没有随之改变
```

```
# 直接赋值与浅拷贝
dict1['num'].remove(1)  #改变子对象的值
>>>print(dict1)
>>>print(dict2)
>>>print(dict3)
{'user': 'root', 'num': [2, 3]}
{'user': 'root', 'num': [2, 3]}
{'user': 'runoob', 'num': [2, 3]}   #浅拷贝引用子对象，所以子对象的值随之改变
```

第四章 库

1.python中模块，包，库的概念

1）模块：就是.py文件，里面定义了一些函数和变量，需要的时候就可以导入这些模块。

2）包：在模块之上的概念，为了方便管理而将文件进行打包。包目录下第一个文件便是*__init__.py*，有了这个文件，我们才能导入这个目录下的 *module*。然后是一些模块文件和子目录。假如子目录中也有_init__.py，那么它就是这个包的子包了。

常见的包结构：

```
package_a
├── __init__.py
├── module_a1.py
└── module_a2.py
```

3）库：具有相关功能模块的集合。这也是Python的一大特色之一，即具有强大的标准库、第三方库以及自定义模块。标准库：就是下载安装的python里那些自带的模块，要注意的是，里面有一些模块是看不到的比如像sys模块；第三方库：就是由其他的第三方机构，发布的具有特定功能的模块。python中真正使用的是包和模块，库是抽象意义上的统称。

2.模块导入

尽量使用 import a 的形式

有节制的使用from a import b的形式

尽量避免 from a import * 的形式，因为会污染命名空间，并且无法清晰表示导入了哪些对象。

以下几种情形可以考虑使用 from ...import...

1）只需要导入部分属性和方法时

2）模块中方法和属性的访问频率过高，导致使用 模块名.名称 的形式进行访问过于繁琐时。

3.Counter计数统计

模块 collections 实现了专门的容器数据类型，提供了Python的通用内置容器dict、list、set和tuple的替代。

1.1.统计某一项出现的次数可以使用 dict 字典进行存储，但是更简单的方法是使用 collections.Counter(),

```
from collections import Counter
some_data = ['a','2',2,4,5,'2','b',4,7,'a',5,'d','a','z']
Counter(some_data)
>>>Counter({'a': 3, '2': 2, 2: 1, 4: 2, 5: 2, 'b': 1, 7: 1, 'd': 1, 'z': 1})
```

```
Counter('success')
>>>Counter({'s': 3, 'u': 1, 'c': 2, 'e': 1})
```

1.2. 使用elements( ) 方法获取Counter中的key值。

```
list(Counter(some_data).elements())
>>>['a', 'a', 'a', '2', '2', 2, 4, 4, 5, 5, 'b', 7, 'd', 'z']
```

3.3. 使用 most_common(N) 方法找出前N个出现频率最高的元素以及他们对应的次数

```
Counter(some_data).most_common(2)
>>>[('a', 3), ('2', 2)]
```

4.序列化的选择：JSON(JavaScript Object Notation)

一种轻量级的**数据交换**格式。易于人阅读和编写。同时也易于机器解析和生成。

标准库json

1.1 常用方法是dump/dumps 用来序列化，把python对象encode为json对象。dump输出到文件，dumps（s:string）输出到一个字符串

load/loads 用来反序列化

```
import json
data = {
    'name':'Alice',
    'shares':100,
    'price':542
}
json_str = json.dumps(data)
data = json.loads(json_str)
```

如果你要处理的是文件而不是字符串，你可以使用 json.dump() 和 json.load()来编码和解码 JSON 数据。

```
with open('data.json','w') as f:
    json.dump(data,f)
with open('data.json','r') as f:
    data = json.load(f)
```

5. sys 模块

待补充

6.os模块

待补充

第五章 内部机制

1. 名字查找机制

python中所谓的变量都是名字，这些名字指向一个或者多个python对象。

例子：

```
a = 1
b = a
id(a) == id(b)
>>>True
```

可以看出，变量 a 和 b 指向同一个python对象，对象的值为1.

所有的变量都存在于一个表（命名空间）里，一般情况下称之为局部变量，通过locals( )函数调用。全局变量在一个加globals的表里，通过globals( )函数调用。

2.self参数

在类中定义实例方法时需要将第一个参数显式声明为self，但是在调用的时候并不需要传入该参数。

self 表示的就是**实例对象本身**，即 Selftest 类的对象在内存中的地址。self 是对对象 st 本身的引用。

self本身并**不是python的关键字**，也可以将self换成其他的名称，但是self更符合 约定俗成 的原则。

3.__call__方法

1)__init__方法负责对象的初始化，比如给实例对象的状态进行初始化，系统执行该方法之前，对象就已经存在了。

2)__call__是一个特殊的实例方法，功能类似于在类中重载 ()运算符，使得类实例对象可以像调用普通函数那样，以“对象名()”的形式使用。

3)首先明确可调用对象的概念，比如：自定义的函数，内置函数和类，但凡是可以把一对括号应用到某个对象身上都可称之为可调用对象。如果在类中实现了__call__方法，那么实例对象也将成为一个可调用对象。

4）应用场景：结合类的特性，类可以记录数据（属性），但是函数不行，利用这种特性可以实现基于类的装饰器，在类里面记录状态。

```
# __call__方法
class Test:
    def __call__(self,name,add):
        print('调用__call__方法',name,add)
# 测试
test = Test()
>>>test('hhhh','zzzz')  #调用实例对象
调用__call__方法 hhhh zzzz
```

可以看到，通过在Test类中实现__call__()方法，使得test实例对象变成了可调用对象。

对于可调用对象，“名称()"可以理解为”名称.__call__()"的简写。

```
>>>test.__call__('hhhh','zzzz')
调用__call__方法 hhhh zzzz
```

4.生成器

生成器函数就是每当它执行一次生成语句（yield），就返回一个迭代器。这个迭代器生成一个值，生成值后，生成器函数的运行函数将被挂起，直到下一次生成。

生成方法：

1）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 0x0000024B7E39CF20>
```

如果要一个个打印出来，可以通过 next()函数获得生成器的下一个返回值

```
>>>next(g)
0
>>>next(g)
1
```

但是不断调用next(g)太麻烦，正确的方法是使用 for 循环，因为generator也是可迭代对象。

```
for n in g:
    print(n)
0
1
4
9
...
```

2）根据生成斐波那契数列的函数，可以知道推算规则就是从第一个元素开始，推算出后续任意的元素，这种逻辑类似generator，因此将fib函数变成generator函数，只需要把 print(b) 改成 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'
```

调用一个generator函数将返回一个generator（不是直接返回值）

```
f = fib(6)
>>>f
<generator object fib at 0x0000024B17A8B890>
```

注意：generator函数和普通函数的执行流程不一样，普通函数是顺序执行，遇到return语句或者最后一行函数语句就返回；但是generator函数在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

5.修饰器（装饰器）

通过装饰器函数，在不修改原函数的前提下，来对函数的功能进行合理的扩充。

预备知识：

1）一切皆对象。我们甚至可以将一个函数赋值给一个变量（注意：没有使用小括号，因为并不是调用）

2）在函数中定义函数（记得最后要调用子函数，要不然子函数不会返回值）

3）将函数作为参数传递给另一个函数

使用装饰器A()去装饰另一个函数B()，其底层执行了两步操作：1.将B作为参数传递给A()函数 2.将A()函数执行完成的返回值反馈给B

例子：

```
def aop(func):
    """aop func"""
    def wrapper():
        """wrapper func"""
        print('before func')
        func()
        print('after func')
    return wrapper

@aop    # @aop: hi_with_deco = aop(hi_with_deco)
def hi_with_deco():
    print('hi')

>>>hi_with_deco()
before func
hi
after func
```

思考：如果被修饰的函数本身带有参数，那么该如何传值呢？

上文的闭包函数中，调用的函数是fun()，是无参数的。那么也就意味着， 如果func()是一个带参数的函数，在用这个修饰器就会报错。

```
def hi_with_deco(a):    #传入参数a
    print('hi'+str(a))

>>>hi_with_deco(1)  #报错
TypeError: wrapper() takes 0 positional arguments but 1 was given
```

那么我们需要把修饰器函数改的通用一些，从functools中导入修饰器函数wrapper

```
from functools import wraps

def aop(func):
    @wraps(func)    #在用修饰器函数的时候，还用了别的修饰器函数
    def wrap(*args,**kwargs):
        print('before')
        func(*args,**kwargs)
        print('after')
    return wrap
@aop
def hi(a,b,c):
    print('hi:{}{}{}'.format(a,b,c))
hi(1,2,3)
```

待补充：

第五章：设计模式

第七章：使用工具辅助项目开发

第八章：性能剖析和优化

参考资料：

[1]张颖,赖勇浩.编写高质量代码-改善Python程序的91个建议[M].北京:机械工业出版社,2014

[2]谷歌开源项目Python风格指南：https://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/contents/