1. **说一说Python中的函数的参数传递方式都有哪些？**

Python中函数的参数传递方式分为：

位置传参

序列传参

关键字传参

字典关键字传参

位置形参

星号元组形参

命名关键字形参

双星号字典形参

1.位置传参:

实际调用参数(实参) 的对应关系与形式参数(形参)的对应关系是按位置来依次对 应的

2.序列传参:

序列传参是批在函数调用过程中,用 \* 将序列拆解后按位置进行传递的传参方式

实参和形参通过序列传递和匹配

3.关键字传参

是指传参时,按着形参的名称给形参赋值。实参和形参按名称进行匹配。注意：实 参和形参按形参名进行匹配,可以不按位置进行匹配

4.字典关键字传参:

实参为字典,用\*\*拆解字典后再进行关键字传参

注意：

字典的键名和形参名必须一致

字典的键名必须为字符串

字典的键名要在形参中存在

5.综合传参:

函数的传参方式在能确定形参能唯一匹配到相应实参的情况下可以任意组合

注意：

通常位置传参和序列传参先传递,其次是关键字传参和字典关键字传参

6.函数的缺省参数

语法:

def 函数名(形参名1=默认实参1, 形参名2=默认实参2, ...):

语句

说明:

缺省参数必须自右至左依次存在

缺省参数可以有0个,1个,多个,甚至全部都有缺省参数

缺省参数的绑定对象存在于函数内,同函数的生命周期一致

7.位置形参:

def 函数名(形参名1, 形参名2, ....):

语句块

8.星号元组形参:

语法:

def 函数名(\*元组形参名):

语句块

作用:

收集多余的位置传参

9.命名关键字形参

语法:

def 函数名(\*, 命名关键字形参):

语句块

或

def 函数名(\*args, 命名关键字形参):

语句块

作用:

所有的命名关键字形参都强制调用者采用关键字传参或字典关键字传参的方 式传递

10.双星号字典形参:

语法:

def 函数名(\*\*字典形参名):

语句块

作用:

收集多余的关键字传参

注意:

字典形参名通常叫 kwargs

函数参数说明:

位置形参,缺省参数,星号元组形参,命名关键字形参,双星号字典形参可以混合使用

函数参数自左至右的顺序为:

1. 位置形参

2. 星号元组形参

3. 命名关键字形参

4. 双星号字典形参

1. **你能详细解释一下什么是Python自省么？**

如果说能够通过一个函数就能学会Python，那这个函数是什么？没错，help()。help()函数的作用就是查看对象的帮助文档。比如：

anwc@anwc:/media/anwc/帅无敌的小安安/课程资料$ python3

Python 3.6.5 (default, Apr 1 2018, 05:46:30)

[GCC 7.3.0] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>> help(123)

Help on int object:

class int(object)

| int(x=0) -> integer

| int(x, base=10) -> integer

|

| Convert a number or string to an integer, or return 0 if no arguments

| are given. If x is a number, return x.\_\_int\_\_(). For floating point

| numbers, this truncates towards zero.

|

| If x is not a number or if base is given, then x must be a string,

| bytes, or bytearray instance representing an integer literal in the

| given base. The literal can be preceded by '+' or '-' and be surrounded

| by whitespace. The base defaults to 10. Valid bases are 0 and 2-36.

| Base 0 means to interpret the base from the string as an integer literal.

| >> > int('0b100', base=0)

...

内容很多，就不一一复制了，help()函数其实就是查看对象的帮助文档，类似于把帮助文档放到linux命令中的less中的效果，按下q就可以退出。

上面的例子，显示了 int 对象的帮助文档，int 是内置类型，代码在编译 Python 的时候已经 编译过了，没有.py 的源代码。

那么我们来看看，help 到底有什么功能，看下面：

>>> help(help)

Help on \_Helper in module \_sitebuiltins object:

class \_Helper(builtins.object)

| Define the builtin 'help'.

|

| This is a wrapper around pydoc.help that provides a helpful message

| when 'help' is typed at the Python interactive prompt.

|

| Calling help() at the Python prompt starts an interactive help session.

| Calling help(thing) prints help for the python object 'thing'.

|

| Methods defined here:

|

| \_\_call\_\_(self, \*args, \*\*kwds)

| Call self as a function.

|

:

如上可以看到，help()函数加与不加参数都可以使用：

>> > help()

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out

the tutorial on the Internet at https: // docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing

Python programs and using Python modules. To quit this help utility and

return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type

"modules", "keywords", "symbols", or "topics". Each module also comes

with a one-line summary of what it does

to list the modules whose name

or summary contain a given string such as "spam", type "modules spam".

help >

问题继续抛出：我们如何知道Python中还有哪些元素等待探索，我们怎么找到这些元素呢？

ok，来看看Python自带说明的第二个函数dir()

我们先用help()查看下这个函数的帮助文档：

>>> help(dir)

Help on built-in function dir in module builtins:

dir(...)

dir([object]) -> list of strings

If called without an argument, return the names in the current scope.

Else, return an alphabetized list of names comprising(some of) the attributes of the given object, and of attributes reachable from it.

If the object supplies a method named \_\_dir\_\_, it will be used

otherwise

the default dir() logic is used and returns:

for a module object:

the module's attributes.

for a class object:

its attributes, and recursively the attributes

of its bases.

for any other object:

its attributes, its class's attributes, and

recursively the attributes of its class's base classes.

简单的讲，dir 就是把这个对象有的属性(非模块对象也包括类属性，父类属性等)都列出来放到一个 list 中。若对象有\_\_dir\_\_方法，就调用该方法，否则就用默认的方法

接下来，我们使用dir函数查看下help函数内都有什么：

>>> dir(help)

['\_\_call\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_',

'\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_']

如上可以看到help()函数有这些属性

问题继续来，如果我想查看Python里所有的东西呢，不管他是属性还是方法还是对象，我都想看，咋整？

很简单，dir()函数不传递参数，如下：

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_']

这个builtins，是内置的意思，也就是这个属性里存放的都是内置对象！？看看：

>>> dir(\_\_builtins\_\_)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError',

'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '\_', '\_\_build\_class\_\_', '\_\_debug\_\_', '\_\_doc\_\_', '\_\_import\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', '\_format', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'cmp', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'xrange', 'zip']

嚯，很多， 刚才我们使用的help,dir也都在里面，随便验证一个：

>>> abs

<built-in function abs >

是吧，内置对象。

Python的内置函数和对象有很多，慢慢学吧。

下面我们来看几个有意思的函数：

1.callable

这个呢，表示的是函数的调用的意思，也就是说函数调用时用的就是他，验证一下：

Help on built-in function callable in module builtins:

callable(obj, /)

Return whether the object is callable(i.e., some kind of function).

Note that classes are callable, as are instances of classes with a

\_\_call\_\_() method.

(END)

解读如上文档，意思很明白，返回对象能不能调用，能调用就返回True，不能调用就返回False

试一下：

>>> def f():

... pass

...

>> > callable(f)

True

在试试匿名函数。

>> > callable(lambda x: x)

True

帮助文档说了类可调用，类的实例若有\_\_call\_\_属性也能调用。 那么怎么知道对象有没有\_\_call\_\_属性呢?

2.hasattr

先help一下：

Help on built-in function hasattr in module builtins:

hasattr(obj, name, / )

Return whether the object has an attribute with the given name.

This is done by calling getattr(obj, name) and catching AttributeError.

(END)

这个函数其实属于对象属性管理函数，作用很简单，返回对象是否具有某一个属性，可是！上面的这段帮助文档写的很清楚，这是通过getattr来实现的！

3.getattr

看下getattr的帮助文档：

>>> help(getattr)

Help on built-in function getattr in module builtins:

getattr(...)

getattr(object, name[, default]) -> value

Get a named attribute from an object

getattr(x, 'y') is equivalent to x.y.

When a default argument is given, it is returned when the attribute doesn't

exist

without it, an exception is raised in that case.

由上看出：getattr(object, name[, default])就是获取 object 对象的 name 属性，若 name 不

存在，若定义了 default 参数返回 default，否则抛出异常。

由上，讲了很多，其实第123行为止，就已经说完了自省。

总结一下：自省，简单点说，就是Python的自带说明，意思就是用Python的代码自己告诉这个对象是什么，有什么，以及列祖列宗是谁，孙子们又是谁。

精华一句话：Python提供的自省机制是：help() and dir().

1. **何谓小整数何谓大整数？区别在哪？**

1.小整数对象池

整数在程序中的使用非常广泛，Python为了优化速度，使用了小整数对象池， 避免为整数频繁申请和销毁内存空间。

Python 对小整数的定义是 [-5, 256] 这些整数对象是提前建立好的，不会被垃圾回收。在一个 Python 的程序中，无论这个整数处于LEGB中的哪个位置，所有位于这个范围内的整数使用的都是同一个对象。同理，单个字母也是这样的。

intern机制处理空格一个单词的复用机会大，所以创建一次，有空格创建多次，但是字符串长度大于20，就不是创建一次了。

2.大整数对象池。

终端是每次执行一次，所以每次的大整数都重新创建，而在pycharm中，每次运行是所有代码都加载都内存中，属于一个整体，所以

这个时候会有一个大整数对象池，即处于一个代码块的大整数是同一个对象。c1 和d1 处于一个代码块，而c1.b和c2.b分别有自己的代码块，所以不相等。

1. **Python中的数据类型是如何划分的？**

按照是否为数字类型：

数字类型：整型，浮点型，复数，布尔型

其他类型：字符串，列表，字典，元组，集合，字节串...

按照是否可变：

可变类型：列表，字典，集合

不可变类型：数字类型，字符串，元组，固定集合

1. **请罗列：列表推导式与字典推导式及其嵌套的语法，并说明二者区别**

列表推导式：列表推导式是用可迭代对象依次生成带有多个元素的列表的表达式

作用：

用简易方法生成列表

语法:

[表达式 for 变量 in 可迭代对象]

或

[表达式 for 变量 in 可迭代对象 if 真值表达式]

列表推导式的嵌套:

语法:

[表达式1

for 变量1 in 可迭代对象1 if 真值表达式1

for 变量2 in 可迭代对象2 if 真值表达式2]

字典推导式:

作用：

用可迭代对象生成字典

语法:

{键表达式 : 值表达式 for 变量 in 可迭代对象 [if 真值表达式]}

注意: []的内容代表可省略

字典推导式嵌套：

语法：

{键表达式：值表达式

for 变量in 可迭代对象[if 真值表达式]

For 变量 in 可迭代对象[if 真值表达式]}

1. **Python中，单双下划线有何区别？**

Python 的代码风格由 PEP 8 描述。这个文档描述了 Python 编程风格的方方面面。在遵守这个文档的条件下，不同程序员编写的 Python 代码可以保持最大程度的相似风格。这样就易于阅读，易于在程序员之间交流。

我们大家在学习Python的时候，好像很多人都不理解为什么在方法（method）前面会加好几个下划线，有时甚至两边都会加，比如像\_\_this\_\_这种。在我看到上面的文章之前，我一直以为Python中这些下划线的作用就像Golang中方法/函数的大小写一样，或是一些其他语言中的private、public的作用一样，但仔细深究，这不全是Python这样设计的初衷。

下面我们具体分析，话不多说了，来一起看看吧。

单下划线开头

我们经常看到方法或者属性前面加了单下划线，并认为它表示该方法或者属性是该类型（Python和Golang一样，不光类可以有方法，很多类型甚至基本类型也可以定义方法）的私有方法或属性。但其实在Python中不存在真正意义上的私有方法或者属性，前面加单下划线\_只是表示你不应该去访问这个方法或者属性，因为它不是API的一部分。

举个例子：

class BaseForm(StrAndUnicode):

...

def \_get\_errors(self):

"Returns an ErrorDict for the data provided for the form"

if self.\_errors is None:

self.full\_clean()

return self.\_errors

errors = property(\_get\_errors)

该代码片段来自Django源码(django/forms/forms.py)。这段代码的设计就是errors属性是对外API的一部分，如果你想获取错误详情，应该访问errors属性，而不是（也不应该）访问\_get\_errors方法。

双下划线开头

之前很多人跟我说Python中双下划线开头表示私有，我在很多地方也见到这样的说法。这样理解可能也不能说错，但这不是Python设计双下划线开头的初衷和目的，Python设计此的真正目的仅仅是为了避免子类覆盖父类的方法。

我们看个例子：

class A(object):

def \_\_method(self):

print("I'm a method in class A")

def method\_x(self):

print("I'm another method in class A\n")

def method(self):

self.\_\_method()

self.method\_x()

class B(A):

def \_\_method(self):

print("I'm a method in class B")

def method\_x(self):

print("I'm another method in class B\n")

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

print("situation 1:")

a = A()

a.method()

b = B()

b.method()

print("situation 2:")

# a.\_\_method()

a.\_A\_\_method()

执行结果：

situation 1:

I'm a method in class A

I'm another method in class A

I'm a method in class A

I'm another method in class B

situation 2:

I'm a method in class A

这里有两个点需要注意：

A类中我们定义了\_\_method()、method\_x和method()三个方法；然后我们重新定义一个类B，继承自A，并且在B类中覆写(override)了其父类的\_\_method()和method\_x方法，但是从输出结果看，B对象调用method()方法时调用了其父类A的\_\_method()方法和自己的method\_x()方法。也就是说，\_\_method()覆写没有生效，而method\_x()覆写生效了。而这也正是Python设计双下划线开头的唯一目的。

这一点也可在Python官方说明中得到答案：[https://www.python.org/dev/peps/pep-0008/#method-names-and-instance-variables](https://www.python.org/dev/peps/pep-0008/" \l "method-names-and-instance-variables" \t "https://www.jb51.net/article/_blank)。

前面我们就说了，Python中不存在真正意义上的私有变量。对于双下划线开头的方法和属性虽然我们不能直接引用，那是因为Python默认在其前面加了前缀\_类名，所以就像situation 2下面的代码，虽然我们不能用a直接访问\_\_method()，但却可以加上前缀去访问，即\_A\_\_method()。

开头结尾双下划线

一般来说像\_\_this\_\_这种开头结尾都加双下划线的方法表示这是Python自己调用的，你不要调用。比如我们可以调用len()函数来求长度，其实它后台是调用了\_\_len\_\_()方法。一般我们应该使用len，而不是直接使用\_\_len\_\_():

a = [1, 2, 3]

print(len(a))

print(a.\_\_len\_\_()) # 和上面等效

num = 10

print(num + 10)

print(num.\_\_add\_\_(10)) # 和上面等效

我们一般称\_\_len\_\_()这种方法为magic methods，一些操作符后台调用的也是也是这些magic methods，比如+后台调用的是\_\_add\_\_，-调用的是\_\_sub\_\_，所以这种机制使得我们可以在自己的类中覆写操作符（见后面例子）。另外，有的时候这种开头结尾双下划线方法仅仅是某些特殊场景的回调函数，比如\_\_init\_\_()会在对象的初始化时调用，\_\_new\_\_()会在构建一个实例的时候调用等等。下面我们看两个例子：

class CrazyNumber(object):

 def \_\_init\_\_(self, n):

 self.n = n

 def \_\_add\_\_(self, other):

 return self.n - other

 def \_\_sub\_\_(self, other):

 return self.n + other

 def \_\_str\_\_(self):

 return str(self.n)

num = CrazyNumber(10)

print(num) # output is: 10

print(num + 5) # output is: 5

print(num - 20) # output is: 30

在上面这个例子中，我们覆写了+和-操作符，将他们的功能交换了。再看个例子：

class Room(object):

 def \_\_init\_\_(self):

 self.people = []

 def add(self, person):

 self.people.append(person)

 def \_\_len\_\_(self):

 return len(self.people)

room = Room()

room.add("Igor")

print len(room) # output is: 1

这个例子中，因为我们实现了\_\_len\_\_()，所以Room对象也可以使用len函数了。

所有此类的方法都在这里有说明：documentation.

结论

使用单下划线(\_one\_underline)开头表示方法不是API的一部分，不要直接访问（虽然语法上访问也没有什么问题）。

使用双下划线开头(\_\_two\_underlines)开头表示子类不能覆写该方法。除非你真的知道你在干什么，否则不要使用这种方式。

当你想让自己定义的对象也可以像Python内置的对象一样使用Python内置的一些函数或操作符（比如len、add、+、-、==等）时，你可以定义该类方法。

当然还有些属性只在末尾加了但下划线，这仅仅是为了避免我们起的一些名字和Python保留关键字冲突，没有特殊含义。

1. **常见的格式化字符串占位符都有哪些？**

格式符为真实值预留位置，并控制显示的格式。格式符可以包含有一个类型码，用以控制显示的类型，如下:

%s    字符串 (采用str()的显示)

%r    字符串 (采用repr()的显示)

%c    单个字符

%b    二进制整数

%d    十进制整数

%i    十进制整数

%o    八进制整数

%x    十六进制整数

%e    指数 (基底写为e)

%E    指数 (基底写为E)

%f    浮点数

%F    浮点数，与上相同

%g    指数(e)\_x0010\_或浮点数 (根据显示长度)

%G    指数(E)或浮点数 (根据显示长度)

%%    字符"%"

1. **简单说说迭代器原理？**

迭代器是访问集合内元素的一种方式。迭代器对象从集合的第一个元素开始访问，直到所有的元素都被访问一遍后结束。

迭代器不能回退，只能往前进行迭代。这并不是什么很大的缺点，因为人们几乎不需要在迭代途中进行回退操作。

迭代器也不是线程安全的，在多线程环境中对可变集合使用迭代器是一个危险的操作。但如果小心谨慎，或者干脆贯彻函数式思想坚持使用不可变的集合，那这也不是什么大问题。

对于原生支持随机访问的数据结构（如tuple、list），迭代器和经典for循环的索引访问相比并无优势，反而丢失了索引值（可以使用内建函数enumerate()找回这个索引值，这是后话）。但对于无法随机访问的数据结构（比如set）而言，迭代器是唯一的访问元素的方式。

迭代器的另一个优点就是它不要求你事先准备好整个迭代过程中所有的元素。迭代器仅仅在迭代至某个元素时才计算该元素，而在这之前或之后，元素可以不存在或者被销毁。这个特点使得它特别适合用于遍历一些巨大的或是无限的集合，比如几个G的文件，或是斐波那契数列等等。这个特点被称为延迟计算或惰性求值(Lazy evaluation)。

迭代器更大的功劳是提供了一个统一的访问集合的接口。只要是实现了\_\_iter\_\_()方法的对象，就可以使用迭代器进行访问。

**创建迭代**

对对象调用一个iter()，就可以得到它的迭代器，

###语法：

iter(obj) #一个参数

iter(func, sentinel ) #两个参数

如果你传递一个参数给 iter() , 它会检查你传递的是不是一个序列, 如果是, 那么很简单:根据索引从 0 一直迭代到序列结束. 另一个创建迭代器的方法是使用类, 一个实现了 \_\_iter\_\_() 和 next() 方法的类可以作为迭代器使用.

如果是传递两个参数给 iter() , 它会重复地调用 func , 直到迭代器的下个值等于sentinel .

**关于迭代**

根本上说, 迭代器就是有一个 next() 方法的对象, 而不是通过索引来计数. 当你或是一个循环机制(例如 for 语句)需要下一个项时, 调用迭代器的 next() 方法就可以获得它. 条目全部取出后, 会引发一个 StopIteration 异常, 这并不表示错误发生, 只是告诉外部调用者, 迭代完成.

例如 for 循环遍历，这种形式的访问清晰、简洁、方便。这种迭代器的用法在 Python 中普遍而且统一。在后台，for 语句在容器对象中调用 iter() 。 该函数返回一个定义了 next() 方法的迭代器对象，它在容器中逐一访问元素。没有后续的元素时，next() 抛出一个 StopIteration 异常通知 for 语句循环结束。

例如：

For循环：

>>> for i in 'abc':

print i

a

b

c

实际运行原理：

>>> s = 'abc'

>>> it = iter(s)

>>> it

<iterator object at 0x00A1DB50>

>>> it.next()

'a'

>>> it.next()

'b'

>>> it.next()

'c'

>>> it.next()

Traceback (most recent call last):

File "<pyshell#6>", line 1, in -toplevel-

it.next()

StopIteration

**序列使用迭代器**

>>> myTuple = (123, 'xyz', 45.67)

>>> i = iter(myTuple)

>>> i.next()

>>> i.next()

'xyz'

>>> i.next()

45.67

>>> i.next()

Traceback (most recent call last):

File "", line 1, in ?

StopIteration

**字典使用迭代器**

字典和文件是另外两个可迭代的 Python 数据类型. 字典的迭代器会遍历它的键(keys).语句 for eachKey in myDict.keys() 可以缩写为 for eachKey in myDict , 例如:

>>> legends = { ('Poe', 'author'): (1809, 1849, 1976),

... ('Gaudi', 'architect'): (1852, 1906, 1987),

... ('Freud', 'psychoanalyst'): (1856, 1939, 1990)

... }

...

>>> for eachLegend in legends:

... print 'Name: %s\tOccupation: %s' % eachLegend

... print ' Birth: %s\tDeath: %s\tAlbum: %s\n' \

... % legends[eachLegend]

...

Name: Freud Occupation: psychoanalyst

Birth: 1856 Death: 1939 Album: 1990

Name: Poe Occupation: author

Birth: 1809 Death: 1849 Album: 1976

Name: Gaudi Occupation: architect

Birth: 1852 Death: 1906 Album: 1987

另外, Python 还引进了三个新的内建字典方法来定义迭代: myDict.iterkeys() (通过 keys 迭代), myDict.itervalues() (通过 values 迭代), 以及 myDicit.iteritems() (通过 key/value 对来迭代).

##### **列表解析**

###语法:

[expr for iter\_var in iterable]

这个语句的核心是 for 循环, 它迭代 iterable 对象的所有条目. 前边的 expr 应用于序列的每个成员, 最后的结果值是该表达式产生的列表. 迭代变量并不需要是表达式的一部分.

#例如:

>>> map(lambda x: x \*\* 2, range(6))

[0, 1, 4, 9, 16, 25]

#可以由下面的列表解析替换：

>>> [x \*\* 2 for x in range(6)] #可以写成[(x \*\* 2) for x in range(6)]，更清楚

[0, 1, 4, 9, 16, 25]

列表解析的表达式可以取代内建的 map() 函数以及 lambda , 而且效率更高. 结合if语句，列表解析还提供了一个扩展版本的语法:

[expr for iter\_var in iterable if cond\_expr]

这个语法在迭代时会过滤/捕获满足条件表达式 cond\_expr 的序列成员.

###样例：序列中的奇数

>>> seq = [11, 10, 9, 9, 10, 10, 9, 8, 23, 9, 7, 18, 12, 11, 12]>>> [x for x in seq if x % 2]

[11, 9, 9, 9, 23, 9, 7, 11]

#如下这样一个数据文件 hhga.txt , 需要计算出所有非空白字符的数目:

>>> f = open('hhga.txt', 'r')>>> len([word for line in f for word in line.split()]) #嵌套for循环

1. **编码模拟实现迭代器？**

class MyListIterator:

'''此类定义一个迭代器类，用于生成能够访问MyList对象的

迭代器'''

def \_\_init\_\_(self, lst\_data):

print("迭代器已经创建")

self.data = lst\_data # 绑定要访问的数据列表

self.cur\_pos = 0 # 设置迭代器的起始位置为0

def \_\_next\_\_(self):

'''此方法用来访问可迭代对象的数据，如果没有

数据时触发 StopIterator 异常来通知调用者停止迭代，

即"迭代器协议" '''

# 判断是否索引越界,如果已越界则触发异常停止迭代

if self.cur\_pos >= len(self.data):

raise StopIteration

index = self.cur\_pos

self.cur\_pos += 1 # 将当前位置向后移动准备下次获取

return self.data[index] # 返回当前位置的数据

1. **解释一下AOP？什么叫面向切面编程？**

AOP为Aspect Oriented Programming的缩写，意为：[面向切面编程](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%88%87%E9%9D%A2%E7%BC%96%E7%A8%8B/6016335" \t "https://baike.baidu.com/item/AOP/_blank)，通过[预编译](https://baike.baidu.com/item/%E9%A2%84%E7%BC%96%E8%AF%91/3191547" \t "https://baike.baidu.com/item/AOP/_blank)方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是[OOP](https://baike.baidu.com/item/OOP" \t "https://baike.baidu.com/item/AOP/_blank)的延续，是软件开发中的一个热点，也是[Spring](https://baike.baidu.com/item/Spring" \t "https://baike.baidu.com/item/AOP/_blank)框架中的一个重要内容，是[函数式编程](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0%E5%BC%8F%E7%BC%96%E7%A8%8B/4035031" \t "https://baike.baidu.com/item/AOP/_blank)的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的[耦合度](https://baike.baidu.com/item/%E8%80%A6%E5%90%88%E5%BA%A6/2603938" \t "https://baike.baidu.com/item/AOP/_blank)降低，提高程序的可重用性，同时提高了开发的效率。

Python中的装饰器就应用于有切面需求的场景。

1. **说一说我们常说的‘鸭子类型’的含义？**

1.python不支持多态也用不到多态，多态的概念是应用于java和C# 这一类强类型语言中，而Python崇尚鸭子类型（Duck Typing）

2.鸭子类型：是一种动态类型的风格。一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。这个概念的名字来源于由James Whitcomb Riley提出的鸭子测试，“鸭子测试”可以这样表述：“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

3.在鸭子类型中，关注的不是对象类型本身，而是它是如何使用的。我们可以编写一个函数，它接受一个类型为鸭的对象，并调用它的走和叫方法。在使用鸭子类型的语言中，这样的一个函数可以接受一个任意类型的对象，并调用它的走和叫方法。如果这些需要被调用的方法不存在，那么将引发一个运行时错误。

class F1:

pass

# 假设，S1是我们的正统类，它继承于根正苗红的F1，是我们的正统类

class S1(F1):

def show(self):

print('S1.show')

# S2是路人甲，是个歪瓜裂枣，但是他自己也有一个叫show的方法。

class S2:

def show(self):

print('S2.show')

# 在Java或C#中定义函数参数时，必须指定参数的类型，也即是说，我们如果用

# Java写下面的Func，需要告知，obj是F1类还是其他什么东西。

# 如果限定了F1，那么S2是不可以被采纳的。

# 然而，在Python中，一切都是Obj，它不care你到底是什么类，直接塞进去就可以

def Func(obj):

"""Func函数需要接收一个F1类型或者F1子类的类型"""

obj.show()

s1\_obj = S1()

Func(s1\_obj) # 在Func函数中传入S1类的对象 s1\_obj，执行 S1 的show方法，结果：S1.show

s2\_obj = S2()

Func(s2\_obj) # 在Func函数中传入Ss类的对象 ss\_obj，执行 Ss 的show方法，结果：S2.show

1. **对比Python与JavaScript，说一说作用域方面，有何区别？**

Python作用域：

局部作用域(函数内) Local(function) L

外部嵌套函数作用域 Enclosing function locals E

函数定义所在模块的作用域 Global(module) G

python 内置模块的作用域 Builtin(python) B

JavaScript作用域：

全局作用域

函数作用域

相同点：

全局作用域下均为全局变量

函数作用域下均为局部变量

函数嵌套会划分作用域

不同点：

Python函数形成嵌套，各级函数可以逐级引用任意局部变量但必须加nonlocal语句声明，而JavaScript中形成的函数嵌套，各级函数可直接逐级引用任意局部变量

JavaScript中的作用域有作用域链的概念

1. **闭包最典型的应用是什么？面向对象中用到装饰器的地方有哪些？举例说明？**

闭包最典型的应用为装饰器。

装饰器：修改其他函数功能的函数，实现不改变原函数的情况下，调用原函数实现其他的功能。

装饰器的语法：

@装饰器函数名

Def 被装饰函数名():

...//方法体

装饰器可以装饰一个函数，也可以装饰一个类甚至一个对象。

在Python面向对象中，用到装饰器的地方，比如：

创建类方法，需要添加装饰器@classmethod

创建静态方法，需要添加装饰器@staticmethod

[创建抽象方法，需要添加装饰器@abc.abstractmethod](mailto:创建抽象方法，需要添加装饰器@abc.abstractmethod)

...

1. **来解释一下‘函数式编程’到底是什么意思？**

函数是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

我们首先要搞明白计算机（Computer）和计算（Compute）的概念。

在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

1. **详细解释深浅拷贝。对于不可变类型，采用何种拷贝机制？为什么？**

赋值：变量在内存中指向同一个对象。

如果是可变(mutable)类型，比如列表，修改其中一个，另一个必定改变

如果是不可变类型(immutable),比如字符串，修改了其中一个，另一个并不会变

浅拷贝：对于包含多层对象关系引用的对象，其第一层拷贝对象+引用，从第二层开始只拷贝引用关系。

深拷贝：无论对象中有多少层对象引用关系，统统都是拷贝对象+引用，彻底实现对象间的分离。

深浅拷贝的概念只适用于可变类型的对象。对于不可变类型，只有深拷贝，没有浅拷贝。

深浅拷贝对比各自优势：

深拷贝：彻底实现对象间的分离，但内存空间占用更大。

浅拷贝：节省内存空间，但是对象间分离做的不好。

1. **如何实现元组和列表的转换？**

通过列表和元组的构造函数进行转换

列表-->元组：tuple([参数：列表])

元组-->列表：list([参数：元组])

1. **使用两种以上的方法实现：去除一个列表中重复的元素**

方法一：转换为集合即可

li = [1, 2, 3, 4, 1, 2]

s = set(li)

li = list(s)

print li #[1, 2, 3, 4]

方法二：使用字典的fromkeys()方法

li = [1, 2, 3, 4, 1, 2]

d = {}.fromkeys(li)

print d.keys() #[1, 2, 3, 4]

方法三：in not in运算符实现

li = [1,2,1,2,3,4,2,1]

Newl = []

For i in li:

If i not in newl:

Newl.append(i)

Print(newl)

1. **异常处理语句有什么劣势？是否有替代方案？有的话，方案是什么？**

与python异常相关的关键字

raise：手动抛出/引发异常：raise [exception[,data]  
try/except：捕获异常并处理  
pass：忽略异常  
as：定义异常实例(except IOError as e)  
finally：无论是否出现异常，都执行的代码]  
else：如果try中的语句没有引发异常，则执行else中的语句  
except Exception as error

python中的异常类型

StandardError类：如果程序上出现逻辑错误， 将引发该异常。StandardError类是所有内敛异常的基类，放置在默认的命名空间中，因此使用IOEroor, EOFError, ImportError等类，不需要导入exception模块。

StopIteration类：判断循环是否执行到尾部，如果循环到尾部，则抛出该异常。  
GeneratorExit类：是由Generator函数引发的异常，当调用close()时引发该异常。  
Warning类：表示程序中的代码引起的警告。

异常语句语法规范：

try：

语句1

except  [exception1(,exception2...),[data…]]:

      语句2

else:

      语句3

该种异常处理语法的规则是：

执行try下的语句，如果引发异常，则执行过程会跳到第一个except语句。

如果第一个except中定义的异常与引发的异常匹配，则执行该except中的语句。

如果引发的异常不匹配第一个except，则会搜索第二个except，允许编写的except数量没有限制。

如果所有的except都不匹配，则异常会传递到下一个调用本代码的最高层try代码中。

如果没有发生异常，则执行else块代码。

异常语句语法规范：

try:

      语句1

  finally：

      语句2

该语句的执行规则是：

执行try下的代码。

如果发生异常，在该异常传递到下一级try时，执行finally中的代码。

如果没有发生异常，则执行finally中的代码。

第二种try语法在无论有没有发生异常都要执行代码的情况下是很有用的。例如我们在python中打开一个文件进行读写操作，我在操作过程中不管是否出现异常，最终都是要把该文件关闭的。这两种形式相互冲突，使用了一种就不允许使用另一种，而功能又各异

所以，一般情况下，finally里面执行的都是一些清理工作，比如：关闭文件描述符，释放锁等

注意，finally中，如果出现异常，外部如果没有相应的捕获机制，该异常会层层抛出，直到最顶端，然后解释器停止。一般在外层再加一次try except异常捕获

异常语句语法规范：

With ...... as.....

我们平时在使用类似文件的流对象时，使用完毕后要调用close方法关闭，很麻烦。这里with…as语句提供了一个非常方便的替代方法：open打开文件后将返回的文件流对象赋值给f，然后在with语句块中使用。with语句块完毕之后，会隐藏地自动关闭文件。

如果with语句或语句块中发生异常，会调用默认的异常处理器处理，但文件还是会正常关闭。

这种情况下，会抛出异常，最后的print不执行。

1. **什么叫可迭代对象？range()函数返回的值是什么类型呢？**

大部分对象都是可迭代，只要实现了\_\_iter\_\_方法的对象就是可迭代对象。

\_\_iter\_\_方法会返回迭代器（iterator）本身，例如：

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

>>> lst.\_\_iter\_\_()

<listiterator object at 0x7f97c549aa50>

Python提供一些语句和关键字用于访问可迭代对象的元素，比如for循环、列表解析、逻辑操作符等。

判断一个对象是否是可迭代对象：

>>> from collections import Iterable  # 只导入Iterable方法

>>> isinstance('abc', Iterable)

True

>>> isinstance(1, Iterable)

False

>>> isinstance([], Iterable)

True

这里的isinstance()函数用于判断对象类型，后面会讲到。

可迭代对象一般都用for循环遍历元素，也就是能用for循环的对象都可称为可迭代对象。

迭代器（Iterator）

具有next方法的对象都是迭代器。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。

使用迭代器的好处：

1）如果使用列表，计算值时会一次获取所有值，那么就会占用更多的内存。而迭代器则是一个接一个计算。

2）使代码更通用、更简单。

迭代器规则

回忆下在Python数据类型章节讲解到字典迭代器方法，来举例说明下迭代器规则：

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

>>> d.iteritems()

<dictionary-itemiterator object at 0x7f97c3b1bcb0>

# 判断是否是迭代器

>>> from collections import Iterator

>>> isinstance(d, Iterator)

False

>>> isinstance(d.iteritems(), Iterator)

True

# 使用next方法。

>>> iter\_items = d.iteritems()

>>> iter\_items.next()

('a', 1)

>>> iter\_items.next()

('c', 3)

>>> iter\_items.next()

('b', 2)

由于字典是无序的，所以显示的是无序的，实际是按照顺序获取的下一个元素。

在Python3中：print(range(10))，其结果并不是[0,1,2,3,4,5,6,7,8,9]，而是range(10)

，为什么呢？

先看官网原话：

In many ways the object returned by range() behaves as if it is a list, but in fact it isn’t. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn’t really make the list, thus saving space.

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the for statement is such an iterator. The function list() is another; it creates lists from iterables:

翻译：

可以看到上面这个很奇怪，在很多种情况下，range()函数返回的对象的行为都很像一个列表，但是它确实不是一个列表，它只是在迭代的情况下返回指定索引的值，但是它并不会在内存中真正产生一个列表对象，这样也是为了节约内存空间。

我们称这种对象是可迭代的，或者是可迭代对象，还有一种对象叫迭代器，它们需要从一个可迭代对象中连续获取指定索引的值，一直到索引结束。list()函数就是这样一个迭代器，它可以把range()函数返回的对象变成一个列表。

总结：

range() 函数返回的是一个可迭代对象（类型是对象），而不是列表类型， 所以打印的时候不会打印列表。

list() 函数是对象迭代器，把对象转为一个列表。返回的变量类型为列表。

1. **如何用Python进行查询和替换一个文本字符串？**

可以使用sub()方法来进行查询和替换，sub方法的格式为：sub(replacement, string[, count=0])

replacement是被替换成的文本

string是需要被替换的文本

count是一个可选参数，指最大被替换的数量

例子：

import re  
 p = re.compile(‘(blue|white|red)’)  
 print(p.sub(‘colour’,'blue socks and red shoes’))  
 print(p.sub(‘colour’,'blue socks and red shoes’, count=1))

输出：

colour socks and colour shoes  
 colour socks and red shoes

subn()方法执行的效果跟sub()一样，不过它会返回一个二维数组，包括替换后的新的字符串和总共替换的数量

例如：

import re  
 p = re.compile(‘(blue|white|red)’)  
 print(p.subn(‘colour’,'blue socks and red shoes’))  
 print(p.subn(‘colour’,'blue socks and red shoes’, count=1))

输出：

(‘colour socks and colour shoes’, 2)  
 (‘colour socks and red shoes’, 1)

1. **说一说：match()和search()的区别在哪？**

match（）函数只检测RE是不是在string的开始位置匹配， search()会扫描整个string查找匹配, 也就是说match（）只有在0位置匹配成功的话才有返回，如果不是开始位置匹配成功的话，match()就返回none

1. **简要列举常用的内建模块都有哪些？**

打开终端，进入交互模式，输入dir(\_\_builtins\_\_)即可查看：

dir(\_\_builtins\_\_)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError',

'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '\_', '\_\_build\_class\_\_', '\_\_debug\_\_', '\_\_doc\_\_', '\_\_import\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', '\_format', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'cmp', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'xrange', 'zip']

1. **详细阐述Python的垃圾回收机制**

一、垃圾回收机制

Python中的垃圾回收是以引用计数为主，分代收集为辅。引用计数的缺陷是循环引用的问题。  
 在Python中，如果一个对象的引用数为0，Python虚拟机就会回收这个对象的内存。

**class** ClassA():  
 **def** \_\_init\_\_(self):  
 print **'object born,id:%s'**%str(hex(id(self)))  
 **def** \_\_del\_\_(self):  
 print **'object del,id:%s'**%str(hex(id(self)))  
  
**def** f1():  
 **while True**:  
 c1=ClassA()  
 **del** c1

执行f1()会循环输出这样的结果，而且进程占用的内存基本不会变动

object born,id:0x237cf58

object del,id:0x237cf58

c1=ClassA()会创建一个对象，放在0x237cf58内存中，c1变量指向这个内存，这时候这个内存的引用计数是1  
 del c1后，c1变量不再指向0x237cf58内存，所以这块内存的引用计数减一，等于0，所以就销毁了这个对象，然后释放内存。

导致引用计数+1的情况

对象被创建，例如a=23

对象被引用，例如b=a

对象被作为参数，传入到一个函数中，例如func(a)

对象作为一个元素，存储在容器中，例如list1=[a,a]

导致引用计数-1的情况

对象的别名被显式销毁，例如del a

对象的别名被赋予新的对象，例如a=24

一个对象离开它的作用域，例如f函数执行完毕时，func函数中的局部变量（全局变量不会）

对象所在的容器被销毁，或从容器中删除对象

Demo:

**def** func(c, d):  
 print  
 **'in func function'**, sys.getrefcount(c) - 1  
  
print  
**'init'**, sys.getrefcount(11) - 1  
a = 11  
print  
**'after a=11'**, sys.getrefcount(11) - 1  
b = a  
print  
**'after b=1'**, sys.getrefcount(11) - 1  
func(11)  
print  
**'after func(a)'**, sys.getrefcount(11) - 1  
list1 = [a, 12, 14]  
print  
**'after list1=[a,12,14]'**, sys.getrefcount(11) - 1  
a = 12  
print  
**'after a=12'**, sys.getrefcount(11) - 1  
**del** a  
print  
**'after del a'**, sys.getrefcount(11) - 1  
**del** b  
print  
**'after del b'**, sys.getrefcount(11) - 1  
*# list1.pop(0)  
# print 'after pop list1',sys.getrefcount(11)-1***del** list1  
print  
**'after del list1'**, sys.getrefcount(11) - 1  
输出：  
  
init  
24  
after  
a = 11  
25  
after  
b = 1  
26  
**in** func  
function  
28  
after  
func(a)  
26  
after  
list1 = [a, 12, 14]  
27  
after  
a = 12  
26  
after  
**del** a  
26  
after  
**del** b  
25  
after  
**del** list1  
24

1. 循环引用导致内存泄漏

**def** f2():  
 **while True**:  
 c1 = ClassA()  
 c2 = ClassA()  
 c1.t = c2  
 c2.t = c1  
 **del** c1  
 **del** c2  
  
执行f2()，进程占用的内存会不断增大。  
  
object  
born, id: 0x237cf30  
object  
born, id: 0x237cf58

创建了c1，c2后，0x237cf30（c1对应的内存，记为内存1）,0x237cf58（c2对应的内存，记为内存2）这两块内存的引用计数都是1，执行c1.t=c2和c2.t=c1后，这两块内存的引用计数变成2.  
在del c1后，内存1的对象的引用计数变为1，由于不是为0，所以内存1的对象不会被销毁，所以内存2的对象的引用数依然是2，在del c2后，同理，内存1的对象，内存2的对象的引用数都是1。  
虽然它们两个的对象都是可以被销毁的，但是由于循环引用，导致垃圾回收器都不会回收它们，所以就会导致内存泄露。

1. 垃圾回收

deff3():  
 *# print gc.collect()* c1=ClassA()  
 c2=ClassA()  
 c1.t=c2  
 c2.t=c1  
 **del** c1  
 **del** c2  
 print gc.garbage  
 print gc.collect() *#显式执行垃圾回收* print gc.garbage  
 time.sleep(10)  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 gc.set\_debug(gc.DEBUG\_LEAK) *#设置gc模块的日志* f3()  
输出：  
  
gc: uncollectable <ClassA instance at 0230E918>  
gc: uncollectable <ClassA instance at 0230E940>  
gc: uncollectable <dict 0230B810>  
gc: uncollectable <dict 02301ED0>  
object born,id:0x230e918  
object born,id:0x230e940  
4

垃圾回收后的对象会放在gc.garbage列表里面

gc.collect()会返回不可达的对象数目，4等于两个对象以及它们对应的dict

有三种情况会触发垃圾回收：  
 1.调用gc.collect(),  
 2.当gc模块的计数器达到阀值的时候。  
 3.程序退出的时候

四、gc模块常用功能解析

gc模块提供一个接口给开发者设置垃圾回收的选项。上面说到，采用引用计数的方法管理内存的一个缺陷是循环引用，而gc模块的一个主要功能就是解决循环引用的问题。

常用函数：

gc.set\_debug(flags)

设置gc的debug日志，一般设置为gc.DEBUG\_LEAK

gc.collect([generation])  
 显式进行垃圾回收，可以输入参数，0代表只检查第一代的对象，1代表检查一，二代的对象，2代表检查一，二，三代的对象，如果不传参数，执行一个full collection，也就是等于传2。返回不可达（unreachable objects）对象的数目

gc.set\_threshold(threshold0[, threshold1[, threshold2])  
 设置自动执行垃圾回收的频率。

gc.get\_count()  
 获取当前自动执行垃圾回收的计数器，返回一个长度为3的列表

五、gc模块的自动垃圾回收机制

必须要import gc模块，并且is\_enable()=True才会启动自动垃圾回收。这个机制的主要作用就是发现并处理不可达的垃圾对象。垃圾回收=垃圾检查+垃圾回收  
 在Python中，采用分代收集的方法。把对象分为三代，一开始，对象在创建的时候，放在一代中，如果在一次一代的垃圾检查中，改对象存活下来，就会被放到二代中，同理在一次二代的垃圾检查中，该对象存活下来，就会被放到三代中。

gc模块里面会有一个长度为3的列表的计数器，可以通过gc.get\_count()获取。  
例如(488,3,0)，其中488是指距离上一次一代垃圾检查，Python分配内存的数目减去释放内存的数目，注意是内存分配，而不是引用计数的增加。例如：

print gc.get\_count() *# (590, 8, 0)*a = ClassA()  
print gc.get\_count() *# (591, 8, 0)***del** a  
print gc.get\_count() *# (590, 8, 0)*

六、应用

1.项目中避免循环引用

2.引入gc模块，启动gc模块的自动清理循环引用的对象机制

3.由于分代收集，所以把需要长期使用的变量集中管理，并尽快移到二代以后，减少GC检查时的消耗

4.gc模块唯一处理不了的是循环引用的类都有\_\_del\_\_方法，所以项目中要避免定义\_\_del\_\_方法，如果一定要使用该方法，同时导致了循环引用，需要代码显式调用gc.garbage里面的对象的\_\_del\_\_来打破僵局

1. **Python的2系列版本和3系列版本都有哪些不同？**

1.性能   
 Py3.0运行 pystone benchmark的速度比Py2.5慢30%。Guido认为Py3.0有极大的优化空间，在字符串和整形操作上可以取得很好的优化结果。   
 Py3.1性能比Py2.5慢15%，还有很大的提升空间。   
2.编码   
 Py3.X源码文件默认使用utf-8编码，这就使得以下代码是合法的：   
    >>> 中国 = 'china'   
    >>>print(中国)   
    china   
3. 语法   
 1）去除了<>，全部改用!=   
 2）去除``，全部改用repr()   
 3）关键词加入as 和with，还有True,False,None   
 4）整型除法返回浮点数，要得到整型结果，请使用//   
 5）加入nonlocal语句。使用noclocal x可以直接指派外围（非全局）变量   
 6）去除print语句，加入print()函数实现相同的功能。同样的还有 exec语句，已经改为exec()函数   
   例如：   
     2.X: print "The answer is", 2\*2   
     3.X: print("The answer is", 2\*2)   
     2.X: print x,                              # 使用逗号结尾禁止换行   
     3.X: print(x, end=" ")                     # 使用空格代替换行   
     2.X: print                                 # 输出新行   
     3.X: print()                               # 输出新行   
     2.X: print >>sys.stderr, "fatal error"   
     3.X: print("fatal error", file=sys.stderr)   
     2.X: print (x, y)                          # 输出repr((x, y))   
     3.X: print((x, y))                         # 不同于print(x, y)!   
 7）改变了顺序操作符的行为，例如x<y，当x和y类型不匹配时抛出TypeError而不是返回随即的 bool值    
 8）输入函数改变了，删除了raw\_input，用input代替：   
   2.X:guess = int(raw\_input('Enter an integer : ')) # 读取键盘输入的方法   
   3.X:guess = int(input('Enter an integer : '))

9）去除元组参数解包。不能def(a, (b, c)):pass这样定义函数了   
 10）新式的8进制字变量，相应地修改了oct()函数。   
   2.X的方式如下：   
     >>> 0666   
     438   
     >>> oct(438)   
     '0666'   
   3.X这样：   
     >>> 0666   
     SyntaxError: invalid token (<pyshell#63>, line 1)   
     >>> 0o666   
     438   
     >>> oct(438)   
     '0o666'   
 11）增加了 2进制字面量和bin()函数   
    >>> bin(438)   
    '0b110110110'   
    >>> \_438 = '0b110110110'   
    >>> \_438   
    '0b110110110'   
 12）扩展的可迭代解包。在Py3.X 里，a, b, \*rest = seq和 \*rest, a = seq都是合法的，只要求两点：rest是list 对象和seq是可迭代的。   
 13）新的super()，可以不再给super()传参数，   
    >>> class C(object):   
          def \_\_init\_\_(self, a):   
             print('C', a)   
    >>> class D(C):   
          def \_\_init(self, a):   
             super().\_\_init\_\_(a) # 无参数调用super()   
    >>> D(8)   
    C 8   
    <\_\_main\_\_.D object at 0x00D7ED90>   
 14）新的metaclass语法：   
    class Foo(\*bases, \*\*kwds):   
      pass   
 15）支持class decorator。用法与函数decorator一样：   
    >>> def foo(cls\_a):   
          def print\_func(self):   
             print('Hello, world!')   
          cls\_a.print = print\_func   
          return cls\_a   
    >>> @foo   
    class C(object):   
      pass   
    >>> C().print()   
    Hello, world!   
 class decorator可以用来玩玩狸猫换太子的大把戏。更多请参阅PEP 3129   
4. 字符串和字节串   
 1）现在字符串只有str一种类型，但它跟2.x版本的unicode几乎一样。

2）关于字节串，请参阅“数据类型”的第2条目   
5.数据类型   
 1）Py3.X去除了long类型，现在只有一种整型——int，但它的行为就像2.X版本的long   
 2）新增了bytes类型，对应于2.X版本的八位串，定义一个bytes字面量的方法如下：   
    >>> b = b'china'   
    >>> type(b)   
    <type 'bytes'>   
 str对象和bytes对象可以使用.encode() (str -> bytes) or .decode() (bytes -> str) 方法相互转化。   
    >>> s = b.decode()   
    >>> s   
    'china'   
    >>> b1 = s.encode()   
    >>> b1   
    b'china'   
 3）dict的.keys()、.items 和.values()方法返回迭代器，而之前的iterkeys()等函数都被废弃。同时去掉的还有 dict.has\_key()，用 in替代它吧   
6.面向对象   
 1）引入抽象基类（Abstraact Base Classes，ABCs）。   
 2）容器类和迭代器类被ABCs化，所以cellections模块里的类型比Py2.5多了很多。   
    >>> import collections   
    >>> print('\n'.join(dir(collections)))   
    Callable   
    Container   
    Hashable   
    ItemsView   
    Iterable   
    Iterator   
    KeysView   
    Mapping   
    MappingView   
    MutableMapping   
    MutableSequence   
    MutableSet   
    NamedTuple   
    Sequence   
    Set   
    Sized   
    ValuesView   
    \_\_all\_\_   
    \_\_builtins\_\_   
    \_\_doc\_\_   
    \_\_file\_\_   
    \_\_name\_\_   
    \_abcoll   
    \_itemgetter   
    \_sys   
    defaultdict   
    deque   
 另外，数值类型也被ABCs化。关于这两点，请参阅 PEP 3119和PEP 3141。   
 3）迭代器的next()方法改名为\_\_next\_\_()，并增加内置函数next()，用以调用迭代器的\_\_next\_\_()方法   
 4）增加了@abstractmethod和 @abstractproperty两个 decorator，编写抽象方法（属性）更加方便。   
7.异常   
 1）所以异常都从 BaseException继承，并删除了StardardError   
 2）去除了异常类的序列行为和.message属性   
 3）用 raise Exception(args)代替 raise Exception, args语法   
 4）捕获异常的语法改变，引入了as关键字来标识异常实例，在Py2.5中：   
    >>> try:   
    ...    raise NotImplementedError('Error')   
    ... except NotImplementedError, error:

    ...    print error.message   
    ...   
    Error   
 在Py3.0中：   
    >>> try:   
          raise NotImplementedError('Error')   
        except NotImplementedError as error: #注意这个 as   
          print(str(error))   
    Error   
 5）异常链，因为\_\_context\_\_在3.0a1版本中没有实现   
8.模块变动   
 1）移除了cPickle模块，可以使用pickle模块代替。最终我们将会有一个透明高效的模块。   
 2）移除了imageop模块   
 3）移除了 audiodev, Bastion, bsddb185, exceptions, linuxaudiodev, md5, MimeWriter, mimify, popen2,  rexec, sets, sha, stringold, strop, sunaudiodev, timing和xmllib模块   
 4）移除了bsddb模块(单独发布，可以从<http://www.jcea.es/programacion/pybsddb.htm>获取)   
 5）移除了new模块   
 6）os.tmpnam()和os.tmpfile()函数被移动到tmpfile模块下   
 7）tokenize模块现在使用bytes工作。主要的入口点不再是generate\_tokens，而是 tokenize.tokenize()   
9.其它   
 1）xrange() 改名为range()，要想使用range()获得一个list，必须显式调用：   
    >>> list(range(10))   
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]   
 2）bytes对象不能hash，也不支持 b.lower()、b.strip()和b.split()方法，但对于后两者可以使用 b.strip(b’\n\t\r \f’)和b.split(b’ ‘)来达到相同目的   
 3）zip()、map()和filter()都返回迭代器。而apply()、 callable()、coerce()、 execfile()、reduce()和reload ()函数都被去除了。现在可以使用hasattr()来替换 callable(). hasattr()的语法如：hasattr(string, '\_\_name\_\_')

4）string.letters和相关的.lowercase和.uppercase被去除，请改用string.ascii\_letters 等   
 5）如果x < y的不能比较，抛出TypeError异常。2.x版本是返回伪随机布尔值的   
 6）\_\_getslice\_\_系列成员被废弃。a[i:j]根据上下文转换为a.\_\_getitem\_\_(slice(I, j))或 \_\_setitem\_\_和 \_\_delitem\_\_调用   
 7）file类被废弃，在Py2.5中：   
    >>> file   
    <type 'file'>   
 在Py3.X中：   
    >>> file   
    Traceback (most recent call last):   
    File "<pyshell#120>", line 1, in <module>   
       file   
    NameError: name 'file' is not defined

1. **‘猴子补丁’是什么？说一说其优缺点？**

Monkey Patch（猴子补丁）：属性在运行时的动态替换，叫做猴子补丁（Monkey Patch）。

为什么叫猴子补丁？属性的运行时替换和猴子也没什么关系，关于猴子补丁的由来网上查到两种说法：

1. 这个词原来为Guerrilla Patch，杂牌军、游击队，说明这部分不是原装的，在英文里guerilla发音和gorllia(猩猩)相似，再后来就写了monkey(猴子)。  
    2，还有一种解释是说由于这种方式将原来的代码弄乱了(messing with it)，在英文里叫monkeying about(顽皮的)，所以叫做Monkey Patch。

猴子补丁的叫法有些莫名其妙，只要和“模块运行时替换的功能”对应就行了。

Monkey patch就是在运行时对已有的代码进行修改，达到hot patch的目的。

Eventlet中大量使用了该技巧，以替换标准库中的组件，比如socket。

首先来看一下最简单的monkey patch的实现。

**class** Foo(object):  
 **def** bar(self):  
 print(**'Foo.bar'**)  
**def** bar(self):  
 print(**'Modified bar'**)  
Foo().bar()  
Foo.bar = bar  
Foo().bar()  
*# Foo.bar  
# Modified bar*

由于Python中的名字空间是开放，通过dict来实现，所以很容易就可以达到patch的目的。

1. **你了解PEP8规范么？简单说说？**

PEP8定义了Python在编码中的各项规范，具体详细的规则参见官方文档，链接：

<https://legacy.python.org/dev/peps/pep-0008/>

1. **用过哪些debug工具？**

常用pdb命令

pdb命令和gdb差不多，最常用的命令如下：

(1) 单步调试（进入函数）：s(tep)

(2) 单步调试（不进入函数）：n(ext)

(3) 继续往后执行，直到下个断点：c(ont(inue))

(4) 运行到函数结束：r(eturn)

(5) 运行到当前循环结束：unt(il)

(6) 设置断点：b(reak) 文件名:行号（或行号，或函数名）

(7) 显示当前调用关系：w(here)

(8) 显示当前代码段：l(ist)

(9) 显示变量：p(rint) 变量名

(10) 显示当前函数的参数：a(rgs)

(11) 显示帮助信息：h(elp)

(12) 退出：q(uit)

1. **Xrange()和range()的区别在哪呢？**

range([start,] stop[, step])，根据start与stop指定的范围以及step设定的步长，生成一个序列。

比如：

>>>range(5)[0, 1, 2, 3, 4]

>>>range(1,5)[1, 2, 3, 4]

>>>range(0,6,2)[0, 2, 4]

xrange 用法与 range 完全相同，所不同的是生成的不是一个list对象，而是一个生成器。

>>>xrange(5)xrange(5)

>>>list(xrange(5))[0, 1, 2, 3, 4]

>>>xrange(1,5)xrange(1, 5)

>>>list(xrange(1,5))[1, 2, 3, 4]

>>>xrange(0,6,2)xrange(0, 6, 2)

>>>list(xrange(0,6,2))[0, 2, 4]

1. **Python中导入一个模块或一个包，都有哪些方式呢？**

直接导入所有：import 包名/模块名

单个导入：from 包名/模块名 import 变量/函数/对象

1. **输入三个数，输出三个数中的最大值。要求：只使用一行有效代码实现！**

**class** Is\_Max:  
 **def** \_\_init\_\_(self):  
 self.a = int(input(**"请输入第一个数字："**))  
 self.b = int(input(**"请输入第二个数字："**))  
 self.c = int(input(**"请输入第三个数字："**))  
 **def** is\_max(self):  
 print((self.a **if** self.a > self.c **else** self.c) **if** self.a > self.b **else** (self.b **if** self.b > self.c **else** self.c))  
is\_max = Is\_Max()  
is\_max.is\_max()

1. **变态青蛙问题：**
   1. **一个青蛙，一次可以跳上1级台阶，也可以跳上2级台阶。问青蛙跳上一个n 级台阶总共有多少种跳法？**
   2. **一个青蛙，一次可以跳上1级台阶，也可以跳上2级台阶，也可以跳上3级台 阶，…，也可以跳上n级台阶。问青蛙跳上一个n级台阶总共有多少种跳法？**

*#此代码示例为：将两种青蛙跳问题采用面向对象思想进行实现***import** sys  
  
*# 单例模式***class** Fib:  
 **def** \_\_init\_\_(self):  
 print(**"现有如下两个问题："**)  
 print(**"1.一只青蛙一次可以跳上１级台阶，也可以跳上２级。求该青蛙跳上一个n级台阶总共有多少种跳法"**)  
 print(**"2.一只青蛙一次可以跳上１级台阶，可以跳上２级台阶，......，也可以跳上n级台阶。求该青蛙跳上一个n级台阶总共有多少种跳法"**)  
 print(**"--------------------------------------------------------------------------------------"**)  
 print(**"您要实现哪个问题呢？"**)  
 self.choose = int(input(**"请选择："**))  
 **try**:  
 self.num = int(input(**"请输入青蛙跳的台阶次数："**))  
 **except** ValueError:  
 print(**"您输入的台阶数不合法，只能输入整型，请重新输入！"**)  
 sys.exit()  
  
 **else**:  
 print(**"值合法！程序继续运行！"**)  
  
  
 **def** work\_one(self,num):  
 **if** num == 0:  
 **return "您输入的台阶数不合法，请重新输入！"  
 elif** num == 1:  
 **return** 1  
 **elif** num == 2:  
 **return** 2  
 **else**:  
 **return** self.work\_one(num - 1) + self.work\_one(num - 2)  
  
 **def** work\_two(self,num):  
 **if** num == 0:  
 **return** 1  
 **elif** num == 1:  
 **return** 1  
 **else**:  
 **return** 2 \* self.work\_two(num-1)  
  
 **def** work(self,choose):  
 **if** choose == 1:  
 print(**"第一个问题的答案："**)  
 print(**"青蛙跳上一个%d级台阶总共有%d种方法。"**%(self.num,self.work\_one(self.num)))  
 **if** choose == 2:  
 print(**"第二个问题的答案："**)  
 print(**"青蛙跳上一个%d级台阶总共有%d种方法。"** % (self.num, self.work\_two(self.num)))  
  
fib = Fib()  
  
fib.work(fib.choose)

1. **不死兔子问题：有一对兔子，从第三个月开始，每月都生一对兔子，小兔子长到第三个月时也开始每月都生一对兔子。假设兔子都不死，问一年之后共有多少只兔子？**

*# 递归算法实现  
# 递归算法实现计算每月兔子数***def** fib\_year(m):  
 **if** m == 1:  
 **return** 1  
 **elif** m == 2:  
 **return** 1  
 **else**:  
 **return** fib\_year(m-1) + fib\_year(m-2)  
*# 功能函数：计算一年之后总共的兔子数***def** rib\_num():  
 l = []  
 **for** i **in** range(1,13):  
 num = fib\_year(i)  
 l.append(num)  
 print(**'一年之后一共有：%d只兔子。'**%(2\*sum(l)))  
rib\_num()

1. **有一堆球，白球有4个，黄球有8个，黑球有12个，三种颜色的球放一起。现从中随机取出10个球，且必须有白球。编程实现：输出所有可能的取球方案，并统计总共有多少种方案。**

*#使用函数封装***def** num\_search():  
 count = 0  
 **for** i **in** range(1,5):  
 **for** j **in** range(9):  
 **for** k **in** range(10):  
 **if** i + j + k == 10:  
 print(i,j,k)  
 count += 1  
 **return** count  
  
result = num\_search()  
print(**'总共有%d种取球方案。'**%result)  
  
*#使用闭包结构封装***def** out\_():  
 count = 0  
 **def** in\_():  
 **nonlocal** count  
 **for** i **in** range(1,5):  
 **for** j **in** range(8):  
 **for** k **in** range(10):  
 **if** i + j + k == 10:  
 print(i,j,k)  
 count += 1  
 **return** count  
 **return** in\_  
num\_search = out\_()  
result = num\_search()  
print(**'总共有%d种取球方案。'**%result)

1. **有1,2,3,4四个数字，请问能组成多少个互不相同且无重复数字的三位数？**

*# 基础算法实现***for** i **in** range(1,5):  
 **for** j **in** range(1,5):  
 **for** k **in** range(1,5):  
 **if** i != j **and** i != k **and** j != k:  
 print(i,j,k)  
  
*# 函数封装功能***def** is\_Num():  
 **for** i **in** range(1,5):  
 **for** j **in** range(1,5):  
 **for** k **in** range(1,5):  
 **if** i != j **and** i != k **and** j != k:  
 print(i,j,k)  
is\_Num()  
  
*# 改为闭包***def** out\_is():  
 it = range(1,5)  
 **def** in\_is():  
 **nonlocal** it  
 **for** i **in** it:  
 **for** j **in** it:  
 **for** k **in** it:  
 **if** i != j **and** i != k **and** j != k:  
 print(i,j,k)  
 **return** in\_is  
ok = out\_is()  
ok()

1. **输出1000以内的水仙花数**

*# 基础算法实现***for** i **in** range(100,1000):  
 **if** (i // 100) \*\* 3 + (i // 10 % 10) \*\* 3 + (i % 10) \*\* 3 == i:  
 print(i)  
*# 函数封装功能***def** flower():  
 **for** i **in** range(100,1000):  
 **if** (i // 100) \*\* 3 + (i // 10 % 10) \*\* 3 + (i % 10) \*\* 3 == i:  
 print(i)  
flower()  
*# 闭包封装功能***def** out\_():  
 it = range(100,1000)  
 **def** in\_():  
 **nonlocal** it  
 **for** i **in** it:  
 **if** (i // 100) \*\* 3 + (i // 10 % 10) \*\* 3 + (i % 10) \*\* 3 == i:  
 print(i)  
 **return** in\_  
flower = out\_()  
flower()  
*# 对象封装功能***class** Flower:  
 **def** \_\_init\_\_(self):  
 self.it = range(100,1000)  
 **def** flower(self):  
 **for** i **in** self.it:  
 **if** (i // 100) \*\* 3 + (i // 10 % 10) \*\* 3 + (i % 10) \*\* 3 == i:  
 print(i)  
flower = Flower()  
flower.flower()

1. **实现：分解质因数**

*# 基础算法实现*n = num = int(input(**"请输入一个数字："**))  
l = []  
**for** i **in** range(num // 2 + 1):  
 **for** j **in** range(2,n):  
 **if** n % j == 0:  
 l.append(j)  
 n //= j  
 **break  
if** len(l) == 0:  
 print(**'没有质因数！'**)  
**else**:  
 l.append(n)  
 print(**'%d = %d'**%(num,l[0]),end=**' '**)  
 **for** x **in** range(1,len(l)):  
 print(**'\*%d'**%l[x],end=**' '**)  
*# 函数封装功能***def** fenjie():  
 n = num = int(input(**"请输入一个数字："**))  
 l = []  
 **for** i **in** range(num // 2 + 1):  
 **for** j **in** range(2, n):  
 **if** n % j == 0:  
 l.append(j)  
 n //= j  
 **break  
 if** len(l) == 0:  
 print(**'没有质因数！'**)  
 **else**:  
 l.append(n)  
 print(**'%d = %d'** % (num, l[0]), end=**' '**)  
 **for** x **in** range(1, len(l)):  
 print(**'\*%d'** % l[x], end=**' '**)  
fenjie()  
*# 对象封装功能***class** Fenjie(object):  
 **def** \_\_init\_\_(self):  
 self.n = self.num = int(input(**"请输入一个数字："**))  
 self.l = []  
 **def** fenjie(self):  
 **for** i **in** range(self.num // 2 + 1):  
 **for** j **in** range(2, self.n):  
 **if** self.n % j == 0:  
 self.l.append(j)  
 self.n //= j  
 **break  
 if** len(self.l) == 0:  
 print(**'没有质因数！'**)  
 **else**:  
 self.l.append(self.n)  
 print(**'%d = %d'** % (self.num, self.l[0]), end=**' '**)  
 **for** x **in** range(1, len(self.l)):  
 print(**'\*%d'** % self.l[x], end=**' '**)

1. **随机获取一个字符串，统计其中：空格，制表符，数字和下划线的个数**

*# 基本功能实现*num\_tric = 0  
num\_num = 0  
num\_\_ = 0  
s\_str = input(**"请输入一串字符："**)  
**for** i **in** s\_str:  
 **if** i == **' '**:  
 num\_tric += 1  
 **try**:  
 is\_int = int(i)  
 **except** ValueError:  
 **pass  
 else**:  
 num\_num += 1  
 **if** i == **'\_'**:  
 num\_\_+=1  
  
print(**"空格数为：%d\n数字为：%d\n下划线数量为：%d\n"**%(num\_tric,num\_num,num\_\_))  
  
*# 函数封装功能***def** is\_str():  
 num\_tric = 0  
 num\_num = 0  
 num\_\_ = 0  
 s\_str = input(**"请输入一串字符："**)  
 **for** i **in** s\_str:  
 **if** i == **' '**:  
 num\_tric += 1  
 **try**:  
 is\_int = int(i)  
 **except** ValueError:  
 **pass  
 else**:  
 num\_num += 1  
 **if** i == **'\_'**:  
 num\_\_ += 1  
  
 print(**"空格数为：%d\n数字为：%d\n下划线数量为：%d\n"** % (num\_tric, num\_num, num\_\_))  
is\_str()  
  
*# 闭包封装功能***def** out\_():  
 num\_tric = 0  
 num\_num = 0  
 num\_\_ = 0  
 s\_str = input(**"请输入一串字符："**)  
 **def** in\_str():  
 **nonlocal** num\_num,num\_\_,s\_str,num\_tric  
 **for** i **in** s\_str:  
 **if** i == **' '**:  
 num\_tric += 1  
 **try**:  
 is\_int = int(i)  
 **except** ValueError:  
 **pass  
 else**:  
 num\_num += 1  
 **if** i == **'\_'**:  
 num\_\_ += 1  
 print(**"空格数为：%d\n数字为：%d\n下划线数量为：%d\n"** % (num\_tric, num\_num, num\_\_))  
 **return** is\_str  
is\_str = out\_()  
is\_str()  
  
*# 对象封装功能***class** Is\_str:  
 **def** \_\_init\_\_(self):  
 self.num\_tric = 0  
 self.num\_num = 0  
 self.num\_\_ = 0  
 self.s\_str = input(**"请输入一串字符："**)  
 **def** begin\_is(self):  
 **for** i **in** self.s\_str:  
 **if** i == **' '**:  
 self.num\_tric += 1  
 **try**:  
 self.is\_int = int(i)  
 **except** ValueError:  
 **pass  
 else**:  
 self.num\_num += 1  
 **if** i == **'\_'**:  
 self.num\_\_ += 1  
 print(**"空格数为：%d\n数字为：%d\n下划线数量为：%d\n"** % (self.num\_tric, self.num\_num, self.num\_\_))  
is\_str = Is\_str()  
is\_str.begin\_is()

1. **现有一个序列，这个序列由用户随机输入。问：当前序列中出现次数最多的字符是谁，并输出次数。**

*# 基础算法实现*l = []  
dic = {}  
**while True**:  
 num = input(**"请输入一个字符："**)  
 **if** num == **"over"**:  
 **break** l.append(num)  
**for** i **in** l:  
 **if** i **not in** dic:  
 dic[i] = 1  
 **else**:  
 dic[i] += 1  
print(dic)  
iMax = 0  
iIndex = **''  
for** key **in** dic:  
 **if** dic[key] > iMax:  
 iMax = dic[key]  
 iIndex = key  
print(**"出现次数最多的字符是：%s,总共出现了：%d次"**%(iIndex,iMax))  
*# 函数封装功能***def** is\_max():  
 l = []  
 dic = {}  
 **while True**:  
 num = input(**"请输入一个字符："**)  
 **if** num == **"0"**:  
 **break** l.append(num)  
 **for** i **in** l:  
 **if** i **not in** dic:  
 dic[i] = 0  
 **else**:  
 dic[i] += 1  
 *# print(dic)* iMax = 0  
 iIndex = **''  
 for** key **in** dic:  
 **if** dic[key] > iMax:  
 iMax = dic[key]  
 iIndex = key  
 print(**"出现次数最多的字符是：%s,总共出现了：%d次"**%(iIndex,iMax))  
  
is\_max()  
*# 闭包封装功能***def** out\_():  
 l = []  
 dic = {}  
 iMax = 0  
 iIndex = **''  
 def** in\_():  
 **nonlocal** l,dic,iMax,iIndex  
 **while True**:  
 num = input(**"请输入一个字符："**)  
 **if** num == **"0"**:  
 **break** l.append(num)  
 **for** i **in** l:  
 **if** i **not in** dic:  
 dic[i] = 0  
 **else**:  
 dic[i] += 1  
 print(dic)  
 **for** key **in** dic:  
 **if** dic[key] > iMax:  
 iMax = dic[key]  
 iIndex = key  
 **return** in\_  
is\_max = out\_()  
is\_max()  
*# 对象封装功能***class** Is\_max:  
 **def** \_\_init\_\_(self):  
 self.l = []  
 self.dic = {}  
 self.iMax = 0  
 self.iIndex = **''  
 def** is\_max(self):  
 **while True**:  
 num = input(**"请输入一个字符："**)  
 **if** num == **"0"**:  
 **break** self.l.append(num)  
 **for** i **in** self.l:  
 **if** i **not in** self.dic:  
 self.dic[i] = 0  
 **else**:  
 self.dic[i] += 1  
 print(self.dic)  
 **for** key **in** self.dic:  
 **if** self.dic[key] > self.iMax:  
 self.iMax = self.dic[key]  
 self.iIndex = key  
 print(**"出现次数最多的字符是：%s,总共出现了：%d次"** % (self.iIndex, self.iMax))  
is\_max = Is\_max()  
is\_max.is\_max()

1. **现有一个降序排列的列表，现用户输入一个数字，将其插入列表中形成一个新的降序列表**

*# 基础算法实现*l = []  
**while True**:  
 num = int(input(**"请输入一个字符："**))  
 **if** num == 0:  
 **break** l.append(num)  
print(l)  
**for** i **in** range(len(l)-1):  
 **for** j **in** range(len(l)-1-i):  
 **if** l[j] < l[j+1]:  
 l[j],l[j+1] = l[j+1],l[j]  
print(l)  
  
insert\_num = int(input(**"请输入您想插入的数字："**))  
**for** k **in** range(len(l)):  
 **if** insert\_num <= l[-1]:  
 l.append(insert\_num)  
 **break  
 elif** insert\_num >= l[0]:  
 l.insert(0,insert\_num)  
 **break  
 elif** insert\_num < l[k] **and** insert\_num > l[k+1]:  
 l.insert(k+1,insert\_num)  
 **break  
 elif** insert\_num == l[k]:  
 l.insert(k,insert\_num)  
 **break**print(l)  
*# 函数封装功能***def** insert\_num():  
 l = []  
 **while True**:  
 num = int(input(**"请输入一个字符："**))  
 **if** num == 0:  
 **break** l.append(num)  
 print(l)  
 **for** i **in** range(len(l) - 1):  
 **for** j **in** range(len(l) - 1 - i):  
 **if** l[j] < l[j + 1]:  
 l[j], l[j + 1] = l[j + 1], l[j]  
 print(l)  
  
 insert\_num = int(input(**"请输入您想插入的数字："**))  
 **for** k **in** range(len(l)):  
 **if** insert\_num <= l[-1]:  
 l.append(insert\_num)  
 **break  
 elif** insert\_num >= l[0]:  
 l.insert(0, insert\_num)  
 **break  
 elif** insert\_num < l[k] **and** insert\_num > l[k + 1]:  
 l.insert(k + 1, insert\_num)  
 **break  
 elif** insert\_num == l[k]:  
 l.insert(k, insert\_num)  
 **break** print(l)  
insert\_num()  
*# 闭包封装功能***def** out\_():  
 l = []  
 **def** in\_():  
 **nonlocal** l  
 **while True**:  
 num = int(input(**"请输入一个字符："**))  
 **if** num == 0:  
 **break** l.append(num)  
 print(l)  
 **for** i **in** range(len(l) - 1):  
 **for** j **in** range(len(l) - 1 - i):  
 **if** l[j] < l[j + 1]:  
 l[j], l[j + 1] = l[j + 1], l[j]  
 print(l)  
  
 insert\_num = int(input(**"请输入您想插入的数字："**))  
 **for** k **in** range(len(l)):  
 **if** insert\_num <= l[-1]:  
 l.append(insert\_num)  
 **break  
 elif** insert\_num >= l[0]:  
 l.insert(0, insert\_num)  
 **break  
 elif** insert\_num < l[k] **and** insert\_num > l[k + 1]:  
 l.insert(k + 1, insert\_num)  
 **break  
 elif** insert\_num == l[k]:  
 l.insert(k, insert\_num)  
 **break** print(l)  
 **return** in\_  
insert\_num = out\_()  
insert\_num()  
*# 对象封装功能（通过对象间的继承关系实现功能整合）***class** Make\_list:  
 **def** \_\_init\_\_(self):  
 self.l = []  
 **def** make\_list(self):  
 **while True**:  
 num = int(input(**"请输入一个字符："**))  
 **if** num == 0:  
 **break** self.l.append(num)  
 **return** self.l  
  
**class** Bubble\_sort:  
 **def** bubble\_sort(self,make\_over\_list):  
 **for** i **in** range(len(make\_over\_list) - 1):  
 **for** j **in** range(len(make\_over\_list) - 1 - i):  
 **if** make\_over\_list[j] < make\_over\_list[j + 1]:  
 make\_over\_list[j], make\_over\_list[j + 1] = make\_over\_list[j + 1], make\_over\_list[j]  
 **return** make\_over\_list  
**class** Insert\_number:  
 **def** insert\_number(self,bubble\_sort\_over\_list):  
 insert\_num = int(input(**"请输入您想插入的数字："**))  
 **for** k **in** range(len(bubble\_sort\_over\_list)):  
 **if** insert\_num <= bubble\_sort\_over\_list[-1]:  
 bubble\_sort\_over\_list.append(insert\_num)  
 **break  
 elif** insert\_num >= bubble\_sort\_over\_list[0]:  
 bubble\_sort\_over\_list.insert(0, insert\_num)  
 **break  
 elif** insert\_num < bubble\_sort\_over\_list[k] **and** insert\_num > bubble\_sort\_over\_list[k + 1]:  
 bubble\_sort\_over\_list.insert(k + 1, insert\_num)  
 **break  
 elif** insert\_num == bubble\_sort\_over\_list[k]:  
 bubble\_sort\_over\_list.insert(k, insert\_num)  
 **break  
 return** bubble\_sort\_over\_list  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 finally\_list = Make\_list().make\_list()  
 bubble\_list = Bubble\_sort().bubble\_sort(finally\_list)  
 insert\_list = Insert\_number().insert\_number(bubble\_list)  
 print(**"插入后的列表为：%r"**%insert\_list)

1. **有n个人围成一圈，顺序排号。从第一个人开始报数（1~3报数），凡报到3的人退出圈子，问最后留下的人原来排在第几号。**

num = int(input(**"请输入人数："**))  
k = 0*#计算报数  
#根据人数生成列表，列表包含num个数，代表有num个人，num的值代表每个人的编号*l = [x **for** x **in** range(1,num+1)]  
*# print(l)***while** len(l) > 1:  
 l\_copy = l[:]  
 **for** i **in** range(0,len(l\_copy)):  
 k += 1  
 **if** k % 3 == 0:  
 l.remove(l\_copy[i])  
print(**"最后留下来的是原来的第%d号兄弟"**%l[0])  
  
*#面向对象***class** Remove():  
 **def** \_\_init\_\_(self):  
 self.num = int(input(**"请输入你想要的人数："**))  
 self.count = 0  
  
 **def** make\_list(self):  
 self.L = [x **for** x **in** range(1,self.num+1)]  
 **return** self.L  
  
 **def** remove\_number(self):  
 self.make\_list()  
 **while** len(self.L) > 1:  
 self.L\_copy = self.L[:]  
 **for** i **in** range(0,len(self.L\_copy)):  
 self.count += 1  
 **if** self.count % 3 == 0:  
 self.L.remove(self.L\_copy[i])  
 **return** self.L  
  
rm = Remove()  
print(**"最后留下来的是原来的第%d号兄弟"**%rm.remove\_number()[0])

1. **猴子吃桃问题：有一堆桃子，猴子第一天吃了一半，还不过瘾，又吃了一个，第二天将剩余的桃子又吃掉一半，还不过瘾，又吃了一个，往后的每一天都是吃掉剩余桃子的一半多一个，待第十天再想吃的时候，发现只有一个桃子了。问原来共有多少桃子？**

*# 基础算法实现*x = 1  
**for** i **in** range(9,0,-1):  
 x = 2\*(x+1)  
print(**"原来有%d个桃子"**%x)  
  
*# 闭包封装基本功能***def** out\_():  
 x = 1  
 **def** in\_():  
 **nonlocal** x  
 **for** i **in** range(9,0,-1):  
 x = 2 \* (x + 1)  
 print(**"原来有%d个桃子"**%x)  
 **return** in\_  
monkey = out\_()  
monkey()  
*# 递归实现功能***def** monkey(x,day):  
 day -= 1  
 **if** day == 0:  
 **return** x  
 x = 2 \* (x + 1)  
 **return** monkey(x,day)  
print(**'原来有%d个桃子'**%monkey(1,10))  
*# 闭包实现递归功能***def** out\_():  
 x = 1  
 day = 10  
 **def** in\_():  
 **nonlocal** x, day  
 day -= 1  
 **if** day == 0:  
 **return** x  
 x = 2 \* (x + 1)  
 **return** in\_()  
 **return** in\_  
monkey = out\_()  
num = monkey()  
print(**'原来有%d个桃子'**%num)  
*# 对象实现功能***class** Monkey:  
 **def** \_\_init\_\_(self):  
 self.x = 1  
 self.day = 10  
 **def** monkey(self):  
 self.day -= 1  
 **if** self.day == 0:  
 **return** self.x  
 self.x = 2 \* (self.x + 1)  
 **return** self.monkey()  
monkey = Monkey()  
num = monkey.monkey()  
print(**'原来有%d个桃子'**%num)

1. **递归函数如何结束递归呢？如果不控制递归的深度会怎么样？递归和循环算法可以互相转化么？举个例子说明递归和循环算法互相转换。**

控制递归的循环次数可以结束递归，具体可以使用自然的结束递归方式(即不满足条件无法继续递归)或使用return语句返回一个结果结束递归。如果不控制递归，当递归达到一定深度时，会自动结束，但此时已经占据了很大的内存空间。递归和循环算法可以相互转化。比如实现一个斐波那契数列。

1. **何谓高阶函数？举几个例子？**

高阶函数：传入参数是一个函数，返回的还是一个函数的函数叫高阶函数。比如map(),filter(),sorted()

1. **最优化实现：查找一篇文章（一个文件），打开一个文件，输入关键词，如果匹配到5个关键词，说明整篇文章匹配到，打印5个关键词，如果匹配的关键词不到5个，打印关键词**

**import** re *#正则模块***import** os *#文件处理模块***import** string *#字符串*fres = open(**'C:/Users/Administrator/Desktop/res.txt'**, **'w'**)  
rootdir = **'C:/Users/Administrator/Desktop/log'  
for** parent, dirnames, filenames **in** os.walk(rootdir):  
 **for** filename **in** filenames:  
 *#print filename* fres.write(filename+**'\n'**) *#记录下文件名，便于代码纠错  
 # 提取hostname/sysname* f = open(**'C:/Users/Administrator/Desktop/log/'**+str(filename))  
 lines = f.readlines() *# .readlines() 自动将文件内容分析成一个行的列表，加快编译速度* **for** line **in** lines:  
 pattern = re.compile(**'(hostname|sysname).\*'**) *# s:space;()分组* match = pattern.match(line)  
 **if** match:  
 *# 使用Match获得分组信息* fres.write(match.group() + **'\n'**) *# Ctrl /可以注释多行* f.close()  
  
 *# 提取uptime，将格式转换为days* f1 = open(**'C:/Users/Administrator/Desktop/log/'**+str(filename)) *# 重新打开，不然无法成功写入* lines = f1.read()  
 t = re.findall(**'uptime.+'**, lines) *# findall进行正则匹配，不知为何compile不行了o.o  
 # .：非换行任意字符，+：1次或任意次数；\*：前一个字符0次或任意次* t\_convert = **''**.join(t) *# list转string* list = t\_convert.split(**' '**) *# 根据空格切片* y=0  
 w=0  
 d=0  
 **if** (**'year,' in** list): *# 提取年、周...转换类型计算* y = list[list.index(**'year,'**) - 1]  
 **if** (**'years,' in** list):  
 y = list[list.index(**'years,'**) - 1]  
 **if** (**'weeks,' in** list):  
 w = list[list.index(**'weeks,'**) - 1]  
 **if** (**'days,' in** list):  
 d = list[list.index(**'days,'**) - 1]  
 y = int(y)  
 w = int(w)  
 d = int(d)  
 days = 365 \* y + 7 \* w + d  
 fres.write(**'days: '** + str(days) + **'\n'**) *# 需进行类型转换* f1.close()  
  
 *# loopback要提取的地址在下一行* f2 = open(**'C:/Users/Administrator/Desktop/log/'**+str(filename))  
 lines = f2.read()  
 t = re.findall(**'interface Loopback.+\n ip address \S\*'**, lines)  
 t\_convert = **''**.join(t)  
 list = t\_convert.split(**' '**)  
 fres.write(**'loopback: '** + list[-1] + **'\n\n'**)  
 f2.close()  
fres.close()

1. **先用Python写出计算一个文件中大写字母数的代码。然后用one-liner 的风格简化该代码实现。（代码需要保证在文件大小远大于内存的条件下仍能正常工作）**

**import** os  
os.chdir(**r'C:\Users\Administrator\Desktop'**)*#os.chdir切换到指定目录***with** open(**'a.txt'**) **as** today:  
 count=0  
 **for** i **in** today.read():  
 **if** i.isupper():  
 count+=1  
print(count)

1. **请阐述yield、lambda、with关键字的作用**

生成器是通过一个或多个的yield语句构成的。

Lambda是Python中的匿名函数表达式

With是专门处理异常的语句

1. **磁盘上有两个有序数组，每个数组有10亿条数据，可以使用的内存只有10M，请把这2个有序列表合并成1个有序列表，给出代码过程。**

**def** sortTwoSortedList(list1=[],list2=[]):  
 *"""* **:param** *list1: 有序list,升序* **:param** *list2: 有序list，升序* **:return***: 合并后的有序list，升序  
 """* result=[]  
 *# while len(list1)>0 and len(list2) > 0:* **while** list1 **and** list2 :  
 **if** list1[0] >=list2[0]:  
 result.append(list2[0])  
 *# 顺便回顾一下python中list删除元素的三种方法：result.remove(list1[0])入参是值；  
 # result.pop(0)入参是索引会返回被删除的值，result.pop()删除最后一个元素，和result.append()结合，就是一个栈  
 #del(list1[0]) 直接删除某个list的某个元素* list2.pop(0)  
 **else**:  
 result.append(list1[0])  
 list1.pop(0)  
 **if** list1:  
 **for** i **in** list1:  
 result.append(i)  
 **if** list2:  
 **for** i **in** list2:  
 result.append(i)  
 **return** result  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 print(sortTwoSortedList(list1=[1,5,7,7,19,75],list2=[5,7,9,64,186,879]))