**Python语言的特点**

## 1. 简单易学、明确优雅、开发速度快

## 2. 跨平台、可移植、可扩展、交互式、解释型、面向对象的动态语言(在运行时可以改变其结构)

## 3. “内置电池”，大量的标准库和第三方库

## 4. 社区活跃，贡献者多，互帮互助

## 5. 开源语言，发展动力巨大

**GIL（Global Interpreter Lock）全局解释器锁**

## 这是一种防止多线程并发执行机器码的互斥锁，功能和性能之间权衡后的产物。Python在进行多线程任务的时候，其实是伪多线程，性能较差

**Pip常用命令**

## 卸载已安装的库 pip uninstall pillow

## 列出已经安装的库 pip list

## 将已经安装的库列表保存到文本文件

## pip freeze > requirements.txt

## 根据依赖文件批量安装

## pip install -r requirements.txt

**linux操作系统下的使用**

### 创建虚拟环境：

### $ virtualenv -p /usr/bin/python2.7 –-no-site-packages my\_venv

## -p：指定你要虚拟的Python版本，这里选择了本地的python2.7

## –-no-site-packages：表示在建立虚拟环境时不将原版本中的第三方库拷贝过来，这样就能获得一个纯净的Python环境。

### 启动虚拟环境： $ source my\_venv/bin/activate

### 退出虚拟环境： $ deactivate

### 删除虚拟环境： $ rm –r my\_venv

**编辑器/解释器**

## 编辑器：先整体编译在执行

## 解释器：边解释边执行

### **成员变量**

## 以单下划线开头的变量，例如\_foo代表禁止外部访问的类成员，需通过类提供的接口进行访问，不能用"from xxx import \*"导入；

## 而以双下划线开头的，例如\_\_foo，代表类的私有成员；以双下划线开头和结尾的（\_\_foo\_\_）是python里特殊方法专用的标识，如\_\_init\_\_（）代表类的构造函数。

## Python中，一切事物都是对象，变量引用的是对象或者说是对象在内存中的地址。

## 在Python中，变量本身没有数据类型的概念，通常所说的“变量类型”是变量所引用的对象的类型，或者说是变量的值的类型

### **基本运算**

## %取模（余数）：2%1==0

## \*\*幂运算：2\*\*3 == 8

## //取整除：9//2 == 4

## a = 0011 1100

## b = 0000 1101

## -----------------

## a&b = 0000 1100

## a|b = 0011 1101

## a^b = 0011 0001（异或运算，对位不同为1）

## ~a = 1100 0011

## is用于判断两个变量的引用是否为同一个对象（即为内存地址），而==用于判断变量引用的对象的值是否相等！

## input函数：获取用户输入，保存成一个字符串。

### **数据类型：**

## Python3的整型可以当作Long类型（更长的整型）使用，所以 Python3没有Python2的Long类型。

## Python的整数长度为32位，并且通常是连续分配内存空间的。（有一定的范围）

## >>> id(-2)

## 140705991811824

## >>> id(-1)

## 140705991811856

## >>> id(0)

## 140705991811888

## 从上面的空间地址看，地址之间正好差32。为什么会这样？因为Python在初始化环境的时候就在内存里自动划分了一块空间，专门用于整数对象的存取

## 小整数对象池：

## 这是一个包含262个指向整数对象的指针数组，范围是-5到256。也就是说比如整数10，即使我们在程序里没有创建它，其实在Python后台已经悄悄为我们创建了。

## >>> id(-6)

## 1544976157648

## >>> id(-5)

## 140705991811728

## >>> id(-4)

## 140705991811760

## >>> id(-3)

## 140705991811792

## >>> id(256)

## 140705991820080

## >>> id(257)

## 1544976155728

### **Python整数缓冲区的概念：**

## 也就是刚被删除的整数，不会被真正立刻删除回收，而是在后台缓冲一段时间，等待下一次的可能调用。

## a = 1000000 print(id(a)) del a b = 1000000 print(id(b))

## 结果：

## 1905736246416

## 1905736246416

## complex(x) ：将x转换到一个复数，实数部分为 x，虚数部分为 0。

## complex(x, y)： 将 x 和 y 转换到一个复数，实数部分为 x，虚数部分为 y。

## 布尔值：

## 0、0.0、-0.0、空字符串、空列表、空元组、空字典，这些都被判定为False。而-1、"False"也被判断为True。

## 空值None:

## None不能理解为0，因为0是整数类型，None也不是布尔类型，而是NoneType。

### **列表：**

## Python的列表是一个有序可重复的元素集合，可嵌套、迭代、修改、分片、追加、删除，成员判断。

## 从数据结构角度看，Python的列表是一个可变长度的顺序存储结构，每一个位置存放的都是对象的指针

## 如：alist = [1, “a”, [11,22], {“k1”:”v1”}]

## list(seq) 将序列转换为列表

## >>> s = list((1, "a", "b", 2))

## >>> s

## [1, 'a', 'b', 2]

### **切片**

## 语法：list[start:end]

## 区间是左闭右开的！也就是说[1:4]会截取列表的索引为1/2/3的3个元素，不会截取索引为4的元素。分片不会修改原有的列表，可以将结果保存到新的变量，因此切片也是一种安全操作，常被用来复制一个列表，例如newlist = lis[:]。

### **列表的内置方法：**

## append(obj) 在列表末尾添加新的对象

## count(obj) 统计某个元素在列表中出现的次数

## extend(seq) 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）

## index(obj) 从列表中找出某个值第一个匹配项的索引位置

## insert(index, obj) 将对象插入列表

## pop(obj=list[-1]) 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值

## remove(obj) 移除列表中某个值的第一个匹配项

## reverse() 反向列表中元素

## sort([func]) 对原列表进行排序

## copy() 复制列表

## clear() 清空列表，等于del lis[:]

## 注意：其中的类似 append，insert, remove 等方法会修改列表本身，并且没有返回值（严格的说是返回None）。

### **元组:**

## 元组也是序列结构，但是是一种不可变序列，你可以简单的理解为内容不可变的列表。除了在内部元素不可修改的区别外，元组和列表的用法差不多。

## >>> tup1 = () # 创建空元组

## >>> tup1 = (50,) # 创建只包含一个元素的元组时，要在元素的后面跟个逗号

## >>> tup1 = ('physics', 'chemistry', 1997, 2000)

## >>> tup2 = (1, 2, 3, 4, 5 )

## >>> tup3 = "a", "b", "c", "d"

## >>> tup = (1, 2, 3, 4)

## 元组只保证它的一级子元素不可变，对于嵌套的元素内部，不保证不可变！

## >>> tup = ('a', 'b', ['A', 'B'])

## >>> tup[2][0] = 'X'

## >>> tup[2][1] = 'Y'

## >>> tup

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

### **字符串:**

## 字符串是不可变的序列数据类型，不能直接修改字符串本身，和数字类型一样！

## 虽然字符串本身不可变，但可以像列表序列一样，通过方括号加下标的方式，访问或者获取它的子串，当然也包括切片操作,这一切都不会修改字符串本身.

## 方法：

### 1.string.**encode**(encoding='UTF-8', errors='strict') # 编码成bytes类型

## 以 encoding 指定的编码格式编码 string，编码的结果是一个bytes对象。如果出错默认报一个ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'

### 2.string**.find**(str, beg=0, end=len(string)) # 查找子串

## 检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1

### 3.string.index(str, beg=0, end=len(string)) # 获取下标

## 跟find()方法一样，只不过如果str不在 string中会报一个异常.

### 4.string.replace(str1, str2, num=string.count(str1)) # 替换子串

## 把string 中的 str1 替换成 str2,如果 num 指定，则替换不超过 num 次.

## 5.len(string) # 返回字符串长度，Python内置方法，非字符串方法。

## 6.lower() # 小写字符

## 7.upper() # 大写字符

## 8.string.split(str="", num=string.count(str))) # 分割字符串

### 以str为分隔符切片 string，如果 num有指定值，则仅分隔num个子字符串

## >>> a

## 'aaabbbcccaaabbbcccaaa'

## >>> a.split("a")

## ['', '', '', 'bbbccc', '', '', 'bbbccc', '', '', '']

## >>> a.split("a", 3)

## ['', '', '', 'bbbcccaaabbbcccaaa']

## 9.strip() # 去除两端的指定符号

## 10.string.startswith(obj, beg=0,end=len(string)) # 字符串是否以xxx开头

## 检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查.

## 11.string.endswith(obj, beg=0, end=len(string)) # 字符串是否以xxx结尾

## 检查字符串是否以 obj 结束，如果beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False.

## 简单的format格式化方法基本有两类：

## 1.{0}、{1}、{2}:这一类是位置参数，引用必须按顺序，不能随意调整，否则就乱了。例如：

## tpl = "i am {0}, age {1}, really {0}".format("seven", 18)

## 2.{name}、{age}、{gender}：这一类是关键字参数，引用时必须以键值对的方式，可以随意调整顺序。例如：

## tpl = "i am {name}, age {age}, really {name}".format(name="seven", age=18)

## tpl = "i am {name}, age {age}, really {name}".format(\*\*{"name": "seven", "age": 18})

### **字符串颜色控制（终端显示）**

## 格式：\033[显示方式;前景色;背景色m正文\033[0m

## \033[1;31;40m

## \033[0m 采用终端默认设置，也就是取消颜色设置

### **字符编码**

## ASCII编码：只有255个字符，每个字符需要8位也就是1个字节。不兼容汉字

## Unicode编码：又称万国码，国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。用2个字节来表示汉字。

## UTF-8编码：为了节省字节数，在Unicode的基础上进行优化的编码。用1个字节表示英文字符，3个字符表示汉字。天生兼容ASCII编码，所以最为流行

## Python3在运行时全部使用Unicode编码!

## 另外有这么几条规则，你要记住：

## 操作系统运行时，在内存中，统一使用的都是Unicode编码，当需要将数据保存到硬盘或者网络传输的时候，就转换为UTF-8编码，进行保存和传输。

## 用文本编辑器的时候，从文件系统或者说硬盘上读取的UTF-8编码字符被转换为Unicode字符到内存里，供程序或者操作系统使用。编辑完成后，保存的时候再把Unicode转换为UTF-8保存到文件。

## 浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8传输到客户的浏览器

**字典**：

## 字典可精确描述为不定长、可变、散列的集合类型。字典元素在内存中的存储方式是不连续的，也没有链接关系

## Python的字典数据类型是基于hash散列算法实现的，采用键值对(key:value)的形式，根据key的值计算value的地址，具有非常快的查取和插入速度。

## 字典包含的元素个数不限，值的类型可以是任何数据类型！但是字典的key必须是不可变的对象，例如整数、字符串、bytes和元组，最常见的还是将字符串作为key。列表、字典、集合等就不可以作为key。同时，同一个字典内的key必须是唯一的，但值则不必

## 注意：从Python3.6开始，字典是有序的！它将保持元素插入时的先后顺序！请务必清楚！

## Dict.pop() 必须提供指定的键值，否则报错

## Dict.popitem() 删除并返回字典的最后一个键值对，不接受参数。

## 遍历字典：

## # 1 直接遍历字典获取键，根据键取值

## for key in dic:

## print(key, dic[key])

## # 2 利用items方法获取键值，速度很慢，少用！

## for key,value in dic.items():

## print(key,value)

## #3 利用keys方法获取键

## for key in dic.keys():

## print(key, dic[key])

## #4 利用values方法获取值，但无法获取对应的键。

## for value in dic.values():

## print(value)

### **Bytes：**

## string = b'xxxxxx'.decode() 直接以默认的utf-8编码解码bytes成string

## b = string.encode() 直接以默认的utf-8编码string为bytes

**Set集合**

## >>> s = set([1,1,2,3,3,4])

## >>> s

## {1, 2, 3, 4} # 自动去重

## >>> set("it is a nice day") # 对于字符串，集合会把它一个一个拆开，然后去重

## {'s', 'e', 'y', 't', 'c', 'n', ' ', 'd', 'i', 'a'}

## >>> # 以下演示了两个集合的交、并、差操作

## >>> a = set('abracadabra')

## >>> b = set('alacazam')

## >>> a # a 中唯一的字母

## {'a', 'r', 'b', 'c', 'd'}

## >>> a - b # 在 a 中的字母，但不在 b 中

## {'r', 'd', 'b'}

## >>> a | b # 在 a 或 b 中的字母

## {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

## >>> a & b # 在 a 和 b 中都有的字母

## {'a', 'c'}

## >>> a ^ b # 在 a 或 b 中的字母，但不同时在 a 和 b 中

## {'r', 'd', 'b', 'm', 'z', 'l'}

### **伪代码**

## 伪代码(Pseudocode)是一种算法描述语言

**函数**

## return可以返回什么？

## 什么都不返回，仅仅return：return

## 数字/字符串/任意数据类型： return 'hello'

## 一个表达式：return 1+2

## 一个判断语句：return 100 > 99

## 一个变量：return a

## 一个函数调用：return func()

## 甚至是返回自己！：return self

## 多个返回值，以逗号分隔：return a, 1+2, "hello"

## 简而言之，函数可以return几乎任意Python对象。

### **动态参数：**

## \*args

## 1个星号表示接收任意个参数。调用时，会将实际参数打包成一个元组传入形式参数。如果参数是个列表，会将整个列表当做一个参数传入。例如

def func(\*args):

for arg in args:

print(arg)

func('a', 'b', 'c')

li = [1, 2, 3]func(li)

a

b

c

[1, 2, 3]

def func(\*args):

for arg in args:

print(arg)

li = [1, 2, 3]func(\*li)

1

2

3

1. \*\*kwargs

def func(\*\*kwargs):

for kwg in kwargs:

print(kwg, kwargs[kwg])

dic = {

'k1': 'v1',

'k2': 'v2'}

func(dic)

## 实际结果却是弹出错误，为什么？

## 因为这时候，我们其实是把dic当做一个位置参数传递给了func函数。而func函数并不接收任何位置函数。那怎么办呢？使用两个星号！

func(\*\*dic)

1. 关键字参数

## 关键字参数前面需要一个特殊分隔符\*和位置参数及默认参数分隔开来，\*后面的参数被视为关键字参数。在函数调用时，关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错。

def student(name, age, \*, sex):

pass

student(name="jack", age=18, sex='male')

## 如果函数定义中已经有了一个\*args参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符\*了。

def student(name, age=10, \*args, sex, classroom, \*\*kwargs):

pass

student(name="jack", age=18, sex='male', classroom="202", k1="v1")

## 变量作用域：

name ='jack'

def f1():

print(name)

def f2():

name = 'eric'

f1()

f2()

Jack

## ****Python函数的作用域取决于其函数代码块在整体代码中的位置，而不是调用时机的位置****。调用f1的时候，会去f1函数的定义体查找，对于f1函数，它的外部是name ='jack'，而不是name = 'eric'。

name = 'jack'

def f2():

name = 'eric'

return f1

def f1():

print(name)

ret = f2()

ret()

jack

## Range()

## range函数是Python使用频率非常高的一个内置函数，它除了以上的使用特点外，还具有类似生成器的特性，这里先看个例子：

for i in range(1000000000000000000000000000000000):

print(i)

if i > 10:

break

**注意：range不是生成器也不是迭代器，是本身独有的range类**

## 匿名函数：

**lambda 参数: 表达式**。

## 匿名函数只能有一个表达式，不用也不能写return语句，表达式的结果就是其返回值。

如 :lambda x: x \* x。它相当于下面的函数：

def f(x):

return x \* x

## 列表推导式

lis = [x \* x for x in range(1, 10)]

## 增加条件语句

>>> [x \* x for x in range(1, 11) if x % 2 == 0]

[4, 16, 36, 64, 100]

## 多重循环

>>> [a + b for a in ‘123' for b in ‘abc']

['1a', '1b', '1c', '2a', '2b', '2c', '3a', '3b', '3c']

## 字典推导式

>>> dic = {x: x\*\*2 for x in (2, 4, 6)}

>>> dic

{2: 4, 4: 16, 6: 36}

>>> type(dic)

<class 'dict'>

## 集合推导式

## 大括号除了能用作字典推导式，还可以用作集合推导式，两者仅仅在细微处有差别。

>>> a = {x for x in 'abracadabra' if x not in 'abc'}

>>> a

{'d', 'r'}

>>> type(a)

<class 'set'>

## 无元组推导式

## 要通过类似方法生成元组，需要显式调用元组的类型转换函数tuple()，如下所示：

tup = (x for x in range(9))

print(tup)

print(type(tup))

结果：

<generator object <genexpr> at 0x000000000255DA98>

<class 'generator'>

tup = tuple(x for x in range(9))

print(tup)

print(type(tup))

结果：

(0, 1, 2, 3, 4, 5, 6, 7, 8)

<class 'tuple'>

## 迭代器

## 迭代：通过for循环遍历对象的每一个元素的过程。

## 在Python中，list/tuple/string/dict/set/bytes都是可以迭代的数据类型。

## 可以通过collections模块的Iterable类型来判断一个对象是否可迭代：

>>> from collections import Iterable

>>> isinstance('abc', Iterable) # str是否可迭代

True

>>> isinstance([1,2,3], Iterable) # list是否可迭代

True

>>> isinstance(123, Iterable) # 整数是否可迭代

False

## 迭代器是一种可以被遍历的对象，并且能作用于next()函数。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往后遍历不能回溯，不像列表，你随时可以取后面的数据，也可以返回头取前面的数据。迭代器通常要实现两个基本的方法：iter() 和 next()。

## 很多时候，为了让我们自己写的类成为一个迭代器，需要在类里实现\_\_iter\_\_()和\_\_next\_\_()方法。

## 总结：Python的迭代器表示的是一个元素流，可以被next()函数调用并不断返回下一个元素，直到没有元素时抛出StopIteration错误。可以把这个元素流看做是一个有序序列，但却不能提前知道序列的长度，只能不断通过next()函数得到下一个元素，所以迭代器可以节省内存和空间。

## 迭代器(Iterator)和可迭代(Iterable)的区别：

## 凡是可作用于for循环的对象都是可迭代类型；

## 凡是可作用于next()函数的对象都是迭代器类型；

## ist、dict、str等是可迭代的但不是迭代器，因为next()函数无法调用它们。可以通过iter()函数将它们转换成迭代器。

## Python的for循环本质上就是通过不断调用next()函数实现的。

## 首先，可迭代（iterable）对象：字符串、列表、元组、集合、字典、生成器、都可以放到 for 循环里被加以处理，所以被称为可迭代对象。

## 其次，不是所有的可迭代对象都可以作为迭代器，字符串、列表、元组、集合、字典、生成器、这些可迭代对象，只有生成器可以被直接拿来当做迭代器。其他的几种数据类型，由于不是‘’数据流‘’，所以不能直接当做迭代器，需要用iter()函数加以处理之后才能当成迭代器。即：迭代器一定是可迭代对象，但是可迭代对象不一定是迭代器。

## 迭代器可以被next函数加以处理&调用，不能被其加以处理的，不算是迭代器

## 生成器

## 如果元素可以按照某种算法推算出来，需要就计算到哪个，就可以在循环的过程中不断推算出后续的元素，而不必创建完整的元素集合，从而节省大量的空间。在Python中，这种一边循环一边计算出元素的机制，称为生成器：generator。

## 通过圆括号可以编写生成器推导式：

>>> g = (x \* x for x in range(1, 4))

>>> g

<generator object <genexpr> at 0x1022ef630>

## 可以通过next()函数获得generator的下一个返回值，这点和迭代器非常相似：但更多情况下，我们使用for循环。

## 除了使用生成器推导式，我们还可以使用yield关键字。

## 在 Python中，使用yield返回的函数会变成一个生成器（generator）。 在调用生成器的过程中，每次遇到yield时函数会暂停并保存当前所有的运行信息，返回yield的值。并在下一次执行next()方法时从当前位置继续运行。

## 装饰器

## 装饰器（Decorator）：从字面上理解，就是装饰对象的器件。可以在不修改原有代码的情况下，为被装饰的对象增加新的功能或者附加限制条件或者帮助输出。装饰器有很多种，有函数的装饰器，也有类的装饰器。装饰器在很多语言中的名字也不尽相同，它体现的是设计模式中的装饰模式，强调的是开放封闭原则。装饰器的语法是将@装饰器名，放在被装饰对象上面。

def outer(func):

def inner():

print("认证成功！")

result = func()

print("日志添加成功")

return result

return inner

@outerdef f1():

print("业务部门1数据接口......")

## 内置函数

## hash()

## 为不可变对象，例如字符串生成哈希值的函数！

## isinstance()

## 判断一个对象是否是某个类的实例。比type()方法适用面更广

>>> isinstance("haha", str)

True

>>> isinstance(1, str)

False

## issubclass()

## issubclass(a，b),判断a是否是b的子类。

>>> class Foo:

pass

>>> class Goo(Foo):

pass

>>> issubclass(Goo, Foo)

True

## iter()

## 制造一个迭代器，使其具备next()能力。

## next()

## 通过调用迭代器的\_\_next\_\_()方法，获取下一个元素

## reversed() 反转，逆序对象

>>> reversed # reversed本身是个类

<class 'reversed'>

>>> reversed([1,2,3,4,5]) # 获得一个列表反转器

<list\_reverseiterator object at 0x0000022E322B5128>

>>> a = reversed([1,2,3,4,5])

>>> a

<list\_reverseiterator object at 0x0000022E32359668>

>>> list(a) # 使用list方法将它转换为一个列表

[5, 4, 3, 2, 1]

## slice()

## 返回一个切片类型的对象。slice是一个类，一种Python的数据类型。Python将对列表等序列数据类型的切片功能单独拿出来设计了一个slice类，可在某些场合下使用。

>>> s = slice(1, 10, 2)

>>> s

slice(1, 10, 2)

>>> type(s)

<class 'slice'>

>>> lis = [i for i in range(10)]

>>> lis

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> lis[s] # 注意用法

[1, 3, 5, 7, 9]

## sum() 里面要是可迭代对象

## map()

## 映射函数。使用指定的函数，处理可迭代对象，并将结果保存在一个map对象中，本质上和大数据的mapreduce中的map差不多。

## 使用格式：obj = map(func, iterable),func是某个函数名，iterable是一个可迭代对象。

li = [1,2,3]

data = map(lambda x :x\*100,li) # 这里直接使用了一个匿名函数

print(type(data)) # 返回值是一个map对象，它是个迭代器。

data = list(data) # 可以用list方法将map对象中的元素全部生成出来，保存到一个列表里。

print(data)

运行结果：

<class 'map'>

[100, 200, 300]

## filter()

## 过滤器，用法和map类似。在函数中设定过滤的条件，逐一循环对象中的元素，将返回值为True时的元素留下（注意，不是留下返回值！），形成一个filter类型的迭代器。

def f1(x):

if x > 3:

return True

else:

return Falsel

i = [1,2,3,4,5]

data = filter(f1,li)

print(type(data))

print(list(data))

----------------------------运行结果：

<class 'filter'>

[4, 5]

## zip()

## 组合对象。将对象逐一配对。

list\_1 = [1,2,3]

list\_2 = ['a','b','c']

s = zip(list\_1,list\_2)

print(list(s))

--------------------------------运行结果：

[(1, 'a'), (2, 'b'), (3, 'c')]

## 那么如果对象的长度不一致呢？多余的会被抛弃！以最短的为基础！

## sorted()

## 排序方法。有key和reverse两个重要参数。

## 基础用法: 直接对序列进行排序

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

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

## 指定排序的关键字。关键字必须是一个可调用的对象。例如下面的例子，规则是谁的绝对值大，谁就排在后面。

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

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

## 指定按反序排列。下面的例子，首先按忽略大小写的字母顺序排序，然后倒序排列。

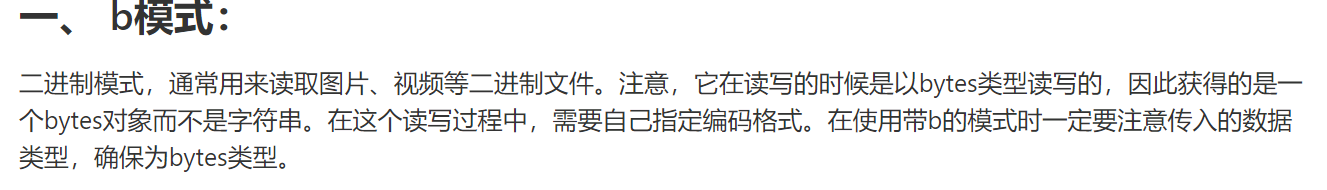
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower, reverse=True)

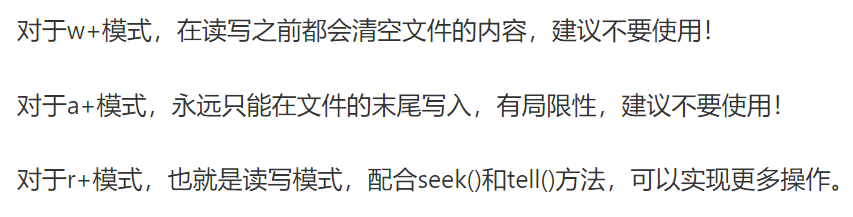
['Zoo', 'Credit', 'bob', 'about']

## 文件管理

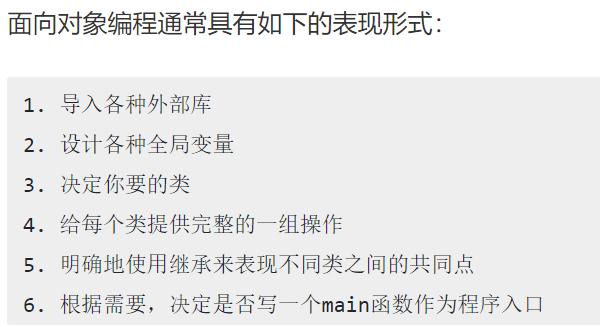
f = open(filename, mode)

## PS：Python中，所有具有read和write方法的对象，都可以归类为file类型。而所有的file类型对象都可以使用open方法打开，close方法结束和被with上下文管理器管理。这是Python的设计哲学之一。





## 面向对象编程：



## 

## 类的方法：

## Python的类中包含实例方法、静态方法和类方法三种方法。这些方法无论是在代码编排中还是内存中都归属于类，区别在于传入的参数和调用方式不同。在类的内部，使用def关键字来定义一个方法。

## 实例方法

## 类的实例方法由实例调用，至少包含一个self参数，且为第一个参数。执行实例方法时，会自动将调用该方法的实例赋值给self。self代表的是类的实例，而非类本身。self不是关键字，而是Python约定成俗的命名，你完全可以取别的名字，但不建议这么做

## 静态方法

## 静态方法由类调用，无默认参数。将实例方法参数中的self去掉，然后在方法定义上方加上@staticmethod，就成为静态方法。它属于类，和实例无关。建议只使用类名.静态方法的调用方式。（虽然也可以使用实例名.静态方法的方式调用）

## **类方法**

## 类方法由类调用，采用@classmethod装饰，至少传入一个cls（代指类本身，类似self）参数。执行类方法时，自动将调用该方法的类赋值给cls。建议只使用类名.类方法的调用方式。（虽然也可以使用实例名.类方法的方式调用）

## 面向对象编程有三大重要特征：封装、继承和多态。

## **封装**

## 封装是指将数据与具体操作的实现代码放在某个对象内部，使这些代码的实现细节不被外界发现，外界只能通过接口使用该对象，而不能通过任何形式修改对象内部实现，正是由于封装机制，程序在使用某一对象时不需要关心该对象的数据结构细节及实现操作的方法。

## **继承**

## 继承机制实现了代码的复用，多个类公用的代码部分可以只在一个类中提供，而其他类只需要继承这个类即可。

## 

## 

## super()函数：

## 我们都知道，在子类中如果有与父类同名的成员，那就会覆盖掉父类里的成员。那如果你想强制调用父类的成员呢？使用super()函数！这是一个非常重要的函数，最常见的就是通过super调用父类的实例化方法\_\_init\_\_！

## 语法：super(子类名, self).方法名()，需要传入的是子类名和self，调用的是父类里的方法，按父类的方法需要传入参数。

## **多态**

## 动态语言调用实例方法时不检查类型，只要方法存在，参数正确，就可以调用

## 成员保护和访问限制

## 在Python中，如果要让内部成员不被外部访问，可以在成员的名字前加上两个下划线\_\_，这个成员就变成了一个私有成员（private）。私有成员只能在类的内部访问，外部无法访问。

## 那么，以双下划线开头的数据成员是不是一定就无法从外部访问呢？其实也不是！本质上，从内部机制原理讲，外部不能直接访问\_\_age是因为Python解释器对外把\_\_age变量改成了\_People\_\_age，也就是\_类名\_\_age（类名前是一个下划线）。因此，投机取巧的话，你可以通过\_ People\_\_age在类的外部访问\_\_age变量：

## 此外，有些时候，你会看到以一个下划线开头的成员名，比如\_name，这样的数据成员在外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的标识符时，意思就是，“虽然我可以被外部访问，但是，请把我视为私有成员，不要在外部访问我！”。

## **@property装饰器**

### **property()函数**

## 除了使用装饰器的方式将一个方法伪装成属性外，Python内置的builtins模块中的property()函数，为我们提供了第二种设置类属性的手段。

## Python内置的@property装饰器可以把类的方法伪装成属性调用的方式。也就是本来是Foo.func()的调用方法，变成Foo.func的方式。在很多场合下，这是一种非常有用的机制。

# 

## 特殊成员和魔法方法

## 1. \_\_doc\_\_

## 说明性文档和信息。Python自建，无需自定义。

## 7. \_\_str\_\_()

## 如果一个类中定义了\_\_str\_\_()方法，那么在打印对象时，默认输出该方法的返回值。这也是一个非常重要的方法，需要用户自己定义。

## 8、\_\_getitem\_\_()、\_\_setitem\_\_()、\_\_delitem\_\_()

## 取值、赋值、删除这“三剑客”的套路，在Python中，我们已经见过很多次了，比如前面的@property装饰器。

## Python中，标识符后面加圆括号，通常代表执行或调用方法的意思。而在标识符后面加中括号[]，通常代表取值的意思。Python设计了\_\_getitem\_\_()、\_\_setitem\_\_()、\_\_delitem\_\_()这三个特殊成员，用于执行与中括号有关的动作。它们分别表示取值、赋值、删除数据。

class Foo:

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

print('\_\_getitem\_\_',key)

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

print('\_\_setitem\_\_',key,value)

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

print('\_\_delitem\_\_',key)

obj = Foo()

result = obj['k1'] # 自动触发执行 \_\_getitem\_\_obj['k2'] = 'jack' # 自动触发执行 \_\_setitem\_\_del obj['k1'] # 自动触发执行 \_\_delitem\_\_

## 9. \_\_iter\_\_()

## 这是迭代器方法！列表、字典、元组之所以可以进行for循环，是因为其内部定义了 \_\_iter\_\_()这个方法。如果用户想让自定义的类的对象可以被迭代，那么就需要在类中定义这个方法，并且让该方法的返回值是一个可迭代的对象。当在代码中利用for循环遍历对象时，就会调用类的这个\_\_iter\_\_()方法。

## 1.返回一个个迭代对象：

class Foo:

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

self.sq = sq

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

return iter(self.sq)

obj = Foo([11,22,33,44])

for i in obj:

print(i)

# 这下没问题了！

## 2.最好的方法是使用生成器：

class Foo:

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

pass

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

yield 1

yield 2

yield 3

obj = Foo()for i in obj:

print(i)

## 14. \_\_slots\_\_

## Python作为一种动态语言，可以在类定义完成和实例化后，给类或者对象继续添加随意个数或者任意类型的变量或方法，这是动态语言的特性。

# **reflect反射**

## 标识名”和看起来相同的“字符串”。两者字面上看起来一样，却是两种东西，比如下面的func函数和字符串“func”：

# commons.py

def login():

print("这是一个登陆页面！")

def logout():

print("这是一个退出页面！")

def home():

print("这是网站主页面！")

def func():

print("func是这个函数的名字！")

s = "func"

print("%s是个字符串" % s)

import commons

def run():

inp = input("请输入您想访问页面的url： ").strip()

func = getattr(commons,inp)

func()

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

run()

## func = getattr(commons,inp)语句是关键，通过getattr()函数，从commons模块里，**查找到和inp字符串“外形”相同的函数名，并将其返回，然后赋值给func变量**。变量func此时就指向那个函数，func()就可以调用该函数。**getattr()函数的使用方法：接收2个参数，前面的是一个类或者模块，后面的是一个字符串，注意了！是个字符串！**

## 前面的代码还有个小瑕疵，那就是如果用户输入一个非法的url，比如j

## 那怎么办呢？python提供了一个hasattr()的内置函数，用法和getattr()基本类似，它可以判断commons中是否具有某个成员，返回True或False。

# visit.pyimport commons

def run():

inp = input("请输入您想访问页面的url： ").strip()

if hasattr(commons,inp):

func = getattr(commons,inp)

func()

else:

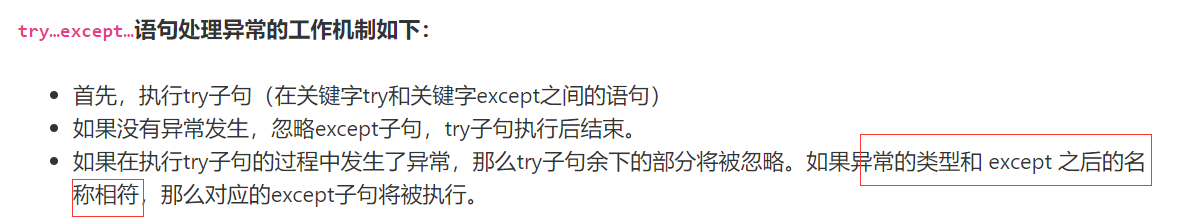
print("404")

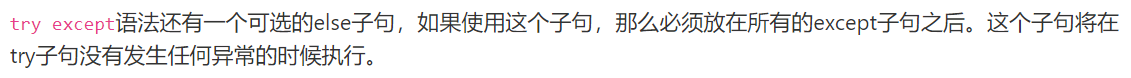
if \_\_name\_\_ == '\_\_main\_\_':

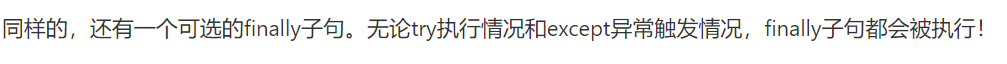
run()

## Python的四个重要内置函数：getattr()、hasattr()、delattr()和setattr()较为全面的实现了基于字符串的反射机制。

### **异常处理：**

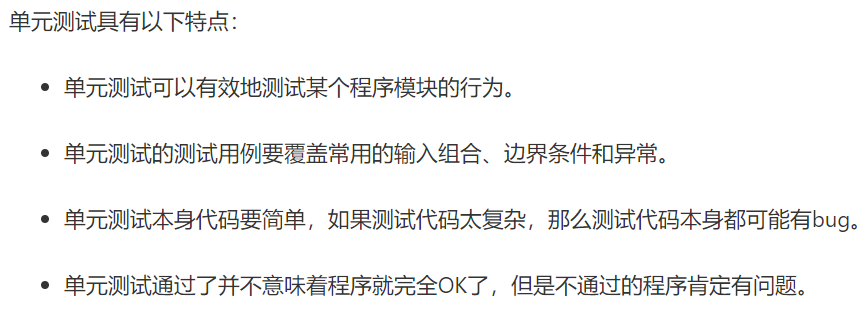


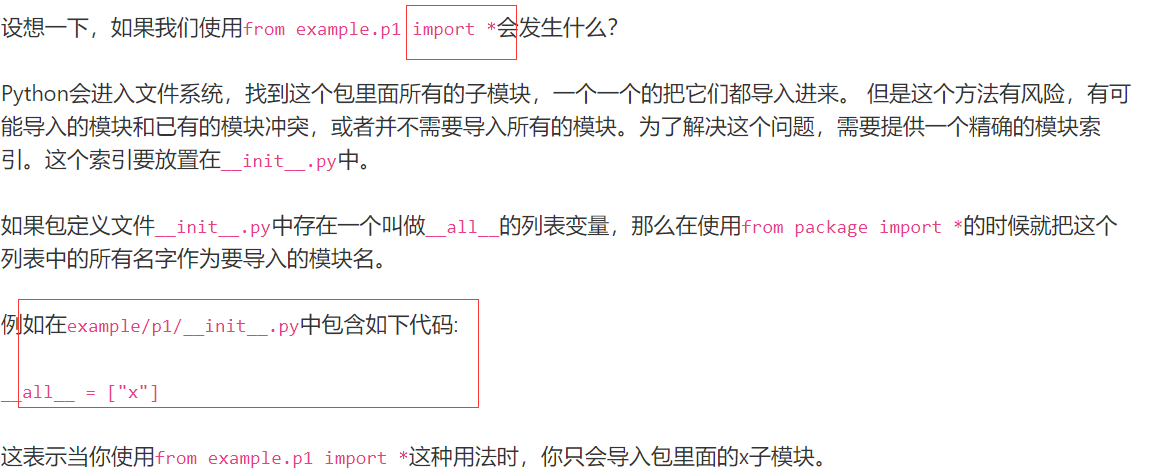




### **断言assert：**

### **单元测试之unittest：**



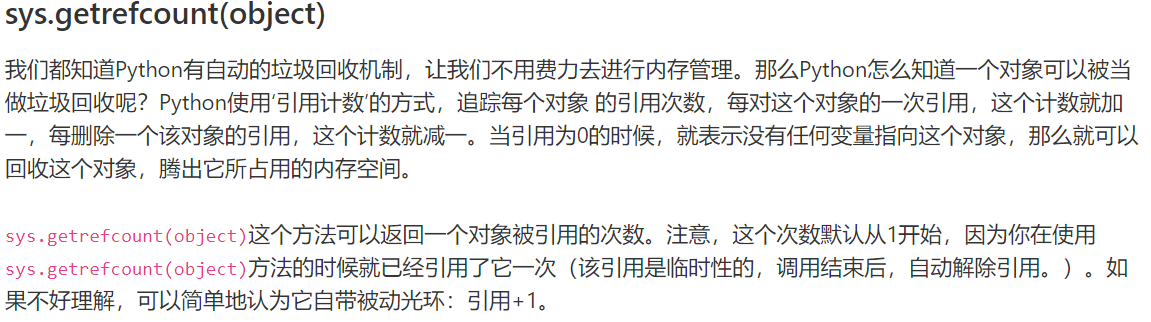


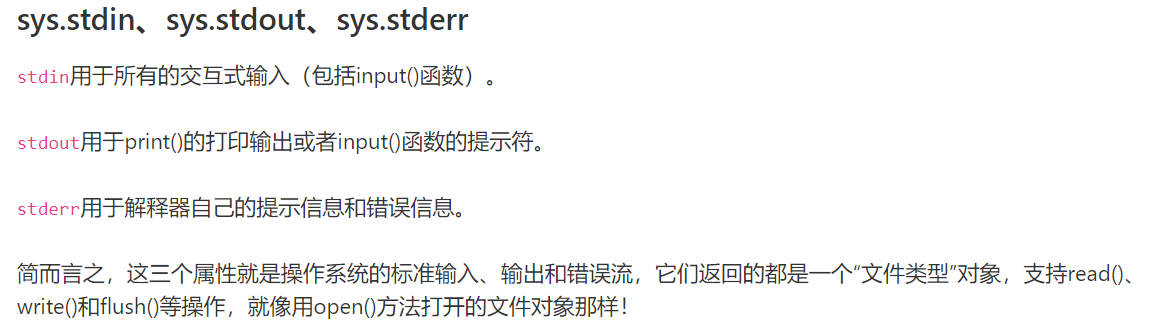
# **os：**

## os模块的主要功能：系统相关、目录及文件操作、执行命令和管理进程

# **sys：**

## sys模块主要是针对与Python解释器相关的变量和方法，不是主机操作系统。





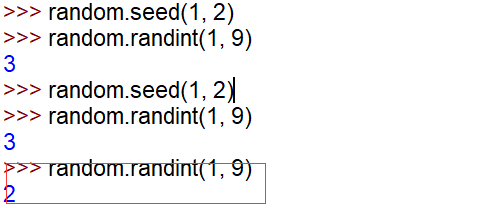
# **random：**

## **random模块用于生成伪随机数。**

### 

## **random.seed(a=None, version=2)**

## 初始化随机数生成器。如果未提供a或者a=None，则使用系统时间为种子。如果a是一个整数，则作为新的种子。Version = 2默认可以不写



## **random.randint(a, b)**

## 返回一个a <= N <= b的随机整数N。等同于randrange(a, b+1)

### **3. 针对序列类型的方法**

### **random.choice(seq)**

## 从非空序列seq中随机选取一个元素。如果seq为空则弹出IndexError异常。

**random.shuffle(x[, random])**

## 随机打乱序列x内元素的排列顺序，俗称“洗牌”。只能用于可变的序列，对于不可变序列，请使用下面的sample()方法。

**random.sample(population, k)**

## 从population样本或集合中随机抽取K个不重复的元素形成新的序列。常用于不重复的随机抽样。**返回的是一个新的序列，不会破坏原有序列。**比如从一个整数区间随机抽取一定数量的整数random.sample(range(10000000), k=60)，这非常有效和节省空间。 如果k大于population的长度，则弹出ValueError异常。

**random.random()**

返回一个介于左闭右开[0.0, 1.0)区间的浮点数。

>>> from random import \*

>>> random() # 随机浮点数: 0.0 <= x < 1.00.37444887175646646

>>> uniform(2.5, 10.0) # 随机浮点数: 2.5 <= x < 10.03.1800146073117523

>>> randrange(10) # 0-9的整数：7

>>> randrange(0, 101, 2) # 0-100的偶数26

>>> choice(['win', 'lose', 'draw']) # 从序列随机选择一个元素

'draw'

>>> deck = 'ace two three four'.split()

>>> shuffle(deck) # 对序列进行洗牌，改变原序列

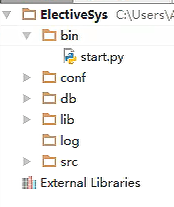
>>> deck

['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4) # 不改变原序列的抽取指定数目样本，并生成新序列

[40, 10, 50, 30]

### **软件开发规范：**



## bin：软件程序入口（一个脚本）

## conf: 配置文件（静态配置settings）

## src: 软件运行的主逻辑（里面实现了主要的方法）

## db：数据库

## log: 日志

## lib: 公共类库