# Python笔记v1.4

|  |  |  |
| --- | --- | --- |
| 版本号 | 修改日期 | 修改内容 |
| 1.0 | 2018-01-28 | 增加Python基础知识1-5小节 |
| 1.1 | 2018-02-04 | 增加python基础知识6-11小节 |
| 1.2 | 2081-02-10 | 调整python基础知识小节顺序，新增面向对象、调试、序列化等小节内容 |
| 1.3 | 2018-03-03 | 补充6、7小节 |
| 1.4 | 2018-03-10 | 补充7.4-7.6，13.5小节，以及python对象池 |
|  |  |  |

## 一、Python基础知识

### 1.变量

**理解**：存储了数据的内存空间，变量类型由数据类型决定；

**举例**：num = 10; str = ’string’; lis = [1,2,3];

局部变量：定义在函数内部或程序特定部分的变量；

全局变量：定义在程序顶部，全程序可调用的变量，global x强制声明为全局变量；

**备注**：其中变量名只是对数据所在空间的索引，由字母、下划线、数字组成；

**深入**：[垃圾回收机制](#_1.垃圾回收机制)；[内存泄漏](#_2.内存泄漏)；

### 2.运算符

理解：对数据进行操作的标识；

举例：x + y; x & y; x << 2;

优先级：or < and < not < in/not in < is/is not < 比较 < | < < & < <</>> < +/- < \*, /, % < 正负号 < 位反 < \*\*

**备注**：is/is not和 in/not in用于判断；

### 3.数据类型

1. 数字

* int

理解：范围限制在 -2,147,483,648 到 2,147,483,647 内的整数

举例：10; -99; 3480;

* long

理解：范围无限制的整数

举例：10L; -99L; 3480L;

* float

理解：带小数点的数

举例：1.25; -3.9; -978.7;

* complex

理解：复数（虚数）

举例：4+0.5j; -3j; 4j;

1. 序列

* string

理解：用 ’ ’/ “ ”/ “”” “”” 包含在内的一连串字符；

举例：’hello’; “world”; “””I am fine”””;

* list

理解：可包含多种数据类型内容有序可变的容器；

举例：[1,2,3]; [‘a’, ’b’, ’c’]; [‘a’,12,[1,2,3]];

列表表达式：[ generate\_element\_expr for x in list if expr ]

生成器表达式：( generate\_element\_expr for x in list if expr )

备注：“+”运算符可用于拼接两个list

深入：[迭代对象、迭代器、生成器、生成器函数](#_3._迭代对象、迭代器、生成器、生成器函数)；

* tuple

理解：可包含多种数据类型内容有序不可变的容器；

举例：(1,); (‘a’ , ’b’); (1,2,’a’,[‘x’,’y’]);

* 操作

1. 切片

理解：选取序列其中一部分，[startIndex : destIndex+1 : step]；

举例：str = ‘abcde’

str[:] 🡪 ‘abcde’

str[::2] 🡪 ‘ace’ # ‘ace’可理解为python内部新建了一个’ace’并返回

str[-5:-2:-1] 🡪 ‘dcb’

**备注**：索引从左到右：0,1,2,3…；从右到左：-1,-2,-3,-4…；

step为正时从左到右取序列元素，step为负时从右到左取序列元素；

1. 内置函数

理解：常用的序列处理函数

举例：+ \* in len() max() min() cmp() filter() zip() map() reduce()

**备注**：is 是否是同一个对象

== 值是否相等

cmp(x,y) x>y 🡪 1; x<y 🡪 -1; x==y 🡪 0;

1. 字典

理解：可包含多种数据类型的“键：值”对的内容无序可变（键值不可变）的容器；

举例：{ ‘name’:’John’, ‘age’:30, ‘tel’:138 }

深入：[list、tuple、dict实现](#_3.list、tuple、dict实现)；

### 4.结构

1. if else

理解：判断分支结构；

举例：if expr:

Statements

elif:

Statements

else:

Statements

**备注**：用缩进代替语句分组；

1. for

理解：次数既定的循环结构；

举例：for x in range(10):

print x

else:

print ‘okay’

**备注**：正常执行完循环后执行else，可将for和else看作一个整体，break将跳出这个整体；

1. while

理解：次数不定的循环结构

举例：while expr:

Statements

else:

Statements

**备注**：expr为假时执行else

1. switch

理解：用字典实现的类if else 多分支判断结构；

举例：dict={ “+”:add, “-”:sub, “\*”:mul, “/”:div } #add/sub/mul/div为函数名

dict.get(operator)(x,y) #dict[“+”]=add

### 5.函数

理解：实现特定功能的程序块；

举例：def fun ( par1, par2 ) # par1=??? 定义默认形参，默认形参置于最右边，

Print “ %s --- %s ” % ( par1, par2 )

#回车结束函数，默认return None

**备注**：fun( \*tuple ) 声明传递tuple（list直接传递），形参按tuple元素对应一一取值；

fun( \*\*dict ) 声明传递dict，形参和字典中的key一一对应，取key对应的value；

fun( x, \*tuple, \*\*dict ) 可吸收多余参数；

* lambda函数

理解：对结构简洁的函数用一句代码总结；

举例：lambda par1, par2 : par1+par2 #冒号‘：’后为返回值

* 内置函数

理解：python已经定义好的函数，可直接调用；

举例：help() callable() isinstance() range()

### 6.模块和包

模块：编译好的包含了特定函数的\*.py/pyc/pyo文件

举例：import re; import copy as c; from os import walk;

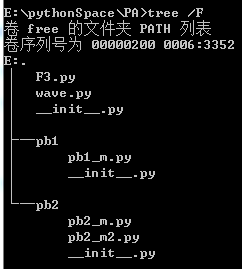
**备注**：执行import导入模块语句时，会先查询该模块是否已被加载过（在sys.modules中），如果未被加载则获取模块查找路径sys.path，按顺序查找模块；找到模块后，会将模块编译成中间代码，并将模块中的代码都执行一遍。也因此对于模块的属性\_\_name\_\_，使用不同的调用方式 python xxx.py（直接调用） 和 import xxx.py（间接调用）会有不同的输出结果：

python xxx.py 🡪 输出 \_\_main\_\_

import xxx.py 🡪 输出模块名字xxx

* 包

只要一个文件夹下面有个 \_\_init\_\_.py 文件，那么这个文件夹就可以看做是一个包。包导入的过程和模块的基本一致，只是导入包的时候会执行此包目录下的 \_\_init\_\_.py 而不是模块里面的语句了。另外，如果只是单纯的导入包，而包的 \_\_init\_\_.py 中又没有明确的其他初始化操作，那么此包下面的模块是不会自动导入的。



**举例：**

包PA，有目录树结构如右图所示，可知包PA下包含了两个模块F3和wave，以及两个包pb1和pb2。不断执行import语句导入模块，其测试代码和执行结果如下图所示，由测试结果可得出结论：

1. import xxx || XXX.xxx

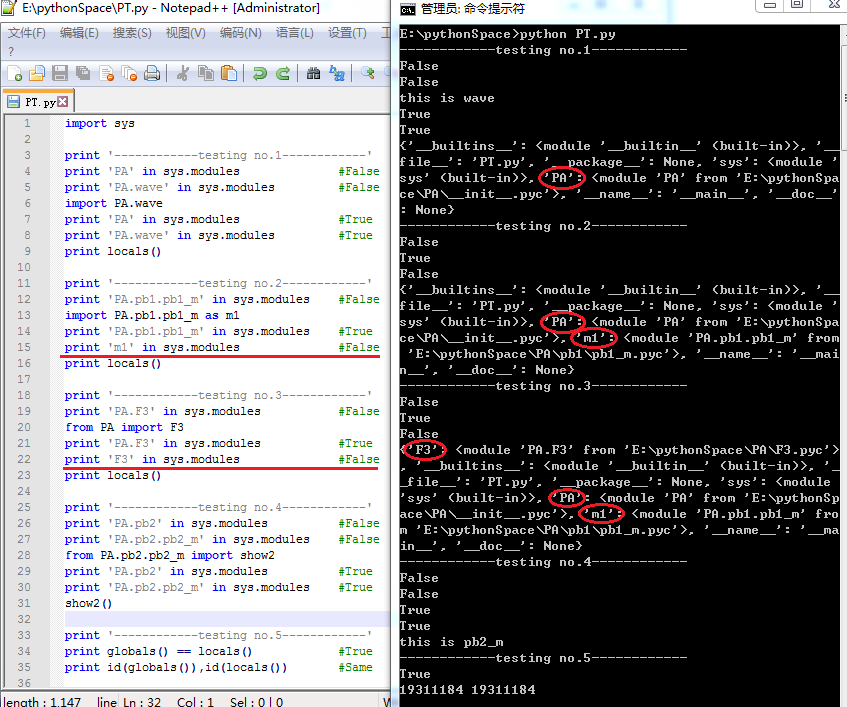
首次执行会将xxx或者XXX、XXX.xxx添加到sys.modules（使用完全引用名（如PA.wave、PA.pb1.pb1\_m）保存所有加载到内存中的模块），并将xxx或XXX等可**直接使用**名称.属性**调用的模块**（如PA）加入到当前脚本的命名空间global（或local，保存该脚本中可直接使用名称(.属性)调用的模块、类、方法等）中。

1. import axx || XXX.axx as bxx

为所导入的模块添加别名，可直接使用**bxx.属性**对模块进行访问，也因此会将bxx添加到脚本的命名空间中，但仍然是使用完全引用名添加到sys.modules中。

1. from axx || XXX.axx import bxx

从axx或XXX.axx包中导入模块或方法，若导入模块则将axx、axx.bxx或XXX.axx、XXX.axx.bxx添加到sys.modules中；否则只将axx或XXX.axx添加到sys.modules中。而bxx将会被添加到命名空间中，并且不能使用axx.bxx或者XXX.axx.bxx进行访问。



**参考：**

[1] python import 导入模块执行流程分析

<http://blog.csdn.net/tuxl_c_s_d_n/article/details/45462139>

[2] Python之import机制详解<http://www.jb51.net/article/51815.htm>

[3] python中获取执行脚本路径方法<https://www.cnblogs.com/chengd/p/7100782.html>

[4] python命名空间 <http://blog.csdn.net/u012436149/article/details/72819539>

### 7.面向对象

面向对象是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，因此有了类的定义。类的出现使面向对象的语言具有了以下特点：封装、继承、多态。

#### 封装

类是定义了一系列属性和方法的集合，有些属性和方法可以被外部访问，有些不可以，只能在类内部被访问，这就是封装，Python中使用 \_(protected) 和 \_\_(private) 对受保护的属性和方法进行标识。

定义一个类：

class ClassName(ParentClass):

Definition

类有一些特殊的属性和方法支持自行定制类，如下所示：

**特殊的类属性**

|  |  |
| --- | --- |
| 类属性 | 含义 |
| \_\_name\_\_ | 类的名字（字符串） |
| \_\_doc\_\_ | 类的文档字符串 |
| \_\_bases\_\_ | 类的所有父类组成的元组 |
| \_\_dict\_\_ | 类的属性组成的字典 |
| \_\_module\_\_ | 类所属的模块 |
| \_\_class\_\_ | 用于查看类对象是由哪个对象生成的 |
| \_\_slots\_\_ | 限制当前类所能拥有的属性，对子类不起作用 |
| \_\_mro\_\_ | 类的方法解析顺序，等同于mro() |
| \_\_metaclass\_\_ | 控制类的创建 |

**特殊的类方法**

|  |  |
| --- | --- |
| 类方法 | 含义 |
| \_\_new\_\_() | 定义类对象的创建，在\_\_init\_\_()之前执行 |
| \_\_init\_\_() | 定义类对象的初始化 |
| \_\_str\_\_() | 定义>>> print class时的输出信息 |
| \_\_repr\_\_() | 定义>>> class时的输出信息 |
| \_\_iter\_\_() | 返回一个迭代对象，可用于for in循环 |
| \_\_getitem\_\_() | 使类可通过下标获取元素，相应还有\_\_setitem\_\_()和\_\_delitem\_\_()等方法 |
| \_\_getattr\_\_() | 定义访问类不存在的属性时的输出信息，相应还有\_\_setattr\_\_()和\_\_delattr\_\_()等方法 |
| \_\_call\_\_() | 定义直接调用实例本身instance()时的输出信息 |
| \_\_cmp\_\_() | 定义类对象如何进行比较，以便使用sorted()进行排序 |
| \_\_len\_\_() | 定义类对象的返回长度 |
| \_\_del\_\_() | 定义在类对象销毁时的析构函数 |

* @property

Property可使getter()方法变成属性访问，同时需定义相应的setter()方法，否则该属性只可读不可写。

举例：@property

def Xxx(self): #getter()

pass

@Xxx.setter

def Xxx(self,value): #setter()

pass

#### 继承

Python支持多继承，子类会继承父类除\_\_doc\_\_外的所有属性和方法。当子类没有构造方法时会继承父类的构造方法，但多继承的子类应该继承哪一个父类的构造方法？为了解决这一问题，python引入了**方法解析顺序**（Method Resolution Order, MRO）[1]，可解析出子类、子类的父类以及父类的父类，直到基类Obejct这样一个顺序链。无构造方法的子类则会继承，按方法解析顺序搜索到的第一个父类的构造方法，而Python从2.3开始使用C3算法[2]计算出类的方法解析顺序。

* **C3算法 🡪 方法解析顺序**

假设有A、B、C、D、E、F等类，有类的继承关系图如下所示，则C3算法计算出类的方法解析顺序公式如下所示：

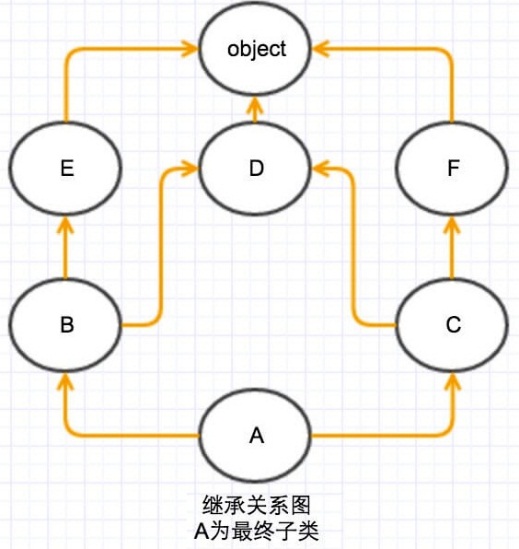
**L( Child( B1B2…BN ) ) = Child + merge( L(B1), L(B2), …, L(BN), B1B2…BN )**

**L(object) = object**

合并merge规则有：

**取出merge中第一个列表的头，也就是L(B1)[0]（头之外的剩下部分称为尾）；如果该头不在其他列表的尾中，那么将该头添加到Child的线性化中，并将其从merge操作的所有列表中删除，否则查看下一个列表的头并操作它。然后，重复该操作直到所有的类都被删掉或者无法再找到符合要求的头。**

L( O ) = O

L( E ) = E O #单继承类可以直接写出

L( D ) = D O

L( F ) = F O

L( B ) = B + merge( L(E), L(D), ED )

= B + merge( EO, DO, ED )

= B + E + merge( O, DO, D )

= B + E + D +merge( O, O )

= B E D O

同理，可得 L( C ) = C D F O

因此，可得 L( A ) = A + merge( L(B), L(C), BC )

= A + merge( BEDO, CDFO, BC )

= A + B + merge( EDO, CDFO, C )

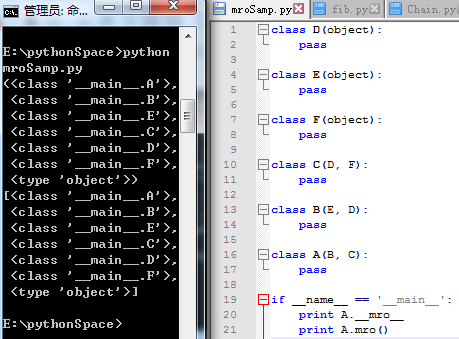
= A + B + E + merge( DO, CDFO, C )

= A + B + E + C + merge( DO, DFO )

= A + B + E + C + D + merge( O, FO )

= A B E C D F O

通过访问类A的\_\_mro\_\_属性和mro()方法可得类A的方法解析顺序和上述分析一致。



但是也有可能报错 “Cannot create a consistent method resolution order (MRO) for bases”， 无法解析出子类的方法解析顺序，因此在python中编写类时需要注意以下几点：

1. **在继承父类时，更具体的父类应该更先继承**
2. **避免定义父类继承顺序不一样的新子类，如：**

class A( X, Y ): pass #避免定义 class B( Y, X ): pass 这样的新子类

**参考：**

[1] python2.3方法解析顺序（译）<http://www.nanerbang.com/article/40/>

[2] python多继承C3算法 <http://blog.csdn.net/fmblzf/article/details/52512145>

* **Super()调用**

在子类中调用父类方法有两种，一种是直接使用父类名调用方法，一种是使用super()进行方法调用。其中，对于多继承的子类，使用super()调用父类方法时，如何定位父类？

Super()的工作原理如下[1]：

**def super( cls, inst ):**

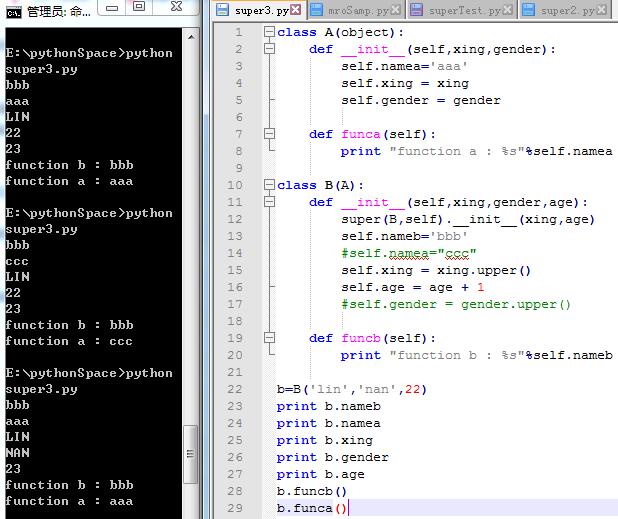
**mro = inst.\_\_class\_\_.mro()**

**return mro[ mro.index( cls ) + 1 ]**

其中，cls 代表类，inst 代表实例，上面的代码做了两件事：获取 inst 的 MRO 列表；查找 cls 在当前 MRO 列表中的 index, 并返回它的下一个类，即 mro[index + 1]。因此使用 super(cls, inst) ，Python 会在 inst 的 MRO 列表上搜索位于 cls 后的下一个类。

子类使用 super( Child, self ) 时，是将子类对象转换为父类对象，因而可以调用父类方法。而self参数已在super()中传入，因此不需要在调用方法时再次传入，在后续调用中也将隐式传递（维持不变）。

**举例**：



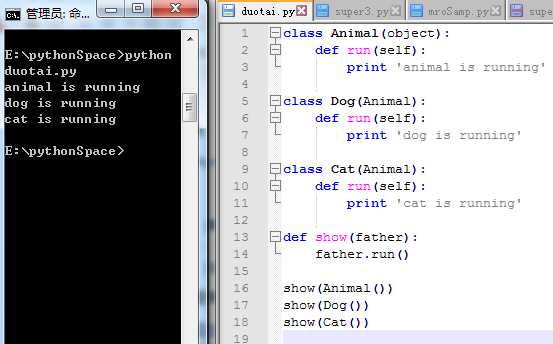
其中，第二次和第三次执行结果分别为取消注释 self.namea = ’ccc’ 和 self.gender = gender.upper() 的执行结果，由此可见，使用super()调用父类方法后，若子类并未覆盖父类属性，则子类将原样继承父类属性。

**参考**：

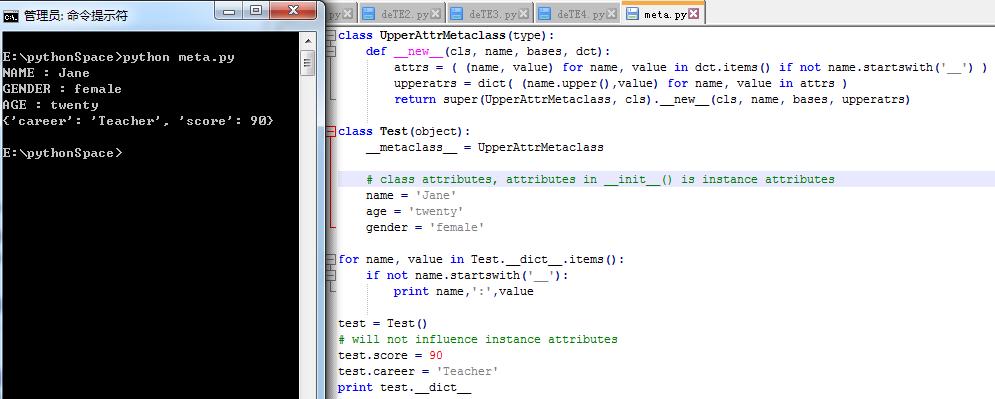
[1] Python你不知道的super <http://python.jobbole.com/86787/>

#### 多态

由于子类会继承父类的全部功能，子类的方法又可以覆盖父类的方法，因此若是一个函数需要根据实际传入的子类对象显示不同操作，或不确定子类对象如何定义时，可定义一个父类形参，则函数会根据传入的实参调用相应的类的属性或方法。



#### 元类

由于python是动态脚本语言，在程序运行期间才能确定数据类型，实际上创建类就是使用type( name, bases, dict )函数创建出一个类对象，因此type(class)会返回< type ‘type’ >。python将这种可以创建类的类称为元类[1]，type就是python中最顶层的元类。Python中比较特殊的可以控制类的创建的对象metaclass，就是继承type而来。通过在类定义中设置\_\_metaclass\_\_属性就可使该类按元类的要求来创建，而在模块中定义\_\_metaclass\_\_属性则会使该模块中的所有类都按照元类的要求来创建。元类的主要作用就是：拦截类的创建 🡪 修改类 🡪 返回修改之后的类，具体示例如下图所示。

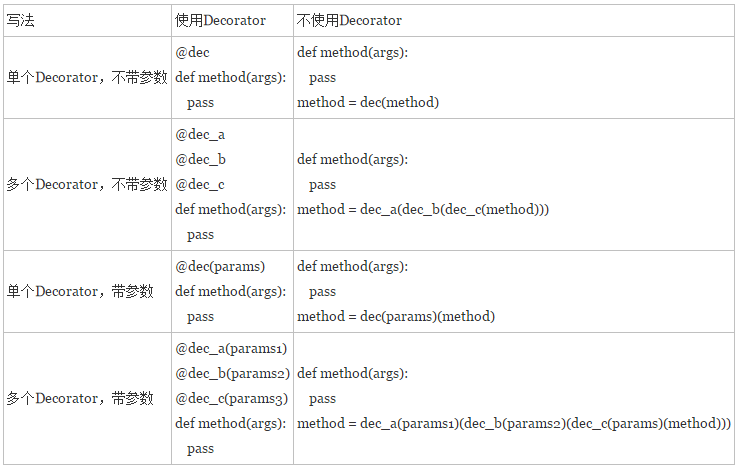
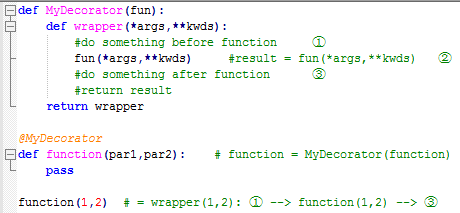
参考：

[1] 深刻理解python中的元类（metaclass） <http://blog.jobbole.com/21351/>

#### 修饰器

修饰器主要分为两种：函数修饰器和类修饰器，分别用于管理函数和类。

* **函数修饰器**

修饰器的实质是函数，定义了对被修饰函数的一些操作，如下图所示。

* 对于带参数的修饰器，只需要在修饰器函数外面再套一层带参数的函数声明即可；
* 对于使用多个修饰器修饰的函数，如：

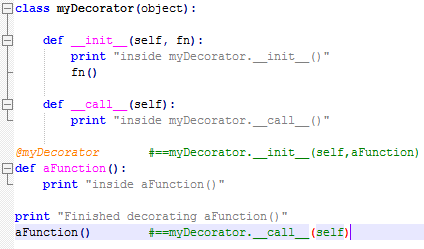
@DecoratorA

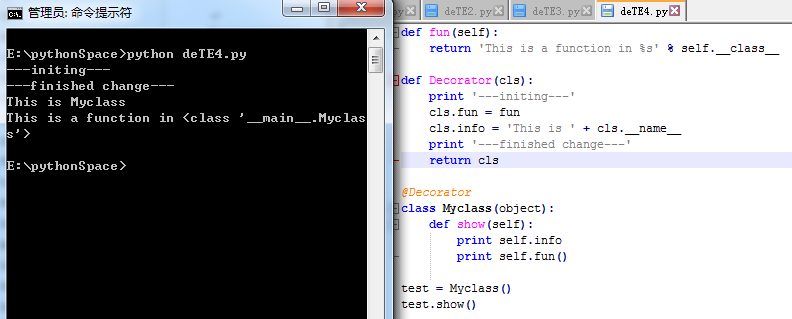
@DecoratorB

def fun() : pass

fun()

有执行顺序：wrapperA①, wrapperB①, fun(), wrapperB③, wrapperA③

* 对于class式的修饰器，有\_\_init\_\_()和\_\_call\_\_()方法，执行顺序如下：
* **类修饰器**

类修饰器的原理和函数修饰器一致，只是类装饰器对类进行管理，也因此类修饰器需要将类作为输入参数，在类修饰器中可添加或修改类的函数及属性，并返回修改后的类，如下图所示。

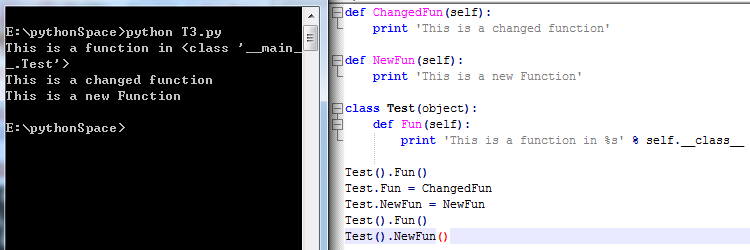
参考：

[1] python之美——Decorator深入详解之美（一）<https://www.cnblogs.com/SeasonLee/articles/1719444.html>

[2] python修饰器的作用 <http://blog.csdn.net/xx5595480/article/details/72510854>

[3] python深入学习——decorator强大的修饰器 <http://blog.csdn.net/majianfei1023/article/details/45001021>

#### 猴子补丁

猴子补丁同修饰器和元类一样，可以在运行时动态替换或增加模块的方法，而不需要修改原始代码。一个简单的例子如下所示。

参考：

[1] python基础：Monkey patch（猴子补丁）<http://blog.csdn.net/fly910905/article/details/77152110>

### 8.正则表达式

理解：对字符串做过滤的表达式

举例：

|  |  |  |
| --- | --- | --- |
| 符号 | 含义 | 备注 |
| [ ] | 匹配字符集中的一个 | 元字符不起作用，做普通字符 |
| ^ | 匹配开头 | 在[ ]中表示取反 |
| $ | 匹配结尾 |  |
| . | 匹配除换行符（\n）之外的任意单字符 |  |
| \d | 匹配数字集 | 等于[0-9]，取反\D |
| \s | 匹配空字符集 | 等于[\f\n\r\t\v]，取反\S |
| \w | 匹配字母和数字集 | 等于[a-zA-Z0-9\_]，取反\W |
| \* | （最大匹配）匹配>=0次 | 最小匹配\*? |
| + | （最大匹配）匹配>=1次 | 最小匹配+? |
| ? | 匹配0或1次 |  |
| {n} | 重复n次 |  |
| {n,m} | 重复至少n次，至多m次 |  |
| ( ) | 分组，匹配时优先返回符合字符串 |  |

import re

p\_tel=re.compile(r’\d{3,4}-?\d{8}’)

p\_tel.findall(‘010-12345678’) #相等于 re.findall( r’\d{3,4}-?\d{8}’ , ‘010-12345678’ )

### 9.深浅拷贝

1. 浅拷贝

理解：对列表等父对象的内存空间的拷贝

举例：import copy

a=[1,2,3,[‘a’, ‘b’, ‘c’]] # b=a a,b引用同一块内存空间，一起改变

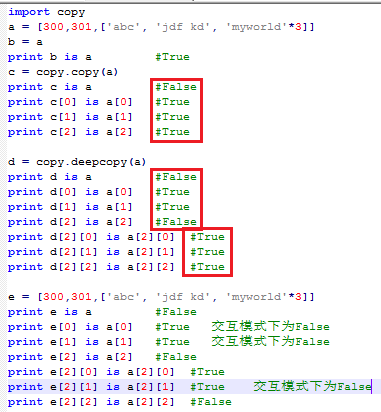
c=copy.copy(a) #具体内存空间引用见

1. 深拷贝

理解：对列表等资源的内存空间的拷贝

举例：d=copy.deepcopy(a) #具体内存空间引用见图

深入：[python对象池，小整数，大整数，intern池？](#_5._python对象池，小整数，大整数，intern池？)



### 10.文件读写

理解：对文件进行读取或写入

举例：fo=open( ‘test.txt’, ’w’ ) # fo=file( ‘test.txt’, ‘rb’ )

fo.write( ‘hello’ )

fo.flush()

fo.close()

**备注**：read() readline() readlines() next() writelines(list) seek()

使用with as 语句可不用手动关闭文件手柄，如：

with open( ‘test.txt’, ‘a’ ) as fo:

fo.write( ‘This is a test.’ )

### 11. os模块

理解：对系统文件夹和文件进行操作

举例：import os

os.makedirs(‘a/b/c’)

os.walk(‘a/b/c’)

os.removedirs(‘a/b/c’)

**备注**：mkdir() rmdir() chdir() getcwd() listdir() path.isdir() path.isfile() path.join()

Python还提供了shutil模块作为对os模块的补充，可实现移动、复制、打包、压缩、解压等功能

### 12.异常

理解：对代码报错进行处理

举例：try:

fo=open(‘abc.txt’)

raise NameError(‘this is a nameError!’)

except IOError, msg:

Statements

except NameError, msg:

Statements

finally:

fo.close()

### 13.调试

#### Assert断言

理解：若数据类型不符合预期则抛出异常

举例：div\_num=int( raw\_intput( ‘Please input an integer:’ )

assert div\_num != 0, ‘div\_num equals zero! ’

‘’’ assert语句相等于

if not expr:

raise AssertionError(msg)

‘’’

#### Pdb模块

理解：调试模块，可执行设置断点、单步执行以及查看变量等操作

举例：

Pdb模块常用命令



备注：使用pdb模块调试python脚本<http://blog.csdn.net/jerry_1126/article/details/43915609>

#### Logging日志

理解：可按照相应格式记录不同等级日志信息

举例：import logging

logging.basicConfig( filename=‘log.txt’, filemode=‘w’, level=logging.INFO )

logging.info( ‘Information from … ’ )

logging.warning( ‘Warning from …’ )

**备注**：

Python中的logging模块<http://python.jobbole.com/86887/>

python logging模块<https://www.cnblogs.com/liujiacai/p/7804848.html>

#### 文档测试

理解：对’’’中所记录的交互模式下预期输入与输出’’’进行测试判断

举例：import doctest

def multi( n ):

‘’’ #doctest

>>> multi(4) # >>>和multi(4)之间有个空格

24

>>> multi()

3628800

‘’’

res = 1

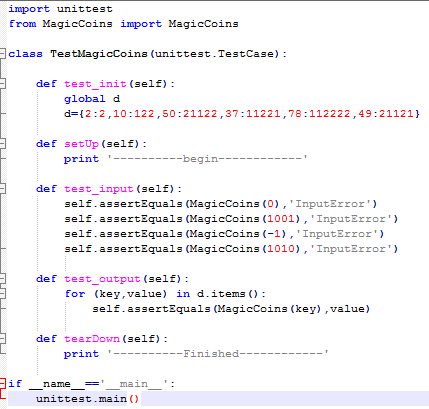
for i in range(1,n+1):

res\*=i

return res

**doctest.testmod( verbose = True )**

#### 单元测试

单元测试是对软件中最小可测试单元进行测试和验证。在python中专门提供了单元测试框架unittest，可对函数、类和模块等进行测试。例如，对一个函数的预期输入输出进行测试，可以通过文档测试来做，也可以使用单元测试来做，具体示例如下图所示。

* 单元测试类需要继承unittest.TestCase
* test开头的方法是测试方法，非测试方法测试时不会执行
* setup()和tearDown()方法分别在每个测试方法开始之前和结束之后执行
* unittest.TestCase常用的方法有：

assertEqual( a, b) 判断a,b是否相等

assertTrue(expr) 判断expr是否为真

assertRaises(Excep) 判断是否抛出异常

( with self.assertRaises( TypeError ):

raise TypeError )

以上断言判断为假时抛出AssertionError

参考：

[1] python unittest之断言及示例 <http://blog.csdn.net/julia294/article/details/70098942>

### 14.序列化

理解：数据从内存中变为可存储或可传输的过程

举例：import pickle

d = { ‘name’:’Bob’, ‘age’:20, ‘score’:88 }

d\_str = pickle.dumps(d) #将字典序列化为字符串

with open(‘saveDict.txt’, ‘wb’) as fo:

pickle.dump( d, fo) #将字典序列化成字符串并保存于文件中

str\_d = pickle.loads(d\_str) #反序列化成字典，也可使用pickle.load(fo)

**备注**：python还可将字典、类对象等序列化为json格式，同样有dumps() dump() loads() load()等方法，但json格式反序列化所得对象中的字符串皆为unicode格式，而python2.7默认字符串是string格式，要完全反序列化为python2.7对象可能还需要进行转码操作，或者使用yaml模块来进行反序列化。

[**http://blog.csdn.net/duoduo\_smile/article/details/52783292**](http://blog.csdn.net/duoduo_smile/article/details/52783292)

* 对类对象进行序列化，参考<https://www.liaoxuefeng.com/wiki/001374738125095c955c1e6d8bb493182103fac9270762a000/00138683221577998e407bb309542d9b6a68d9276bc3dbe000>

## 二、Python实践运用

## 三、Python深入了解

### 1.垃圾回收机制

python采用的是引用计数机制为主，标记-清除和分代收集两种机制为辅的垃圾回收策略（引用计数不能处理含有循环引用的数据结构）。

* 引用计数

创建一个对象时，Python总是在对象的C结构体里保存一个整数，称为引用数。期初，Python将这个值设置为1。每当对象的引用数减为0，Python立即将其释放，把内存还给操作系统。

* 标记-清除

标记-清除机制，顾名思义，首先标记对象（垃圾检测），然后清除垃圾（垃圾回收）。首先初始所有对象标记为白色，并确定根节点对象，标记它们为黑色。将根节点引用的对象标记为灰色，检查完灰色对象引用的对象后，将灰色标记为黑色。重复直到不存在灰色节点为止，最后白色结点都是需要清除的对象。

* 分代收集

Python使用一种不同的链表来持续追踪活跃的对象，Python的内部C代码将其称为零代(Generation Zero)。每次当你创建一个对象或其他什么值的时候，Python会将其加入零代链表，零代链表是一个完全内部的Python运行时（？？？）。随后，Python会循环遍历零代链表上的每个对象，检查列表中每个互相引用的对象，根据规则减掉其引用计数。当对象的引用计数变为零时，收集器可以释放它们并回收内存空间了，剩下的活跃的对象则被移动到一个新的链表：一代链表。通过不同的GC阈值设置，Python可以在不同的时间间隔处理这些对象。Python处理零代最为频繁，其次是一代然后才是二代。

**参考**：

[1] Python垃圾回收机制——完美讲解！<https://www.cnblogs.com/pinganzi/p/6646742.html>

[2] python垃圾回收机 <https://www.cnblogs.com/mingaixin/archive/2013/01/31/2886680.html>

### 2.内存泄露

如果没有禁用垃圾回收，那么Python中的内存泄露有两种情况：要么是对象被生命周期更长的对象所引用，比如global作用域对象；要么是循环引用中的对象定义了\_\_del\_\_函数。当一个对象理论上（或者逻辑上）不再被使用了，但事实上没有被释放，那么就存在内存泄露；当一个对象事实上已经不可达，即不能通过任何变量找到这个对象，但这个对象没有立即被释放，那么则可能存在循环引用。

参考：

[1] 使用 GC、Objgraph 干掉 Python 内存泄露与循环引用！<http://python.jobbole.com/88827/>

### 3.list、tuple、dict实现

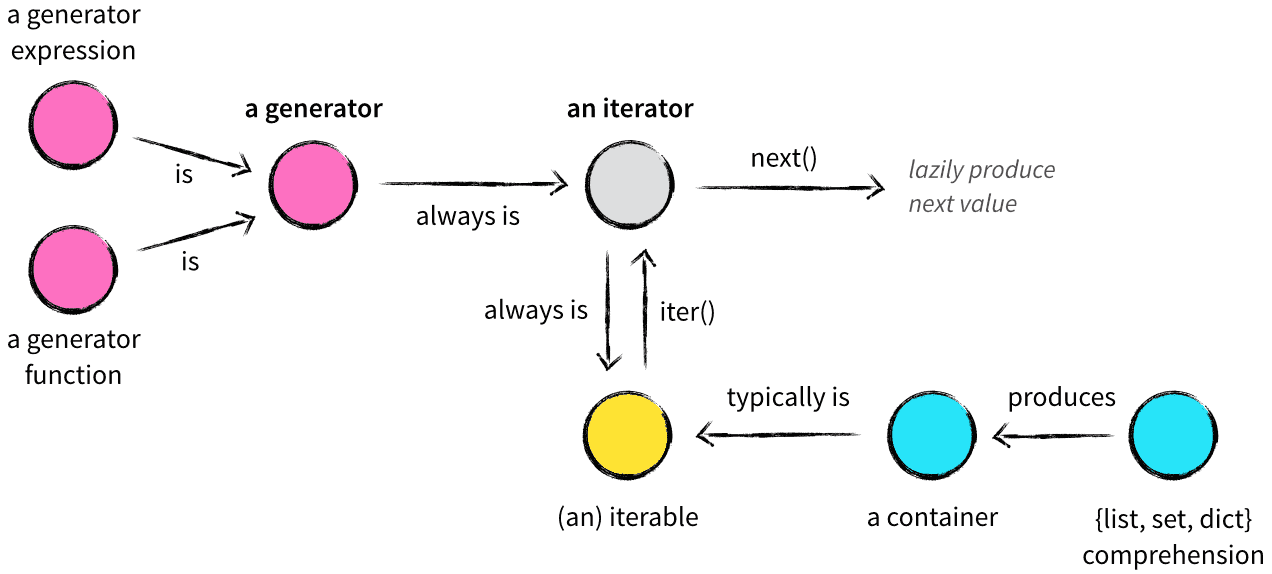
List、tuple的实质类似于指针数组，每个元素分别指向实际所存储的数据内存空间，因此可包含多种数据类型。Dict是通过哈希表实现的，也可视作彼此关联的两个数组，通过哈希函数处理key值后可寻址到value。

参考：

[1] Python中list的实现<https://www.jianshu.com/p/J4U6rR>

[2] 深入Python字典的内部实现<http://python.jobbole.com/85040/>

### 4.迭代对象、迭代器、生成器、生成器函数



for x in iterable

迭代器是一个带状态的对象，能在调用next()方法的时候返回迭代器中的下一个值。但凡是可以返回一个迭代器（iter()）的对象都可称之为可迭代对象，大部分容器都是可迭代对象。

* 生成器generator

使用生成器表达式可创建生成器，生成器表达式和列表表达式相似，只是生成器的访问需要不断地调用next()方法，访问完全部元素后再调用next()方法则会抛出StopIteration异常。但是这样操作太过繁琐，一般使用for循环进行迭代访问。

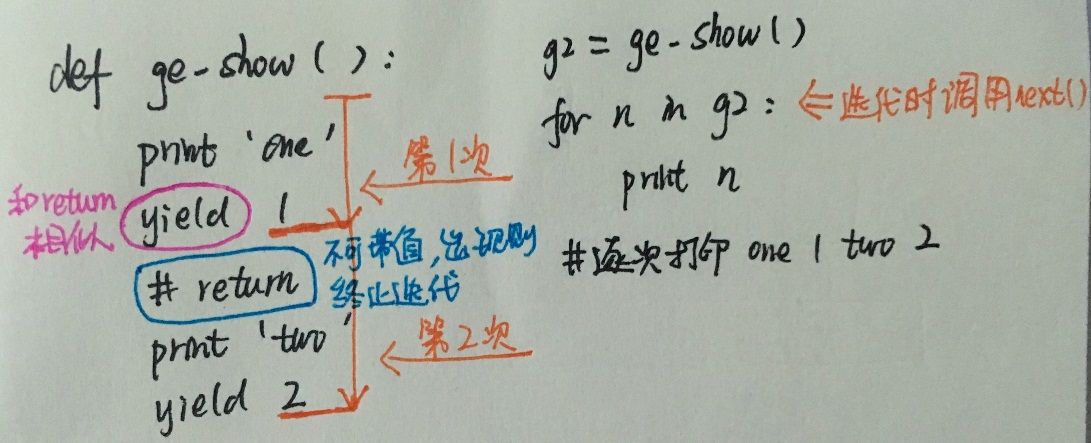
还有一种创建生成器的方式是，在函数中添加yield字段，就将函数变成了生成器。而这种生成器函数的执行流程也不同于普通函数，生成器函数在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

理解：保存元素的递推算法，需要访问时再逐次计算的机制。

举例：g = ( x\*x for x in range(1,10) if x%2==0 ) #生成器表达式创建的生成器

for n in g:

print n #逐次打印 4 16 36 64



参考：

[1] 完全理解Python迭代对象、迭代器、生成器<http://python.jobbole.com/87805/>

[2] Python生成器<https://www.liaoxuefeng.com/wiki/001374738125095c955c1e6d8bb493182103fac9270762a000/00138681965108490cb4c13182e472f8d87830f13be6e88000>

### 5. python对象池，小整数，大整数，intern池？

小整数：范围在[ -5, 256 ]范围内的整数，不会被重复创建，位于这个范围内的指向相同整数值的变量都是指向同一个对象。

大整数：超出小整数范围内的整数称为大整数，每运行一次，大整数都重新创建一次。对于位于同一块代码中的变量，若指向相同的大整数值，则该变量实际上指向的是同一个对象。

Intern机制：

* 对于只包含数字、字母、下划线的字符串，只会创建一次，其余字符串则会创建多次（交互模式下及不同的代码块中，在同一块代码中无此限制）。
* 使用运算符如‘+’或‘\*’对字符串进行运算时，若运算后的字符串长度不超过20，仍只创建一次，否则创建多次，调用了函数或‘[]’切片对字符串进行处理也会创建新的字符串对象（任何情况下都成立）。

参考：

[1] python中小整数对象池和大整数对象池<http://blog.csdn.net/wangyunfeis/article/details/77607156>

[2] 什么是string interning以及python中字符串的intern机制

<https://www.cnblogs.com/brucemengbm/p/6952822.html>