

Python 黑魔法指南

+ Python 黑魔法指南

微信公众号@Python编程时光

作者：王炳明

版本：v1.0

发布时间：2020年05年12日

微信公众号：Python编程时光

联系邮箱：wongbingming@163.com

Github: <https://github.com/iswbm/magic-python>

版权归个人所有，欢迎交流分享，不允许用作商业及为个人谋利等用途，违者必究。

01. 默默无闻的省略号很好用

在Python中，一切皆对象，省略号也不例外。

在 Python 3 中你可以直接写 `...` 来得到它

```
>>> ...
Ellipsis
>>> type(...)
<class 'ellipsis'>
```

而在 Python 2 中没有 `...` 这个语法，只能直接写 `Ellipsis` 来获取。

```
>>> Ellipsis
Ellipsis
>>> type(Ellipsis)
<type 'ellipsis'>
>>>
```

它转为布尔值时为真

```
>>> bool(...)
True
```

最后，这东西是一个单例。

```
>>> id(...)
4362672336
>>> id(...)
4362672336
```

那这东西有啥用呢？

1. 它是 Numpy 的一个语法糖
2. 在 Python 3 中可以使用 `...` 代替 `pass`

```
$ cat demo.py
def func01():
    ...

def func02():
    pass

func01()
```

```
func02()

print("ok")

$ python3 demo.py
ok
```

02. 使用 end 来结束代码块

有不少编程语言，循环、判断代码块需要用 end 标明结束，这样一定程度上会使代码逻辑更加清晰一点。

但是其实在 Python 这种严格缩进的语言里并没有必要这样做。

如果你真的想用，也不是没有办法，具体你看下面这个例子。

```
__builtins__.end = None

def my_abs(x):
    if x > 0:
        return x
    else:
        return -x
    end
end

print(my_abs(10))
print(my_abs(-10))
```

执行后，输出如下

```
[root@localhost ~]$ python demo.py
10
10
```

03. 可直接运行的 zip 包

我们可以经常看到有 Python 包，居然可以以 zip 包进行发布，并且可以不用解压直接使用。

这与大多数人的认识的 Python 包格式不一样，正常人认为 Python 包的格式要嘛是 egg，要嘛是 whl 格式。

那么这个 zip 是如何制作的呢，请看下面的示例。

```
[root@localhost ~]# ls -l demo
total 8
-rw-r--r-- 1 root root 30 May  8 19:27 calc.py
-rw-r--r-- 1 root root 35 May  8 19:33 __main__.py
[root@localhost ~]#
[root@localhost ~]# cat demo/__main__.py
import calc

print(calc.add(2, 3))
[root@localhost ~]#
[root@localhost ~]# cat demo/calc.py
def add(x, y):
    return x+y
[root@localhost ~]#
[root@localhost ~]# python -m zipfile -c demo.zip demo/*
[root@localhost ~]#
```

制作完成后，我们可以执行用 python 去执行它

```
[root@localhost ~]# python demo.zip
5
[root@localhost ~]#
```

04. 反斜杠的倔强: 不写最后

\ 在 Python 中的用法主要有两种

1、在行尾时，用做续行符

```
[root@localhost ~]$ cat demo.py
print("hello "\
      "world")
```

```
[root@localhost ~]$  
[root@localhost ~]$ python demo.py  
hello world
```

2、在字符串中，用做转义字符，可以将普通字符转化为有特殊含义的字符。

```
>>> str1='\nhello'    # 换行  
>>> print(str1)  
  
hello  
>>> str2='\thello'    # tab  
>>> print(str2)  
    hello
```

但是如果你用单 `\` 结尾是会报语法错误的

```
>>> str3="\n"  
File "<stdin>", line 1  
    str3="\n"  
        ^  
SyntaxError: EOL while scanning string literal
```

就算你指定它是个 raw 字符串，也不行。

```
>>> str3=r"\n"  
File "<stdin>", line 1  
    str3=r"\n"  
        ^  
SyntaxError: EOL while scanning string literal
```

05. 单行实现 for 死循环如何写？

如果让你在不借助 while ，只使用 for 来写一个死循环？

你会写吗？

如果你还说简单，你可以自己试一下。

...

如果你尝试后，仍然写不出来，那我给出自己的做法。

```
for i in iter(int, 1):pass
```

是不是傻了？iter 还有这种用法？这为啥是个死循环？

关于这个问题，你如果看中文网站，可能找不到相关资料。

还好你可以通过 IDE 看py源码里的注释内容，介绍了很详细的使用方法。

原来iter有两种使用方法。

- 通常我们的认知是第一种，将一个列表转化为一个迭代器。
- 而第二种方法，他接收一个 callable对象，和一个sentinel 参数。第一个对象会一直运行，直到它返回 sentinel 值才结束。

那 `int` 呢？

这又是一个知识点，int 是一个内建方法。通过看注释，可以看出它是有默认值0的。你可以在 console 模式下输入 `int()` 看看是不是返回0。

由于int() 永远返回0，永远返回不了1，所以这个 for 循环会没有终点。一直运行下去。

作者：王炳明，微信公众号《Python编程时光》。版权归个人所有，欢迎分享。仅用于学习交流，但勿用作商业用途，违者必究。

06. 懒人必备技能：使用“_”

对于 `_`，大家对于他的印象都是用于 占位符，省得为一个不需要用到的变量，绞尽脑汁的想变量名。

今天要介绍的是他的第二种用法，就是在 console 模式下的应用。

示例如下：

```
>>> 3 + 4
7
>>> _
7
>>> name='公众号：Python编程时光'
>>> name
'公众号：Python编程时光'
>>> _
'公众号：Python编程时光'
```

它可以返回上一次的运行结果。

但是，如果是print函数打印出来的就不行了。

```
>>> 3 + 4
7
>>> _
7
>>> print("公众号：Python编程时光")
ming
>>> _
7
```

我自己写了个例子，验证了下，用 `__repr__` 输出的内容可以被获取到的。
首先，在我们的目录下，写一个文件 `demo.py`。内容如下

```
# demo.py
class mytest():
    def __str__(self):
        return "hello"

    def __repr__(self):
        return "world"
```

然后在这个目录下进入交互式环境。

```
>>> import demo
>>> mt=demo.mytest()
>>> mt
```

```
world
>>> print(mt)
hello
>>> _
world
```

知道这两个魔法方法的人，一看就明白了，这里不再解释啦。

07. 最快查看包搜索路径的方式

当你使用 `import` 导入一个包或模块时，Python 会去一些目录下查找，而这些目录是有优先级顺序的，正常人会使用 `sys.path` 查看。

```
>>> import sys
>>> from pprint import pprint
>>> pprint(sys.path)
['',
 '/usr/local/Python3.7/lib/python37.zip',
 '/usr/local/Python3.7/lib/python3.7',
 '/usr/local/Python3.7/lib/python3.7/lib-dynload',
 '/home/wangbm/.local/lib/python3.7/site-packages',
 '/usr/local/Python3.7/lib/python3.7/site-packages']
>>>
```

那有没有更快的方式呢？

我这有一种连 `console` 模式都不用进入的方法，一行命令即可解决

```
[wangbm@localhost ~]$ python3 -m site
sys.path = [
  '/home/wangbm',
  '/usr/local/Python3.7/lib/python37.zip',
  '/usr/local/Python3.7/lib/python3.7',
  '/usr/local/Python3.7/lib/python3.7/lib-dynload',
  '/home/wangbm/.local/lib/python3.7/site-packages',
  '/usr/local/Python3.7/lib/python3.7/site-packages',
]
USER_BASE: '/home/wangbm/.local' (exists)
USER_SITE: '/home/wangbm/.local/lib/python3.7/site-packages' (exists)
ENABLE_USER_SITE: True
```


从输出你可以发现，这个列的路径会比 `sys.path` 更全，它包含了用户环境的目录。

08. and 和 or 的取值顺序

`and` 和 `or` 是我们再熟悉不过的两个逻辑运算符，在 Python 也有它有妙用。

- 当一个 `or` 表达式中所有值都为真，Python 会选择第一个值
- 当一个 `and` 表达式 所有值都为真，Python 会选择第二个值。

示例如下：

```
>>>(2 or 3) * (5 and 7)
14 # 2*7
```

09. 如何修改解释器提示符

这个当做今天的一个小彩蛋吧。应该算是比较冷门的，估计知道的人很少了吧。

正常情况下，我们在 终端下 执行Python 命令是这样的。

```
>>> for i in range(2):
...     print (i)
...
0
1
```

你是否想过 `>>>` 和 `...` 这两个提示符也是可以修改的呢？

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'...'
>>>
>>> sys.ps2 = '----- '
>>> sys.ps1 = 'Python编程时光>>>'
Python编程时光>>>for i in range(2):
```

```
----- print (i)
-----
0
1
```

10. 逗号也有它独特的用法

逗号，虽然是个很不起眼的符号，但在 Python 中也有他的用武之地。

第一个用法

元组的转化

```
[root@localhost ~]# cat demo.py
def func():
    return "ok",

print(func())
[root@localhost ~]# python3 demo.py
('ok',)
```

第二个用法

print 的取消换行

```
[root@localhost ~]# cat demo.py
for i in range(3):
    print i
[root@localhost ~]#
[root@localhost ~]# python demo.py
0
1
2
[root@localhost ~]#
[root@localhost ~]# vim demo.py
[root@localhost ~]#
[root@localhost ~]# cat demo.py
for i in range(3):
    print i,
[root@localhost ~]#
```

```
[root@localhost ~]# python demo.py
0 1 2
[root@localhost ~]#
```

作者：王炳明，微信公众号《Python编程时光》。版权归个人所有，欢迎分享。仅用于学习交流，但勿用作商业用途，违者必究。

11. 默认参数最好不为可变对象

函数的参数分三种

- 可变参数
- 默认参数
- 关键字参数

当你在传递默认参数时，有新手很容易踩雷的一个坑。

先来看一个示例

```
def func(item, item_list=[]):
    item_list.append(item)
    print(item_list)

func('iphone')
func('xiaomi', item_list=['oppo', 'vivo'])
func('huawei')
```

在这里，你可以暂停一下，思考一下会输出什么？

思考过后，你的答案是否和下面的一致呢

```
['iphone']
['oppo', 'vivo', 'xiaomi']
['iphone', 'huawei']
```

如果是，那你可以跳过这部分内容，如果不是，请接着往下看，这里来分析一下。

Python 中的 def 语句在每次执行的时候都初始化一个函数对象，这个函数对象就是我们要调用的函数，可以把它当成一个一般的对象，只不过这个对象拥有一个可执行的方法和部分属性。

对于参数中提供了初始值的参数，由于 Python 中的函数参数传递的是对象，也可以认为是传地址，在第一次初始化 def 的时候，会先生成这个可变对象的内存地址，然后将这个默认参数 item_list 会与这个内存地址绑定。在后面的函数调用中，如果调用方指定了新的默认值，就会将原来的默认值覆盖。如果调用方没有指定新的默认值，那就会使用原来的默认值。

第一次调用时，会执行初始化，生成 [] 的内存地址是：2830870084744
第二次调用时，指定了新的默认对象（2830874211912），将原来的覆盖，就像“压栈”一样，在函数结束后，会将这个“栈”弹出。
第三次调用时，item_list 的默认参数还是指向 2830870084744（因为上一次调用已经将新对象的引用弹出了）



12. 访问类中的私有方法

大家都知道，类中可供直接调用的方法，只有公有方法（protected类型的方法也可以，但是不建议）。也就是说，类的私有方法是无法直接调用的。

这里先看一下例子

```
class Kls():
    def public(self):
        print('Hello public world!')

    def __private(self):
        print('Hello private world!')

    def call_private(self):
        self.__private()

ins = Kls()

# 调用公有方法，没问题
ins.public()
```

```
# 直接调用私有方法，不行
ins.__private()

# 但你可以通过内部公有方法，进行代理
ins.call_private()
```

既然都是方法，那我们真的没有方法可以直接调用吗？

当然有啦，只是建议你千万不要这样弄，这里只是普及，让你了解一下。

```
# 调用私有方法，以下两种等价
ins._Kls__private()
ins.call_private()
```

13. 时有时无的切片异常

这是个简单例子，alist 只有5 个元素，当你取第 6 个元素时，会抛出索引异常。这与我们的认知一致。

```
>>> alist = [0, 1, 2, 3, 4]
>>> alist[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

但是当你使用 alist[5:] 取一个区间时，即使 alist 并没有 第 6个元素，也不抛出异常，而是会返回一个新的列表。

```
>>> alist = [0, 1, 2, 3, 4]
>>> alist[5:]
[]
>>> alist[100:]
[]
```

14. 哪些情况下不需要续行符？

在写代码时，为了代码的可读性，代码的排版是尤为重要的。

为了实现高可读性的代码，我们常常使用到的就是续行符 `\`。

```
>>> a = 'talk is cheap,\n...      'show me the code.'\n>>>\n>>> print(a)\ntalk is cheap,show me the code.
```

那有哪些情况下，是不需要写续行符的呢？

经过总结，在这些符号中间的代码换行可以省略掉续行符：`[]`，`()`，`{}`

```
>>> my_list=[1,2,3,\n...         4,5,6]\n\n>>> my_tuple=(1,2,3,\n...           4,5,6)\n\n>>> my_dict={"name": "MING",\n...          "gender": "male"}
```

另外还有，在多行文本注释中 `'''`，续行符也是可以不写的。

```
>>> text = '''talk is cheap,\n...         show me the code'''
```

15. Python2下 也能使用 print(“”)

可能会有不少人，觉得只有 Python 3 才可以使用 `print()`，而 Python 2 只能使用 `print ""`。

但是其实并不是这样的。

在Python 2.6之前，只支持

```
print "hello"
```

在Python 2.6和2.7中，可以支持如下三种

```
print "hello"  
print("hello")  
print ("hello")
```

在Python3.x中，可以支持如下两种

```
print("hello")  
print ("hello")
```

虽然 在 Python 2.6+ 可以和 Python3.x+ 一样，像函数一样去调用 print ，但是这仅用于两个 python 版本之间的代码兼容，并不是说在 python2.6+下使用 print() 后，就成了函数。

作者：王炳明，微信公众号《Python编程时光》。版权归个人所有，欢迎分享。仅用于学习交流，但勿用作商业用途，违者必究。

16. 迷一样的字符串

示例一

```
# Python2.7  
>>> a = "Hello_Python"  
>>> id(a)  
32045616  
>>> id("Hello" + "_" + "Python")  
32045616  
  
# Python3.7  
>>> a = "Hello_Python"  
>>> id(a)  
38764272  
>>> id("Hello" + "_" + "Python")  
32045616
```

示例二

```
>>> a = "MING"
>>> b = "MING"
>>> a is b
True

# Python2.7
>>> a, b = "MING!", "MING!"
>>> a is b
True

# Python3.7
>>> a, b = "MING!", "MING!"
>>> a is b
False
```

示例三

```
# Python2.7
>>> 'a' * 20 is 'aaaaaaaaaaaaaaaaaaaaaa'
True
>>> 'a' * 21 is 'aaaaaaaaaaaaaaaaaaaaaa'
False

# Python3.7
>>> 'a' * 20 is 'aaaaaaaaaaaaaaaaaaaaaa'
True
>>> 'a' * 21 is 'aaaaaaaaaaaaaaaaaaaaaa'
True
```

17. return不一定是函数的终点

众所周知，try...finally... 的用法是：不管try里面是正常执行还是有报异常，最终都能保证finally能够执行。

同时我们又知道，一个函数里只要遇到 return 函数就会立马结束。

那问题就来了，以上这两种规则，如果同时存在，Python 解释器会如何选择？哪个优先级更高？

写个示例验证一下，就明白啦

```
>>> def func():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> func()
'finally'
```

从输出中，我们可以发现：在try...finally...语句中，try中的 return 会被直接忽视（这里的 return 不是函数的终点），因为要保证 finally 能够执行。

如果 try 里的 return 真的是直接被忽视吗？

我们都知道如果一个函数没有 return，会隐式的返回 None，假设 try 里的 return 真的是直接被忽视，那当finally 下没有显式的 return 的时候，是不是会返回None呢？

还是写个 示例来验证一下：

```
>>> def func():
...     try:
...         return 'try'
...     finally:
...         print('finally')
...
>>>
>>> func()
finally
'try'
>>>
```

从结果来看，当 finally 下没有 return，其实 try 里的 return 仍然还是有效的。

那结论就出来了，如果 finally 里有显式的 return，那么这个 return 会直接覆盖 try 里的 return，而如果 finally 里没有 显式的 return，那么 try 里的 return 仍然有效。

18. 用户无感知的小整数池

为避免整数频繁申请和销毁内存空间，Python 定义了一个小整数池 [-5, 256] 这些整数对象是提前建立好的，不会被垃圾回收。

以上代码请在 终端Python环境下测试，如果你是在IDE中测试，由于 IDE 的影响，效果会有所不同。

```
>>> a = -6
>>> b = -6
>>> a is b
False

>>> a = 256
>>> b = 256
>>> a is b
True

>>> a = 257
>>> b = 257
>>> a is b
False

>>> a = 257; b = 257
>>> a is b
True
```

问题又来了：最后一个示例，为啥是True？

因为当你在同一行里，同时给两个变量赋同一值时，解释器知道这个对象已经生成，那么它就会引用到同一个对象。如果分成两成的话，解释器并不知道这个对象已经存在了，就会重新申请内存存放这个对象。

19. 神奇的 intern 机制

字符串类型作为Python中最常用的数据类型之一，Python解释器为了提高字符串使用的效率和使用性能，做了很多优化。

例如：Python解释器中使用了 intern（字符串驻留）的技术来提高字符串效率，什么是intern机制？就是同样的字符串对象仅仅会保存一份，放在一个字符串储蓄池中，是共用的，当然，肯定不能改变，这也决定了字符串必须是不可变对象。

```
>>> s1="hello"
>>> s2="hello"
>>> s1 is s2
True

# 如果有空格，默认不启用intern机制
>>> s1="hell o"
>>> s2="hell o"
>>> s1 is s2
False

# 如果一个字符串长度超过20个字符，不启动intern机制
>>> s1 = "a" * 20
>>> s2 = "a" * 20
>>> s1 is s2
True

>>> s1 = "a" * 21
>>> s2 = "a" * 21
>>> s1 is s2
False

>>> s1 = "ab" * 10
>>> s2 = "ab" * 10
>>> s1 is s2
True

>>> s1 = "ab" * 11
>>> s2 = "ab" * 11
>>> s1 is s2
False
```

20. 反转字符串/列表最优雅的方式

反转序列并不难，但是如何做到最优雅呢？

先来看看，正常是如何反转的。

最简单的方法是使用列表自带的reverse()方法。

```
>>> ml = [1,2,3,4,5]
>>> ml.reverse()
>>> ml
[5, 4, 3, 2, 1]
```

但如果你要处理的是字符串，reverse就无能为力了。你可以尝试将其转化成list，再reverse，然后再转化成str。转来转去，也太麻烦了吧？需要这么多行代码（后面三行是不能合并成一行的），一点都Pythonic。

```
mstr1 = 'abc'
ml1 = list(mstr1)
ml1.reverse()
mstr2 = str(ml1)
```

对于字符串还有一种稍微复杂一点的，是自定义递归函数来实现。

```
def my_reverse(str):
    if str == "":
        return str
    else:
        return my_reverse(str[1:]) + str[0]
```

在这里，介绍一种最优雅的反转方式，使用切片，不管你是字符串，还是列表，简直通杀。

```
>>> mstr = 'abc'
>>> ml = [1,2,3]
>>> mstr[::-1]
'cba'
>>> ml[::-1]
[3, 2, 1]
```

作者：王炳明，微信公众号《Python编程时光》。版权归个人所有，欢迎分享。仅用于学习交流，但勿用作商业用途，违者必究。

21. 改变默认递归次数限制

上面才提到递归，大家都知道使用递归是有风险的，递归深度过深容易导致堆栈的溢出。如果你这字符串太长啦，使用递归方式反转，就会出现问题的。

那到底，默认递归次数限制是多少呢？

```
>>> import sys
>>> sys.getrecursionlimit()
1000
```

可以查，当然也可以自定义修改次数，退出即失效。

```
>>> sys.setrecursionlimit(2000)
>>> sys.getrecursionlimit()
2000
```

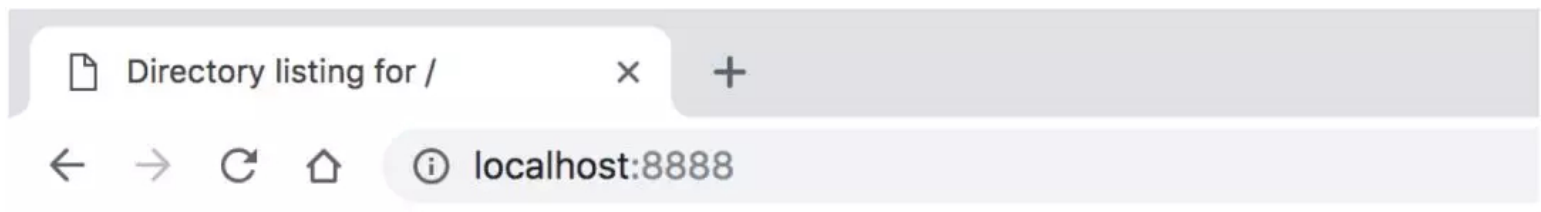
22. 一行代码实现FTP服务器

搭建FTP，或者是搭建网络文件系统，这些方法都能够实现Linux的目录共享。但是FTP和网络文件系统的功能都过于强大，因此它们都有一些不够方便的地方。比如你想快速共享Linux系统的某个目录给整个项目团队，还想在一分钟内做到，怎么办？很简单，使用Python中的SimpleHTTPServer。

SimpleHTTPServer是Python 2自带的一个模块，是Python的Web服务器。它在Python 3已经合并到http.server模块中。具体例子如下，如不指定端口，则默认是8000端口。

```
# python2
python -m SimpleHTTPServer 8888

# python3
python3 -m http.server 8888
```



Directory listing for /

- [.bash_history](#)
- [.bash_sessions/](#)
- [.CFUserTextEncoding](#)
- [.DS_Store](#)
- [.matplotlib/](#)
- [.oracle_jre_usage/](#)
- [.Trash/](#)
- [.viminfo](#)
- [Applications/](#)
- [Desktop/](#)
- [Documents/](#)
- [Downloads/](#)

SimpleHTTPServer有一个特性，如果待共享的目录下有index.html，那么index.html文件会被视为默认主页；如果不存在index.html文件，那么就会显示整个目录列表。

23. 让你晕头转向的 else 用法

if else 用法可以说最基础的语法表达式之一，但是今天不是讲这个的，一定要讲点不一样的。

if else 早已烂大街，但可能有很多人都不曾见过 for else 和 try else 的用法。为什么说它曾让我晕头转向，因为它不像 if else 那么直白，非黑即白，脑子经常要想一下才能才反应过来代码怎么走。反正我是这样的。

先来说说，for else

```
def check_item(source_list, target):
```

```
for item in source_list:
    if item == target:
        print("Exists!")
        break

else:
    print("Does not exist")
```

在往下看之前，你可以思考一下，什么情况下才会走 else。是循环被 break，还是没有break？

给几个例子，你体会一下。

```
check_item(["apple", "huawei", "oppo"], "oppo")
# Exists!

check_item(["apple", "huawei", "oppo"], "vivo")
# Does not exist
```

可以看出，没有被 break 的程序才会正常走else流程。

再看看，try else 用法。

```
def test_try_else(attr1 = None):
    try:
        if attr1:
            pass
        else:
            raise
    except:
        print("Exception occurred...")
    else:
        print("No Exception occurred...")
```

同样来几个例子。当不传参数时，就抛出异常。

```
test_try_else()
# Exception occurred...

test_try_else("ming")
```

```
# No Exception occurred...
```

可以看出，没有 try 里面的代码块没有抛出异常的，会正常走else。

总结一下，for else 和 try else 相同，只要代码正常走下去不被 break，不抛出异常，就可以走 else。

24. 字符串里的缝隙是什么？

在Python中求一个字符串里，某子字符（串）出现的次数。

大家都懂得使用 count() 函数，比如下面几个常规例子：

```
>>> "aabb".count("a")
2
>>> "aabb".count("b")
2
>>> "aabb".count("ab")
1
```

但是如果我想计算空字符串的个数呢？

```
>>> "aabb".count("")
5
```

奇怪了吧？

不是应该返回 0 吗？怎么会返回 5？

实际上，在 Python 看来，两个字符之间都是一个空字符，通俗的说就是缝隙。

因此 对于 `aabb` 这个字符串在 Python 来看应该是这样的



字符

缝隙

理解了這個“縫隙”的概念后，以下這些就好理解了。

```
>>> (" " * 10).count("")
11
>>>
>>> "" in ""
True
>>>
>>> "" in "M"
True
```

25. 正负得正，负负得正

从初中开始，我们就开始接触了 **负数**，并且都知道了 **负负得正** 的思想。

Python 作为一门高级语言，它的编写符合人类的思维逻辑，包括 **负负得正**。

```
>>> 5-3
2
>>> 5--3
8
>>> 5+-3
2
>>> 5++3
8
>>> 5---3
2
```

作者：王炳明，微信公众号《Python编程时光》。版权归个人所有，欢迎分享。仅用于学习交流，但勿用作商业用途，违者必究。

26. 数值与字符串的比较

在 Python2 中，数字可以与字符串直接比较。结果是数值永远比字符串小。

```
>>> 1000000000 < ""
True
>>> 1000000000 < "hello"
True
```

但在 Python3 中，却不行。

```
>>> 1000000000 < ""
TypeError: '<' not supported between instances of 'int' and 'str'
```

27. 循环中的局部变量泄露

在Python 2中 x 的值在一个循环执行之后被改变了。

```
# Python2
>>> x = 1
>>> [x for x in range(5)]
[0, 1, 2, 3, 4]
>>> x
4
```

不过在Python3 中这个问题已经得到解决了。

```
# Python3
>>> x = 1
>>> [x for x in range(5)]
[0, 1, 2, 3, 4]
```

```
>>> x
1
```

28. 字典居然是可以排序的？

在 Python 3.6 之前字典不可排序的思想，似乎已经根深蒂固。

```
# Python2.7.10
>>> mydict = {str(i):i for i in range(5)}
>>> mydict
{'1': 1, '0': 0, '3': 3, '2': 2, '4': 4}
```

假如哪一天，有人跟你说字典也可以是有顺序的，不要惊讶，那确实是真的

在 Python3.6 + 中字典已经是有序的，并且效率相较之前的还有所提升，具体信息你可以去查询相关资料。

```
# Python3.6.7
>>> mydict = {str(i):i for i in range(5)}
>>> mydict
{'0': 0, '1': 1, '2': 2, '3': 3, '4': 4}
```

29. 有趣但没啥用的 import 用法

import 是 Python 导包的方式。

你知道 Python 中内置了一些很有（wu）趣（liao）的包吗？

Hello World

```
>>> import __hello__
Hello World!
```

Python之禅

```
>>> import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

反地心引力漫画

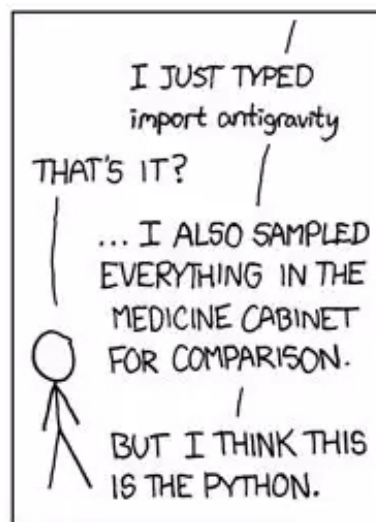
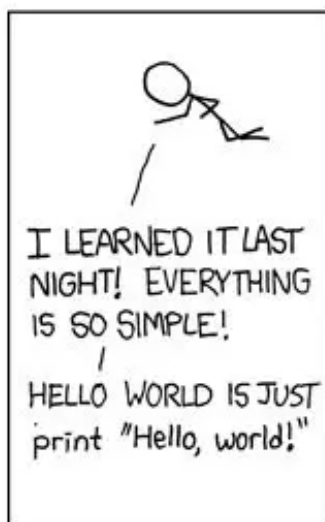
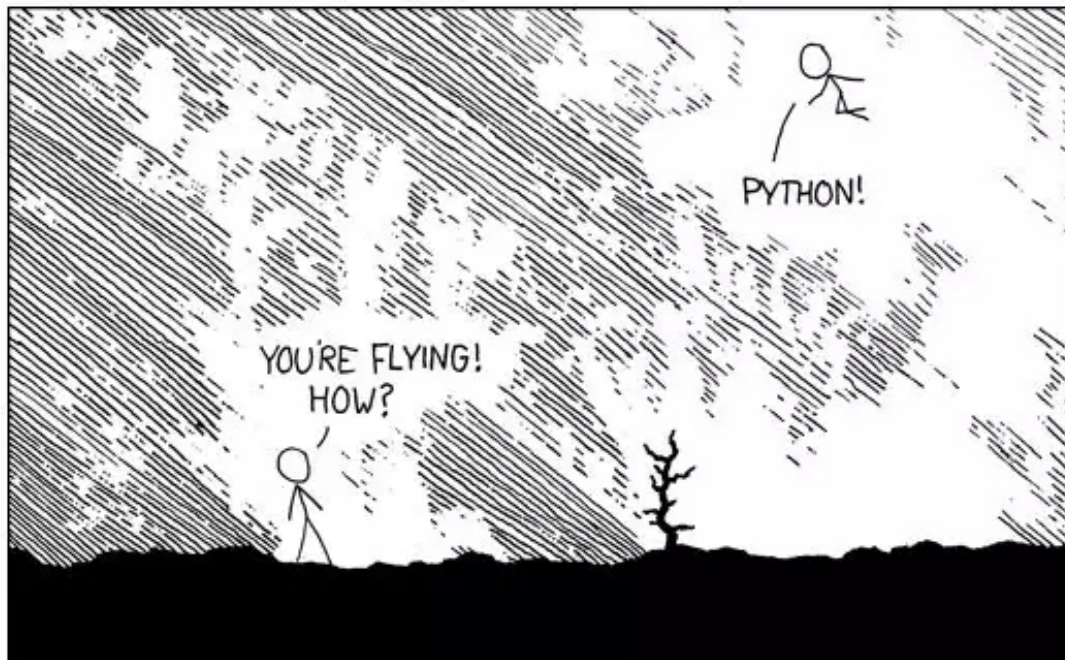
在 cmd 窗口中导入 `antigravity`

```
>>> import antigravity
```

就会自动打开一个网页。

PYTHON

< PREV RANDOM NEXT >



< PREV RANDOM NEXT >

30. 局部/全局变量傻傻分不清

在开始讲之前，你可以试着运行一下下面这小段代码。

```
# demo.py
```

```
a = 1

def add():
    a += 1

add()
```

看似没有毛病，但实则已经犯了一个很基础的问题，运行结果如下：

```
$ python demo.py
Traceback (most recent call last):
  File "demo.py", line 6, in <module>
    add()
  File "demo.py", line 4, in add
    a += 1
UnboundLocalError: local variable 'a' referenced before assignment
```

回顾一下，什么是局部变量？在非全局下定义声明的变量都是局部变量。

当程序运行到 `a += 1` 时，Python 解释器就认为在函数内部要给 `a` 这个变量赋值，当然就把 `a` 当做局部变量了，但是做为局部变量的 `a` 还没有被定义。

因此报错是正常的。

理解了上面的例子，给你留个思考题。为什么下面的代码不会报错呢？

```
$ cat demo.py
a = 1

def output():
    print(a)

output()

$ python demo.py
1
```

作者：王炳明，微信公众号《Python编程时光》。版权归个人所有，欢迎分享。仅用于学习交流，但勿用作商业用途，违者必究。

31. 字母也玩起了障眼法

以下我分别在 Python2.7 和 Python 3.7 的 console 模式下，运行了如下代码。

在Python 2.x 中

```
>>> value = 32
File "<stdin>", line 1
    value = 32
        ^
SyntaxError: invalid syntax
```

在Python 3.x 中

```
>>> value = 32
>>> value
11
```

什么？没有截图你不信？

```

[wangbm@35ha02 ansible]$ python
Python 2.7.5 (default, Oct 30 2018, 23:45:53)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> value = 32
      File "<stdin>", line 1
        value = 32
            ^
SyntaxError: invalid syntax
>>>
[wangbm@35ha02 ansible]$
[wangbm@35ha02 ansible]$ python3
Python 3.7.1 (default, Dec 19 2018, 13:22:32)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> value = 32
>>>
>>> █

```

如果你在自己的电脑上尝试一下，结果可能是这样的

```

[wangbm@35ha02 ansible]$
[wangbm@35ha02 ansible]$ python
Python 2.7.5 (default, Oct 30 2018, 23:45:53)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> value = 32
>>>
>>>
[wangbm@35ha02 ansible]$
[wangbm@35ha02 ansible]$ python3
Python 3.7.1 (default, Dec 19 2018, 13:22:32)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> value = 32
>>>
>>>
[wangbm@35ha02 ansible]$ █

```

怎么又好了呢？

如果你想复现的话，请复制我这边给出的代码： `value = 32`

这是为什么呢？

原因在于，我上面使用的 `value` 变量名里的 `е` 又不是我们熟悉的 `e`，它是 Cyrillic（西里尔）字母。

```
>>> ord('е') # cyrillic 'e' (Ye)
```



```
1077
>>> ord('e') # latin 'e', as used in English and typed using standard keyboard
101
>>> 'e' == 'e'
False
```

细思恐极，在这里可千万不要得罪同事们，万一离职的时候，对方把你项目里的 `e` 全局替换成 `e`，到时候你就哭去吧，肉眼根本看不出来嘛。

32. 字符串的分割技巧

当我们对字符串进行分割时，且分割符是 `\n`，有可能会出现这样一个窘境：

```
>>> str = "a\nb\n"
>>> print(str)
a
b

>>> str.split('\n')
['a', 'b', '']
>>>
```

会在最后一行多出一个元素，为了应对这种情况，你可以会多加一步处理。

但我想说的是，完成没有必要，对于这个场景，你可以使用 `splitlines`

```
>>> str.splitlines()
['a', 'b']
```

33. 嵌套上下文管理的另类写法

当我们要写一个嵌套的上下文管理器时，可能会这样写

```
import contextlib

@contextlib.contextmanager
def test_context(name):
    print('enter, my name is {}'.format(name))
```

```

yield

print('exit, my name is {}'.format(name))

with test_context('aaa'):
    with test_context('bbb'):
        print('===== in main =====')
```

输出结果如下

```

enter, my name is aaa
enter, my name is bbb
===== in main =====
exit, my name is bbb
exit, my name is aaa
```

除此之外，你可知道，还有另一种嵌套写法

```

with test_context('aaa'), test_context('bbb'):
    print('===== in main =====')
```

34. += 不等于=+

对列表 进行 `+=` 操作相当于 `extend`，而使用 `=+` 操作是新增了一个列表。

因此会有如下两者的差异。

```

# +=
>>> a = [1, 2, 3, 4]
>>> b = a
>>> a = a + [5, 6, 7, 8]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8]
>>> b
[1, 2, 3, 4]
```

```

# =+
```

```
>>> a = [1, 2, 3, 4]
>>> b = a
>>> a += [5, 6, 7, 8]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8]
>>> b
[1, 2, 3, 4, 5, 6, 7, 8]
```

35. 增量赋值的性能更好

诸如 `+=` 和 `*=` 这些运算符，叫做 增量赋值运算符。

这里使用 `+=` 举例，以下两种写法，在效果上是等价的。

```
# 第一种
a = 1 ; a += 1

# 第二种
a = 1; a = a + 1
```

`+=` 其背后使用的魔法方法是 `__iadd__`，如果没有实现这个方法则会退而求其次，使用 `__add__`。

这两种写法有什么区别呢？

用列表举例 `a += b`，使用 `__add__` 的话就像是使用了 `a.extend(b)`，如果使用 `__iadd__` 的话，则是 `a = a+b`，前者是直接在原列表上进行扩展，而后者是先从原列表中取出值，在一个新的列表中进行扩展，然后再将新的列表对象返回给变量，显然后者的消耗要大些。

所以在能使用增量赋值的时候尽量使用它。

作者：王炳明，微信公众号《Python编程时光》。版权归个人所有，欢迎分享。仅用于学习交流，但勿用作商业用途，违者必究。

36. `x == +x` 吗？

在大多数情况下，这个等式是成立的。

```
>>> n1 = 10086
>>> n2 = +n1
>>>
>>> n1 == n2
True
```

什么情况下，这个等式会不成立呢？

由于Counter的机制，`+` 用于两个 Counter 实例相加，而相加的结果如果元素的个数 `<=` 0，就会被丢弃。

```
>>> from collections import Counter
>>> ct = Counter('abcdbcaa')
>>> ct
Counter({'a': 3, 'b': 2, 'c': 2, 'd': 1})
>>> ct['c'] = 0
>>> ct['d'] = -2
>>>
>>> ct
Counter({'a': 3, 'b': 2, 'c': 0, 'd': -2})
>>>
>>> +ct
Counter({'a': 3, 'b': 2})
```

37. 如何将 print 内容输出到文件

Python 3 中的 print 作为一个函数，由于可以接收更多的参数，所以功能变为更加强大。

比如今天要说的使用 print 将你要打印的内容，输出到日志文件中（但是我并不推荐使用它）。

```
>>> with open('test.log', mode='w') as f:
...     print('hello, python', file=f, flush=True)
>>> exit()

$ cat test.log
hello, python
```

38. site-packages和 dist-packages

如果你足够细心，你会在你的机器上，有些包是安装在 **site-packages** 下，而有些包安装在 **dist-packages** 下。

它们有什么区别呢？

一般情况下，你只见过 **site-packages** 这个目录，而你所安装的包也将安装在这个目录下。

而 **dist-packages** 其实是 debian 系的 Linux 系统（如 Ubuntu）才特有的目录，当你使用 **apt** 去安装的 Python 包会使用 **dist-packages**，而你使用 **pip** 或者 **easy_install** 安装的包还是照常安装在 **site-packages** 下。

Debian 这么设计的原因，是为了减少不同来源的 Python 之间产生的冲突。

如何查找 Python 安装目录

```
>>> from distutils.sysconfig import get_python_lib
>>> print(get_python_lib())
/usr/lib/python2.7/site-packages
```

39. argument 和 parameter 的区别

arguments 和 **parameter** 的翻译都是参数，在中文场景下，二者混用基本没有问题，毕竟都叫参数嘛。

但若要严格再进行区分，它们实际上还有各自的叫法

- **parameter**：形参（**formal parameter**），体现在函数内部，作用域是这个函数体。
- **argument**：实参（**actual parameter**），调用函数实际传递的参数。

举个例子，如下这段代码，`"error"` 为 **argument**，而 `msg` 为 **parameter**。

```
def output_msg(msg):
    print(msg)

output_msg("error")
```

40. 简洁而优雅的链式比较

先给你看一个示例：

```
>>> False == False == True
False
```

你知道这个表达式为什么会返回 False 吗？

它的运行原理与下面这个类似，是不是有点头绪了：

```
if 80 < score <= 90:
    print("成绩良好")
```

如果你还是不明白，那我再给你整个第一个例子的等价写法。

```
>>> False == False and False == True
False
```

这个用法叫做链式比较。

作者：王炳明，微信公众号《Python编程时光》。版权归个人所有，欢迎分享。仅用于学习交流，但勿用作商业用途，违者必究。

41. 连接多个列表最极客的方式

```
>>> a = [1,2]
>>> b = [3,4]
>>> c = [5,6]
>>>
>>> sum((a,b,c), [])
[1, 2, 3, 4, 5, 6]
```

42. 另外 8 种连接列表的方式

1. 最直观的相加

使用 `+` 对多个列表进行相加，你应该懂，不多说了。

```
>>> list01 = [1,2,3]
>>> list02 = [4,5,6]
>>> list03 = [7,8,9]
>>>
>>> list01 + list02 + list03
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

2. 借助 itertools

`itertools` 在 Python 里有一个非常强大的内置模块，它专门用于操作可迭代对象。

在前面的文章中也介绍过，使用 `itertools.chain()` 函数先可迭代对象（在这里指的是列表）串联起来，组成一个更大的可迭代对象。

最后你再利用 `list` 将其转化为 列表。

```
>>> from itertools import chain
>>> list01 = [1,2,3]
>>> list02 = [4,5,6]
>>> list03 = [7,8,9]
>>>
>>> list(chain(list01, list02, list03))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

3. 使用 * 解包

使用 `*` 可以解包列表，解包后再合并。

示例如下：

```
>>> list01 = [1,2,3]
>>> list02 = [4,5,6]
```

```
>>>
>>> [*list01, *list02]
[1, 2, 3, 4, 5, 6]
>>>
```

4. 使用 extend

在字典中，使用 update 可实现原地更新，而在列表中，使用 extend 可实现列表的自我扩展。

```
>>> list01 = [1,2,3]
>>> list02 = [4,5,6]
>>>
>>> list01.extend(list02)
>>> list01
[1, 2, 3, 4, 5, 6]
```

5. 使用列表推导式

Python 里对于生成列表、集合、字典，有一套非常 Pythonic 的写法。

那就是列表解析式，集合解析式和字典解析式，通常是 Python 发烧友的最爱，那么今天的主题：列表合并，列表推导式还能否胜任呢？

当然可以，具体示例代码如下：

```
>>> list01 = [1,2,3]
>>> list02 = [4,5,6]
>>> list03 = [7,8,9]
>>>
>>> [x for l in (list01, list02, list03) for x in l]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

6. 使用 heapq

heapq 是 Python 的一个标准模块，它提供了堆排序算法的实现。

该模块里有一个 merge 方法，可以用于合并多个列表，如下所示

```
>>> list01 = [1,2,3]
```



```
>>> list02 = [4,5,6]
>>> list03 = [7,8,9]
>>>
>>> from heapq import merge
>>>
>>> list(merge(list01, list02, list03))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

要注意的是，`heapq.merge` 除了合并多个列表外，它还会将合并后的最终的列表进行排序。

```
>>> list01 = [2,5,3]
>>> list02 = [1,4,6]
>>> list03 = [7,9,8]
>>>
>>> from heapq import merge
>>>
>>> list(merge(list01, list02, list03))
[1, 2, 4, 5, 3, 6, 7, 9, 8]
>>>
```

它的效果等价于下面这行代码：

```
sorted(itertools.chain(*iterables))
```

如果你希望得到一个始终有序的列表，那请第一时间想到 `heapq.merge`，因为它采用堆排序，效率非常高。但若你不希望得到一个排过序的列表，就不要使用它了。

7. 借助魔法方法

有一个魔法方法叫 `__add__`，当我们使用第一种方法 `list01 + list02` 的时候，内部实际上是作用在 `__add__` 这个魔法方法上的。

所以以下两种方法其实是等价的

```
>>> list01 = [1,2,3]
>>> list02 = [4,5,6]
>>>
>>> list01 + list02
[1, 2, 3, 4, 5, 6]
```

```
>>>
>>>
>>> list01.__add__(list02)
[1, 2, 3, 4, 5, 6]
>>>
```

借用这个魔法特性，我们可以 reduce 这个方法对多个列表进行合并，示例代码如下

```
>>> list01 = [1,2,3]
>>> list02 = [4,5,6]
>>> list03 = [7,8,9]
>>>
>>> from functools import reduce
>>> reduce(list.__add__, (list01, list02, list03))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

8. 使用 yield from

在 yield from 后可接一个可迭代对象，用于迭代并返回其中的每一个元素。

因此，我们可以像下面这样自定义一个合并列表的工具函数。

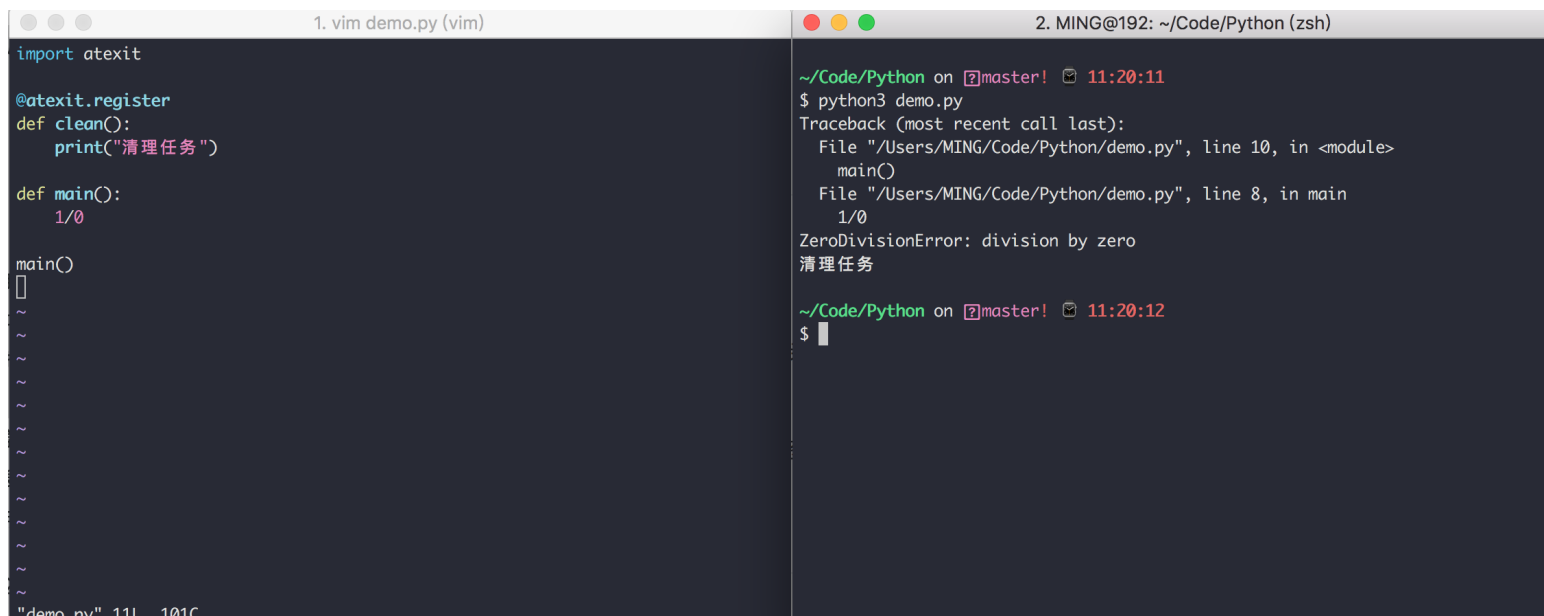
```
>>> list01 = [1,2,3]
>>> list02 = [4,5,6]
>>> list03 = [7,8,9]
>>>
>>> def merge(*lists):
...     for l in lists:
...         yield from l
...
>>> list(merge(list01, list02, list03))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

43. 在程序退出前执行代码的技巧

使用 atexit 这个内置模块，可以很方便的注册退出函数。

不管你在哪个地方导致程序崩溃，都会执行那些你注册过的函数。

示例如下



```
1. vim demo.py (vim)
import atexit

@atexit.register
def clean():
    print("清理任务")

def main():
    1/0

main()

2. MING@192: ~/Code/Python (zsh)
~/Code/Python on master! 11:20:11
$ python3 demo.py
Traceback (most recent call last):
  File "/Users/MING/Code/Python/demo.py", line 10, in <module>
    main()
  File "/Users/MING/Code/Python/demo.py", line 8, in main
    1/0
ZeroDivisionError: division by zero
清理任务

~/Code/Python on master! 11:20:12
$
```

如果 `clean()` 函数有参数，那么你可以不用装饰器，而是直接调用 `atexit.register(clean_1, 参数1, 参数2, 参数3='xxx')`。

可能你有其他方法可以处理这种需求，但肯定比上不使用 `atexit` 来得优雅，来得方便，并且它很容易扩展。

但是使用 `atexit` 仍然有一些局限性，比如：

- 如果程序是被你没有处理过的系统信号杀死的，那么注册的函数无法正常执行。
- 如果发生了严重的 Python 内部错误，你注册的函数无法正常执行。
- 如果你手动调用了 `os._exit()`，你注册的函数无法正常执行。

44. 合并字典的 8 种方法

1. 最简单的原地更新

字典对象内置了一个 `update` 方法，用于把另一个字典更新到自己身上。

```
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info = {"gender": "male"}
>>>
>>> profile.update(ext_info)
>>> print(profile)
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
```

如果想使用 update 这种最简单、最地道原生的方法，但又不想更新到自己身上，而是生成一个新的对象，那请使用深拷贝。

```
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info = {"gender": "male"}
>>>
>>> from copy import deepcopy
>>>
>>> full_profile = deepcopy(profile)
>>> full_profile.update(ext_info)
>>>
>>> print(full_profile)
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
>>> print(profile)
{"name": "xiaoming", "age": 27}
```

2. 先解包再合并字典

使用 `**` 可以解包字典，解包完后再使用 `dict` 或者 `{}` 就可以合并。

```
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info = {"gender": "male"}
>>>
>>> full_profile01 = {**profile, **ext_info}
>>> print(full_profile01)
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
>>>
>>> full_profile02 = dict(**profile, **ext_info)
>>> print(full_profile02)
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
```

若你不知道 `dict(**profile, **ext_info)` 做了啥，你可以将它等价于

```
>>> dict(("name", "xiaoming"), ("age", 27), ("gender", "male"))
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
```

3. 借助 itertools

在 Python 里有一个非常强大的内置模块，它专门用于操作可迭代对象。

正好我们字典也是可迭代对象，自然就可以想到，可以使用 `itertools.chain()` 函数先将多个字典（可迭代对象）串联起来，组成一个更大的可迭代对象，然后再使用 `dict` 转成字典。

```
>>> import itertools
>>>
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info = {"gender": "male"}
>>>
>>>
>>> dict(itertools.chain(profile.items(), ext_info.items()))
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
```

4. 借助 ChainMap

如果可以引入一个辅助包，那我就再提一个，`ChainMap` 也可以达到和 `itertools` 同样的效果。

```
>>> from collections import ChainMap
>>>
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info = {"gender": "male"}
>>>
>>> dict(ChainMap(profile, ext_info))
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
```

使用 `ChainMap` 有一点需要注意，当字典间有重复的键时，只会取第一个值，排在后面的键值并不会更新掉前面的（使用 `itertools` 就不会有这个问题）。

```
>>> from collections import ChainMap
>>>
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info={"age": 30}
>>> dict(ChainMap(profile, ext_info))
{'name': 'xiaoming', 'age': 27}
```

5. 使用dict.items() 合并

在 Python 3.9 之前，其实就已经有 `|` 操作符了，只不过它通常用于对集合（set）取并集。

利用这一点，也可以将它用于字典的合并，只不过得绕个弯子，有点不好理解。

你得先利用 `items` 方法将 `dict` 转成 `dict_items`，再对这两个 `dict_items` 取并集，最后利用 `dict` 函数，转成字典。

```
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info = {"gender": "male"}
>>>
>>> full_profile = dict(profile.items() | ext_info.items())
>>> full_profile
{'gender': 'male', 'age': 27, 'name': 'xiaoming'}
```

当然了，你如果嫌这样太麻烦，也可以简单点，直接使用 `list` 函数再合并（示例为 Python 3.x）

```
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info = {"gender": "male"}
>>>
>>> dict(list(profile.items()) + list(ext_info.items()))
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
```

若你在 Python 2.x 下，可以直接省去 `list` 函数。

```
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info = {"gender": "male"}
>>>
>>> dict(profile.items() + ext_info.items())
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
```

6. 最酷炫的字典解析式

Python 里对于生成列表、集合、字典，有一套非常 Pythonic 的写法。

那就是列表解析式，集合解析式和字典解析式，通常是 Python 发烧友的最爱，那么今天的主题：字典合并，字典解析式还能否胜任呢？

当然可以，具体示例代码如下：

```
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info = {"gender": "male"}
```

```
>>>
>>> {k:v for d in [profile, ext_info] for k,v in d.items()}
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
```

7. Python 3.9 新特性

在 2 月份发布的 Python 3.9.04a 版本中，新增了一个抓眼球的新操作符操作符：`|`，PEP584 将它称之为合并操作符（Union Operator），用它可以很直观地合并多个字典。

```
>>> profile = {"name": "xiaoming", "age": 27}
>>> ext_info = {"gender": "male"}
>>>
>>> profile | ext_info
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
>>>
>>> ext_info | profile
{'gender': 'male', 'name': 'xiaoming', 'age': 27}
>>>
>>>
```

除了 `|` 操作符之外，还有另外一个操作符 `|=`，类似于原地更新。

```
>>> ext_info |= profile
>>> ext_info
{'gender': 'male', 'name': 'xiaoming', 'age': 27}
>>>
>>>
>>> profile |= ext_info
>>> profile
{'name': 'xiaoming', 'age': 27, 'gender': 'male'}
```

看到这里，有没有涨姿势了，学了这么久的 Python，没想到合并字典还有这么多的方法。本篇文章的主旨，并不在于让你全部掌握这 7 种合并字典的方法，实际在工作中，你只要选用一种最顺手的方式即可，但是在协同工作中，或者在阅读他人代码时，你不可避免地会碰到各式各样的写法，这时候你能下意识的知道这是在做合并字典的操作，那这篇文章就是有意义的。

45. 条件语句的七种写法

第一种：原代码

这是一段非常简单的通过年龄判断一个人是否成年的代码，由于代码行数过多，有些人就不太愿意这样写，因为这体现不出自己多年的 Python 功力。

```
if age > 18:
    return "已成年"
else:
    return "未成年"
```

下面我列举了六种这段代码的变异写法，一个比一个还 6，单独拿出来比较好理解，放在工程代码里，没用过这些学法的人，一定会看得一脸懵逼，理解了之后，又不经意大呼：卧槽，还可以这样写？，而后就要开始骂街了：这是给人看的代码？（除了第一种之外）

第二种

语法：

```
<on_true> if <condition> else <on_false>
```

例子

```
>>> age1 = 20
>>> age2 = 17
>>>
>>>
>>> msg1 = "已成年" if age1 > 18 else "未成年"
>>> print msg1
已成年
>>>
>>> msg2 = "已成年" if age2 > 18 else "未成年"
>>> print msg2
未成年
>>>
```

第三种

语法

```
<condition> and <on_true> or <on_false>
```


例子

```
>>> msg1 = age1 > 18 and "已成年" or "未成年"
>>> msg2 = "已成年" if age2 > 18 else "未成年"
>>>
>>> print(msg1)
已成年
>>>
>>> print(msg2)
未成年
```

第四种

语法

```
(<on_true>, <on_false>)[condition]
```

例子

```
>>> msg1 = ("未成年", "已成年")[age1 > 18]
>>> print(msg1)
已成年
>>>
>>>
>>> msg2 = ("未成年", "已成年")[age2 > 18]
>>> print(msg2)
未成年
```

第五种

语法

```
(lambda: <on_false>, lambda:<on_true>)[<condition>]()
```

例子

```
>>> msg1 = (lambda:"未成年", lambda:"已成年")[age1 > 18]()
>>> print(msg1)
```

```
已成年
>>>
>>> msg2 = (lambda: "未成年", lambda: "已成年")[age2 > 18]()
>>> print(msg2)
未成年
```

第六种

语法：

```
{True: <on_true>, False: <on_false>}[<condition>]
```

例子：

```
>>> msg1 = {True: "已成年", False: "未成年"}[age1 > 18]
>>> print(msg1)
已成年
>>>
>>> msg2 = {True: "已成年", False: "未成年"}[age2 > 18]
>>> print(msg2)
未成年
```

第七种

语法

```
((<condition>) and (<on_true>,) or (<on_false>,))[0]
```

例子

```
>>> msg1 = ((age1 > 18) and ("已成年",) or ("未成年",))[0]
>>> print(msg1)
已成年
>>>
>>> msg2 = ((age2 > 18) and ("已成年",) or ("未成年",))[0]
>>> print(msg2)
未成年
```

以上代码，都比较简单，仔细看都能看懂，我就不做解释了。

看到这里，有没有涨姿势了，学了这么久的 Python，这么多骚操作，还真是活久见。。这六种写法里，我最推荐使用的是第一种，自己也经常在用，简洁直白，代码行还少。而其他的写法虽然能写，但是不会用，也不希望在我余生里碰到会在公共代码里用这些写法的同事。

作者：王炳明，微信公众号《Python编程时光》。版权归个人所有，欢迎分享。仅用于学习交流，但勿用作商业用途，违者必究。

46. /usr/bin/env python 有什么用？

我们经常会在别人的脚本或者项目的入口文件里看到第一行是下面这样

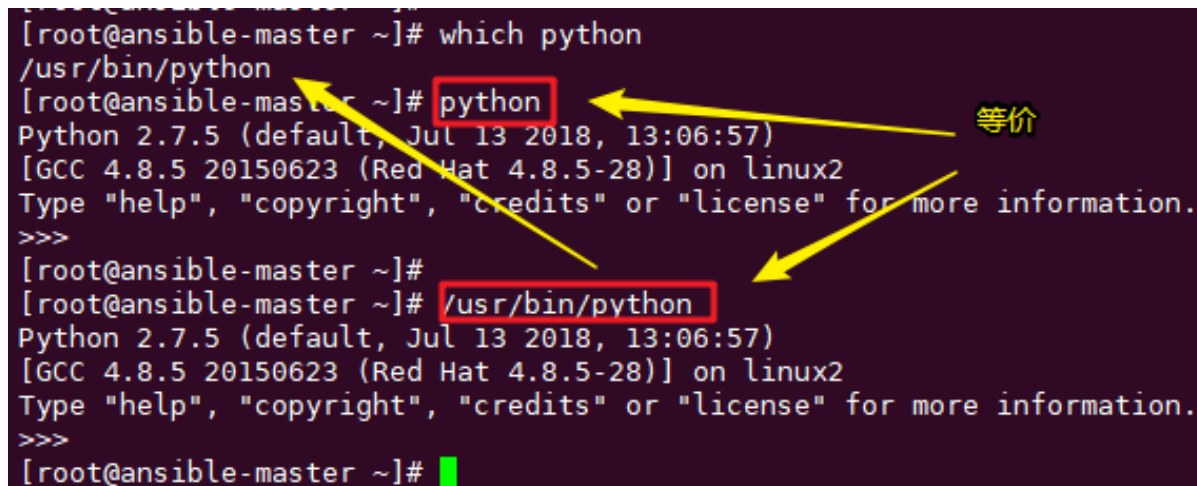
```
#!/usr/bin/python
```

或者这样

```
#!/usr/bin/env python
```

这两者有什么区别呢？

稍微接触过 linux 的人都知道 `/usr/bin/python` 就是我们执行 `python` 进入console 模式里的 `python`



The image shows a terminal window with the following commands and output:

```
[root@ansible-master ~]# which python
/usr/bin/python
[root@ansible-master ~]# python
Python 2.7.5 (default, Jul 13 2018, 13:06:57)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
[root@ansible-master ~]#
[root@ansible-master ~]# /usr/bin/python
Python 2.7.5 (default, Jul 13 2018, 13:06:57)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
[root@ansible-master ~]#
```

Yellow arrows point from the word `python` in the second command to the path `/usr/bin/python` in the third command. A yellow label `等价` (Equivalent) is placed between the two commands, indicating they are equivalent.

而你在可执行文件头里使用 `#!/usr/bin/python`，意思就是说得用哪个软件（python）来执行这个文件。

那么加和不加有什么区别呢？

不加的话，你每次执行这个脚本时，都得这样：`python xx.py`，

```
[root@ansible-master ~]# ll demo.py
-rw-r--r-- 1 root root 15 Mar 31 18:44 demo.py
[root@ansible-master ~]#
[root@ansible-master ~]# cat demo.py
print("Hello")
[root@ansible-master ~]#
[root@ansible-master ~]# python demo.py
Hello
[root@ansible-master ~]#
```

有没有一种方式？可以省去每次都加 `python` 呢？

当然有，你可以文件头里加上 `#!/usr/bin/python`，那么当这个文件有可执行权限时，只直接写这个脚本文件，就像下面这样。

```
[root@ansible-master ~]# cat demo.py
#!/usr/bin/python

print("Hello")
[root@ansible-master ~]# chmod +x demo.py
[root@ansible-master ~]# ll demo.py
-rwxr-xr-x 1 root root 34 Mar 31 18:47 demo.py
[root@ansible-master ~]#
[root@ansible-master ~]# ./demo.py
Hello
[root@ansible-master ~]#
```

明白了这个后，再来看看 `#!/usr/bin/env python` 这个又是什么意思？

当我执行 `env python` 时，自动进入了 python console 的模式。

```
[root@ansible-master ~]#
[root@ansible-master ~]# env python
Python 2.7.5 (default, Jul 13 2018, 13:06:57)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
[root@ansible-master ~]#
[root@ansible-master ~]#
[root@ansible-master ~]#
```

这是为什么？和 直接执行 python 好像没什么区别呀

当你执行 `env python` 时，它其实会去 `env | grep PATH` 里（也就是 `/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin`）这几个路径里去依次查找名为 python 的可执行文件。

找到一个就直接执行，上面我们的 python 路径是在 `/usr/bin/python` 里，在 `PATH` 列表里倒数第二个目录下，所以当我在 `/usr/local/sbin` 下创建一个名字也为 python 的可执行文件时，就会执行 `/usr/bin/python` 了。

具体演示过程，你可以看下面。

```
[root@ansible-master ~]# env | grep PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
[root@ansible-master ~]#
[root@ansible-master ~]#
[root@ansible-master ~]# env python
Python 2.7.5 (default, Jul 13 2018, 13:06:57)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
[root@ansible-master ~]#
[root@ansible-master ~]#
[root@ansible-master ~]# vim /usr/local/sbin/python
[root@ansible-master ~]# cat /usr/local/sbin/python
#!/usr/bin/python

print("Hello")
[root@ansible-master ~]#
[root@ansible-master ~]# chmod +x /usr/local/sbin/python
[root@ansible-master ~]#
[root@ansible-master ~]# env python
Hello
[root@ansible-master ~]#
```

那么对于这两者，我们应该使用哪个呢？

个人感觉应该优先使用 `#!/usr/bin/env python`，因为不是所有的机器的 python 解释器都是 `/usr/bin/python`。

47. 让我爱不释手的用户环境

当你在机器上并没有 root 权限时，如何安装 Python 的第三方包呢？

可以使用 `pip install --user pkg` 将你的包安装在你的用户环境中，该用户环境与全局环境并不冲突，并且多用户之间相互隔离，互不影响。

```
# 在全局环境中未安装 requests
[root@localhost ~]$ pip list | grep requests
[root@localhost ~]$ su - wangbm

# 由于用户环境继承自全局环境，这里也未安装
[wangbm@localhost ~]$ pip list | grep requests
[wangbm@localhost ~]$ pip install --user requests
[wangbm@localhost ~]$ pip list | grep requests
requests (2.22.0)
[wangbm@localhost ~]$

# 从 Location 属性可发现 requests 只安装在当前用户环境中
[wangbm@localhost ~]$ pip show requests
---
Metadata-Version: 2.1
Name: requests
Version: 2.22.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
Installer: pip
License: Apache 2.0
Location: /home/wangbm/.local/lib/python2.7/site-packages
[wangbm@localhost ~]$ exit
logout

# 退出 wangbm 用户，在 root 用户环境中发现 requests 未安装
[root@localhost ~]$ pip list | grep requests
[root@localhost ~]$
```

48. 实现类似 defer 的延迟调用

在 Golang 中有一种延迟调用的机制，关键字是 defer，例如下面的示例

```
import "fmt"

func myfunc() {
    fmt.Println("B")
}
```

```
func main() {  
    defer myfunc()  
    fmt.Println("A")  
}
```

输出如下，myfunc 的调用会在函数返回前一步完成，即使你将 myfunc 的调用写在函数的第一行，这就是延迟调用。

```
A  
B
```

那么在 Python 中否有这种机制呢？

当然也有，只不过并没有 Golang 这种简便。

在 Python 可以使用 **上下文管理器** 达到这种效果

```
import contextlib  
  
def callback():  
    print('B')  
  
with contextlib.ExitStack() as stack:  
    stack.callback(callback)  
    print('A')
```

输出如下

```
A  
B
```

49. 自带的缓存机制不用白不用

缓存是一种将定量数据加以保存，以备迎合后续获取需求的处理方式，旨在加快数据获取的速度。

数据的生成过程可能需要经过计算，规整，远程获取等操作，如果是同一份数据需要多次使用，每次都重新生成会大大浪费时间。所以，如果将计算或者远程请求等操作获得的数据缓存下来，会加快后

续的数据获取需求。

为了实现这个需求，Python 3.2 + 中给我们提供了一个机制，可以很方便的实现，而不需要你去写这样的逻辑代码。

这个机制实现于 `functool` 模块中的 `lru_cache` 装饰器。

```
@functools.lru_cache(maxsize=None, typed=False)
```

参数解读：

- `maxsize`：最多可以缓存多少个此函数的调用结果，如果为`None`，则无限制，设置为 2 的幂时，性能最佳
- `typed`：若为 `True`，则不同参数类型的调用将分别缓存。

举个例子

```
from functools import lru_cache

@lru_cache(None)
def add(x, y):
    print("calculating: %s + %s" % (x, y))
    return x + y

print(add(1, 2))
print(add(1, 2))
print(add(2, 3))
```

输出如下，可以看到第二次调用并没有真正的执行函数体，而是直接返回缓存里的结果

```
calculating: 1 + 2
3
3
calculating: 2 + 3
5
```

50. 重定向标准输出到日志

假设你有一个脚本，会执行一些任务，比如说集群健康情况的检查。

检查完成后，会把各服务的健康状况以 JSON 字符串的形式打印到标准输出。

如果代码有问题，导致异常处理不足，最终检查失败，是很有可能将一些错误异常栈输出到标准错误或标准输出上。

由于最初约定的脚本返回方式是以 JSON 的格式输出，此时你的脚本却输出各种错误异常，异常调用方也无法解析。

如何避免这种情况的发生呢？

我们可以这样做，把你的标准错误输出到日志文件中。

```
import contextlib

log_file="/var/log/you.log"

def you_task():
    pass

@contextlib.contextmanager
def close_stdout():
    raw_stdout = sys.stdout
    file = open(log_file, 'a+')
    sys.stdout = file

    yield

    sys.stdout = raw_stdout
    file.close()

with close_stdout():
    you_task()
```

作者：王炳明，微信公众号《Python编程时光》。版权归个人所有，欢迎分享。仅用于学习交流，但勿用作商业用途，违者必究。



打开微信扫描二维码，回复“**黑魔法**”

获取最新版《Python 黑魔法手册》