Python 黑魔法指南

作者：王炳明
版本：v2.0
发布时间：2020年05月12日
更新时间：2020年08月1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

目录大纲

第一章：魔法冷知识

1.1 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
```


1.2 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
```

> 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。

1.3 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 ~]#
```

> 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。

1.4 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="\"
  File "<stdin>", line 1
    str3="\"
          ^
SyntaxError: EOL while scanning string literal
```

就算你指定它是个 raw 字符串，也不行。

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


1.5 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
```

> 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。

1.6 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。

1.7 and 和 or 的短路效应

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
```

> 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。

1.8 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]
```

1.9 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}
```

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。

1.10 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,'\
...     'show me the code.'
>>>
```

```
>>> print(a)
talk 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,
...          4,5,6]

>>> my_tuple=(1,2,3,
...           4,5,6)

>>> my_dict={"name": "MING",
...          "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,
...           show me the code'''
```


1.11 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。

1.12 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
```


1.13 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
```


1.14 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"
```


1.15 /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

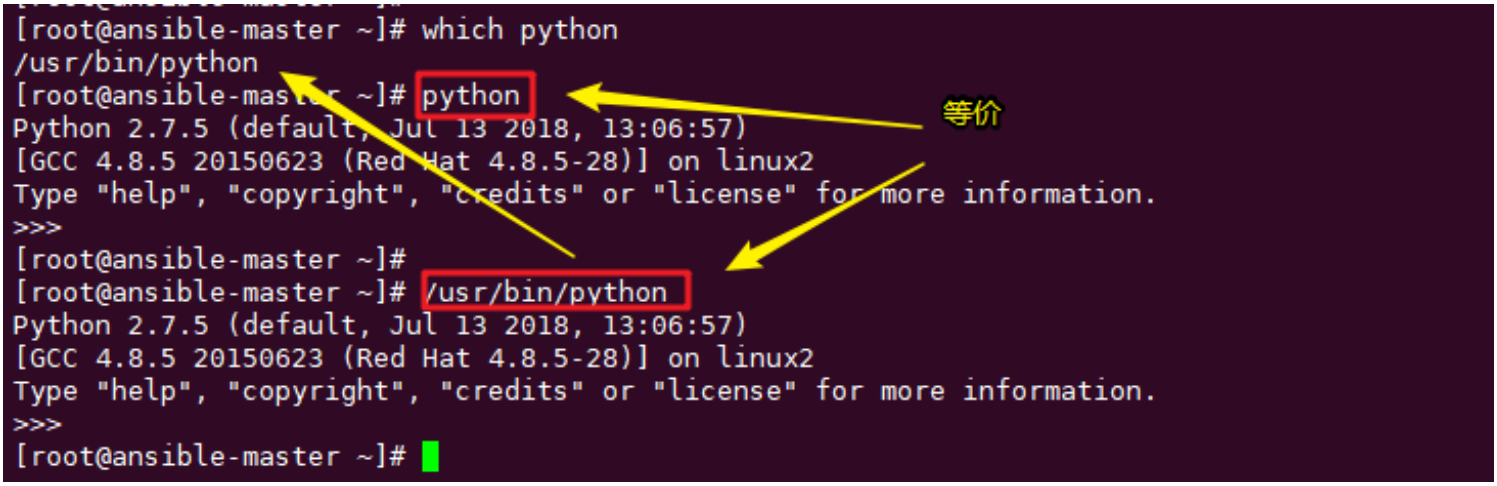
```
#!/usr/bin/python
```

或者这样

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

这两者有什么区别呢?

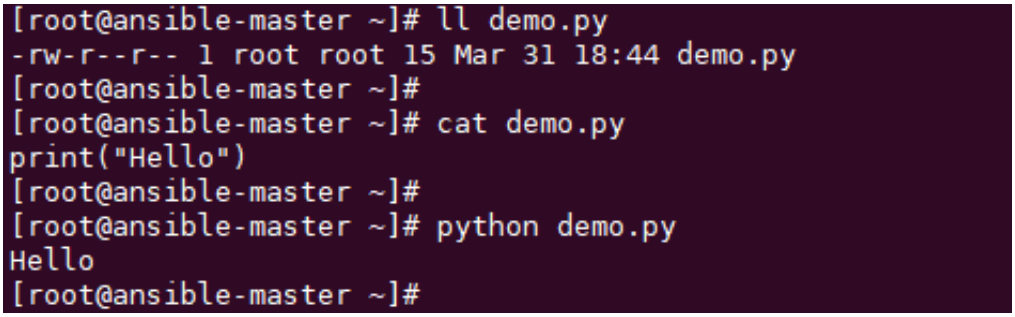
稍微接触过 linux 的人都知道 `/usr/bin/python` 就是我们执行 `python` 进入console 模式里的 `python`



而当你在可执行文件头里使用 `#!` + `/usr/bin/python` ，意思就是说你得用哪个软件（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` ，



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` 。

> 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。

1.16 dict() 与 {} 生成空字典有什么区别?

在初始化一个空字典时，有的人会写 dict()，而有的人会写成 {}

很多人会想当然的认为二者是等同的，但实际情况却不是这样的。

在运行效率上，{} 会比 dict() 快三倍左右。

使用 timeit 模块，可以轻松测出这个结果

```
$ python -m timeit -n 1000000 -r 5 -v "dict()"
raw times: 0.0996 0.0975 0.0969 0.0969 0.0994
1000000 loops, best of 5: 0.0969 usec per loop
$
```

```
$ python -m timeit -n 1000000 -r 5 -v "{}"
raw times: 0.0305 0.0283 0.0272 0.03 0.0317
1000000 loops, best of 5: 0.0272 usec per loop
```

那为什么会这样呢?

探究这个过程，可以使用 dis 模块

当使用 {} 时

```
$ cat demo.py
{}
$
$ python -m dis demo.py
  1           0 BUILD_MAP                0
              2 POP_TOP
              4 LOAD_CONST               0 (None)
              6 RETURN_VALUE
```

当使用 dict() 时：

```
$ cat demo.py
dict()
$
$ python -m dis demo.py
  1           0 LOAD_NAME                0 (dict)
              2 CALL_FUNCTION            0
              4 POP_TOP
              6 LOAD_CONST               0 (None)
              8 RETURN_VALUE
```

可以发现使用 dict()，会多了个调用函数的过程，而这个过程会有进出栈的操作，相对更加耗时。

> 作者：王炳明，微信公众号《Python编程时光》。

1.17 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

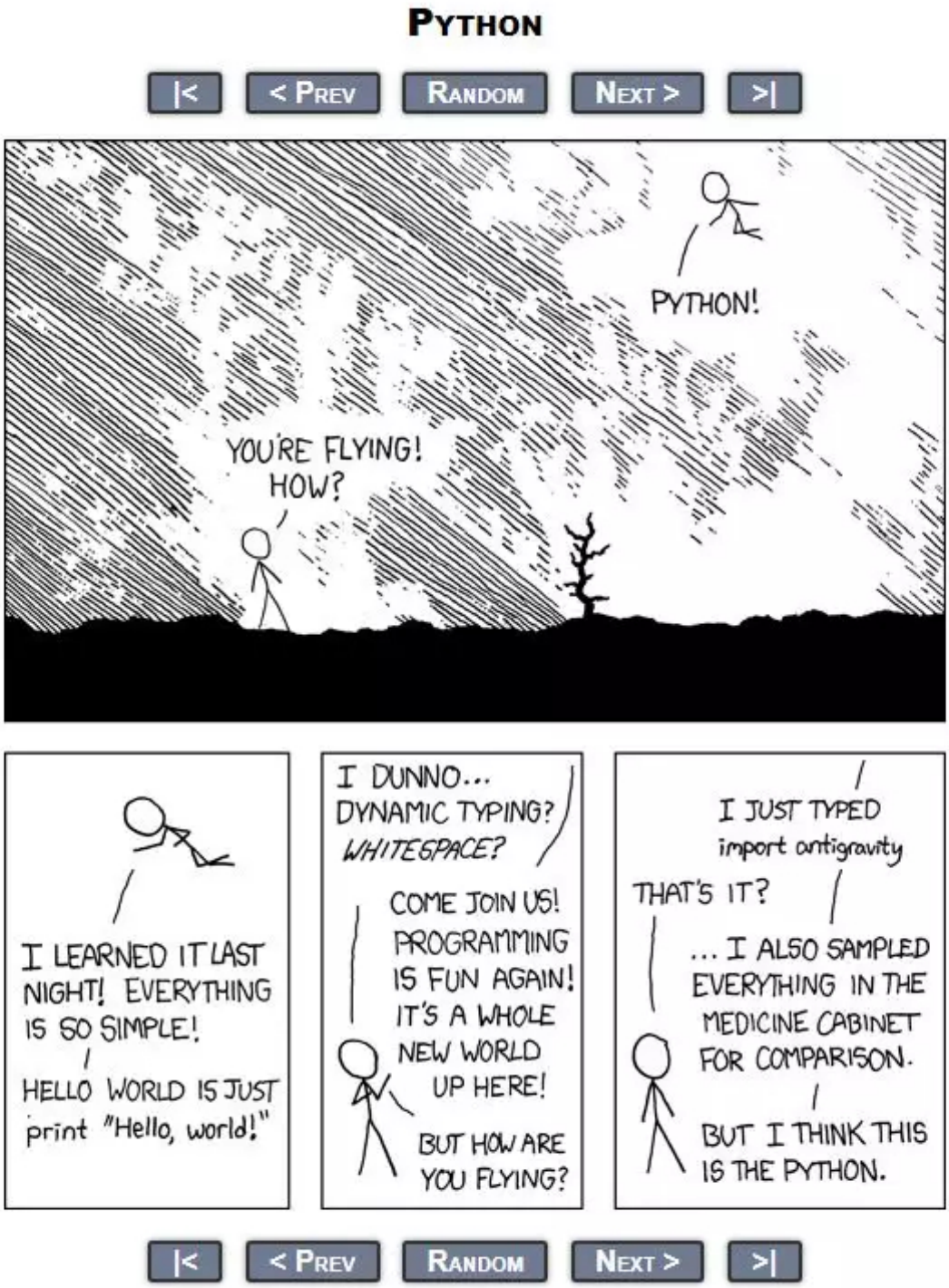
```
>>> import this

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。



1.18 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。*

> 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。

1.19 return不一定都是函数的终点

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 reutrn ，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。

1.20 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?

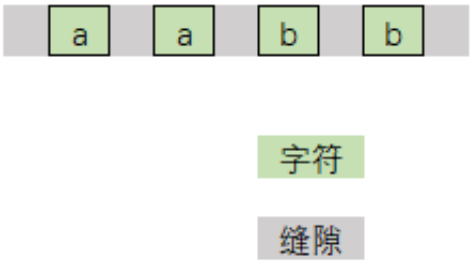
```
>>> "aabb".count("")
5
```

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



理解了这个“**缝隙**”的概念后，以下这些就好理解了。

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

> 作者：王炳明，微信公众号《Python编程时光》。

1.21 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。

1.22 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

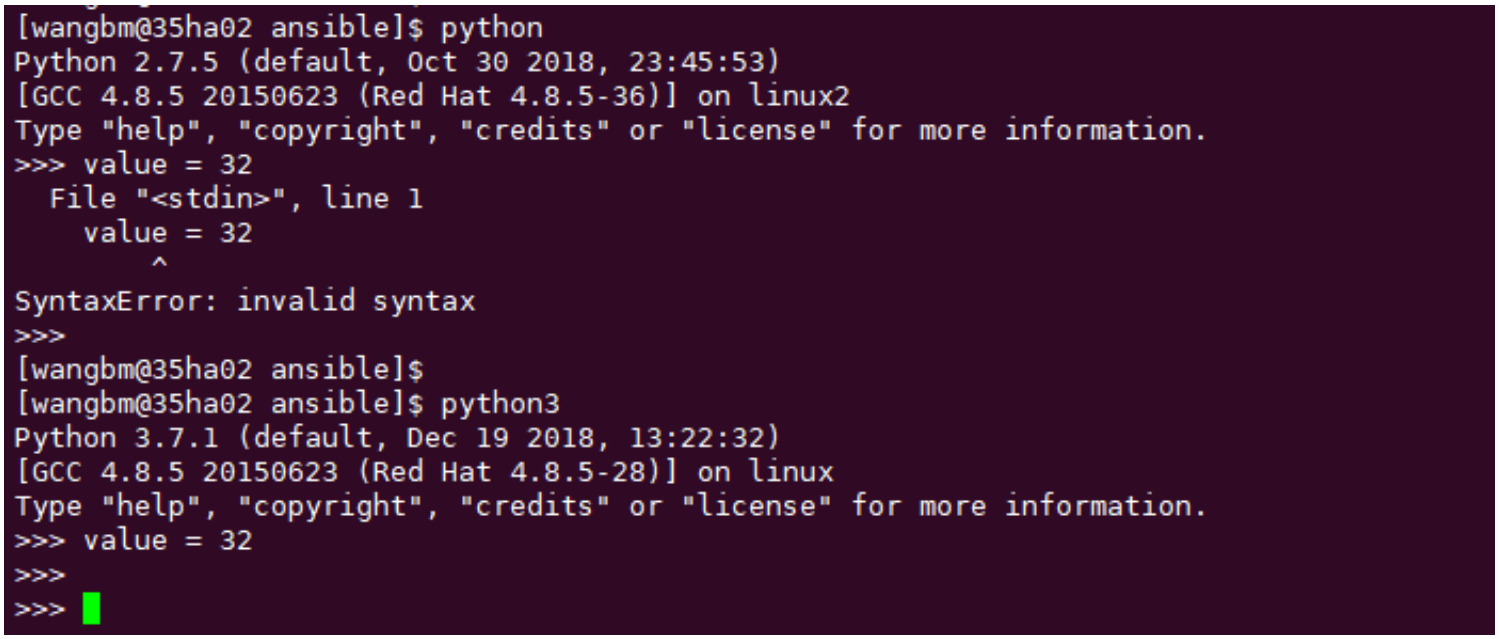
```
>>> value = 32
  File "<stdin>", line 1
    value = 32
        ^
```

```
SyntaxError: invalid syntax
```

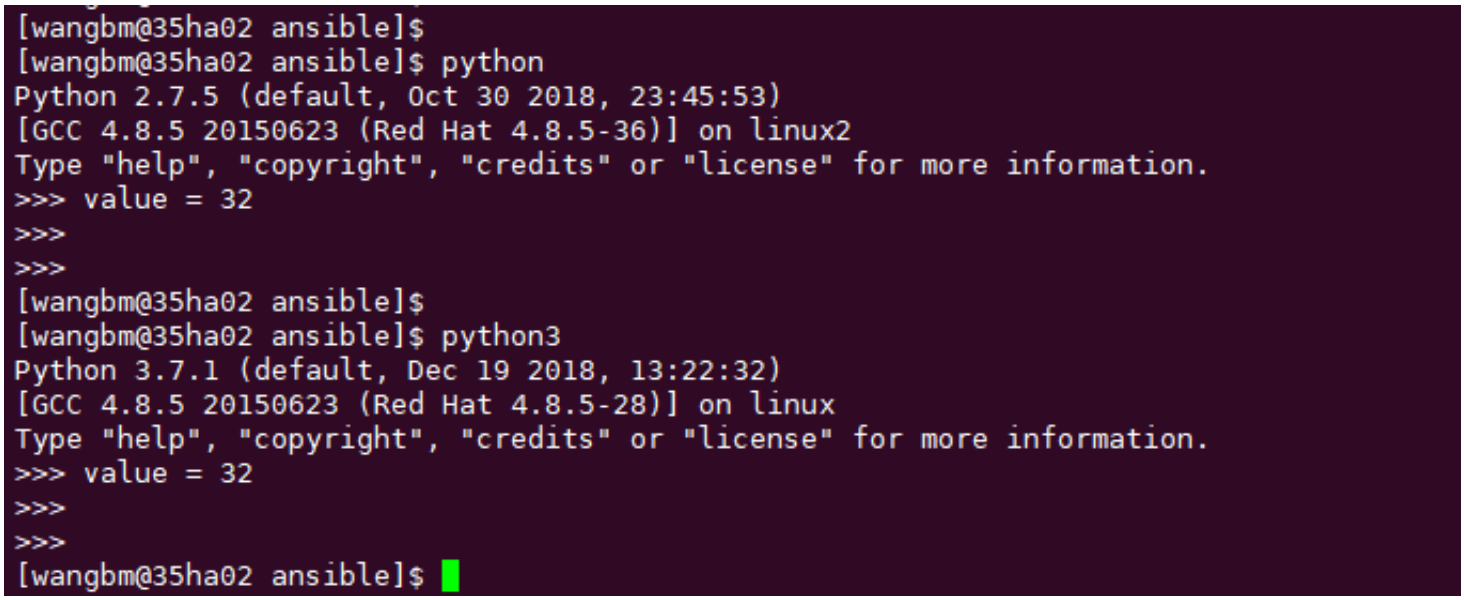
在Python 3.x 中

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

什么？没有截图你不信?



如果你在自己的电脑上尝试一下，结果可能是这样的



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 `e` 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
>>> 'е' == '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 `е`，到时候你就哭去吧，肉眼根本看不出来嘛。

> 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。

1.23 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。

```
>>> 100000000 < ""
True
>>> 100000000 < "hello"
True
```

但在 Python3 中，却不行。

```
>>> 100000000 < ""
TypeError: '<' not supported between instances of 'int' and 'str'
```

1.24 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:]
[]
```

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。

1.25 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 'aaaaaaaaaaaaaaaaaaaa'
True
>>> 'a' * 21 is 'aaaaaaaaaaaaaaaaaaaaa'
False

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

1.26 x 与 +x 等价吗?

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})
```

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。

1.27 += 不等同于=+

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]
```


1.28 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
```

> 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。

1.29 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 还没有被还没被定义。

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。

1.30 break /continue 和 上下文管理器哪个优先级高?

众所周知，在循环体中（无论是 for 还是 while），continue 会用来跳入下一个循环，而 break 则用来跳出某个循环体。

同时我们又知道：在上下文管理器中，被包裹的程序主体代码结束会运行上下文管理器中的一段代码（通常是资源的释放）。

但如果把上下文管理器放在一个循环体中，而在这个上下文管理器中执行了 break ，是否会直接跳出循环呢?

换句话说，上下文管理器与 break/continue 这两个规则哪一个优先级会更高一些?

这个问题其实不难，只要做一下试验都能轻易地得出答案，难就难在很多对这个答案都是半猜半疑，无法肯定的回答。

试验代码如下:

```
import time
import contextlib

@contextlib.contextmanager
def runtime(value):
    time.sleep(1)
```

```
    print("start: a = " + str(value))
    yield
    print("end: a = " + str(value))


a = 0
while True:
    a+=1
    with runtime(a):
        if a % 2 == 0:
            break
```

从输出的结果来看，当 a = 2 时执行了 break ，此时的并不会直接跳出循环，依然要运行上下文管理器里清理释放资源的代码（示例中，我使用 print 来替代）。

```
start: a = 1
end: a = 1
start: a = 2
end: a = 2
```

另外还有几个与此类似的问题，我这里也直接给出答案，不再细说了

1. continue 与 break 一样，如果先遇到上下文管理器会先进行资源的释放
2. 上面只举例了 while 循环体，而 for 循环也是同样的。

第二章：魔法命令行

2.1 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，就是在交互式模式下的应用。

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。

> 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。

2.2 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呢?

你可能会想到这种，但这本质上与上面并无区别

```
[wangbm@localhost ~]$ python -c "print('\n'.join(__import__('sys').path))"

/usr/lib/python2.7/site-packages/pip-18.1-py2.7.egg
/usr/lib/python2.7/site-packages/redis-3.0.1-py2.7.egg
/usr/lib64/python27.zip
/usr/lib64/python2.7
/usr/lib64/python2.7/plat-linux2
/usr/lib64/python2.7/lib-tk
/usr/lib64/python2.7/lib-old
```

```
/usr/lib64/python2.7/lib-dynload
/home/wangbm/.local/lib/python2.7/site-packages
/usr/lib64/python2.7/site-packages
/usr/lib64/python2.7/site-packages/gtk-2.0
/usr/lib/python2.7/site-packages
```

这里我要介绍的是比上面两种都方便的多的方法，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。

> 作者：王炳明，微信公众号《Python编程时光》。

2.3 使用 json.tool 来格式化 JSON

假设现在你需要查看你机器上的json文件，而这个文件没有经过任何的美化，阅读起来是非常困难的。

```
$ cat demo.json
{"_id":"5f12d319624e57e27d1291fe","index":0,"guid":"4e482708-c6aa-4ef9-a45e-d5ce2c72c68d","isActive":false,"balance":"$2,954.93","picture":"http://placehold.it/32x32","age":36,"eyeColor":"green","name":"MasseySaunders","gender":"male","company":"TALAE","email":"masseysaunders@talae.com","phone":"+1(853)508-3237","address":"246IndianaPlace,Glenbrook,Iowa,3896","about":"Velitmagnanostrudexcepteurduisextemporirurefugiataliquasunt.Excepteurvelitquiseuinexinoccaecatoccaecatveliteuet.Commodonisialiquipirureminimconsequatminimconsecteturipsumsitex.\r\n",
```

```
"registered":"2017-02-06T06:42:20-08:00","latitude":-10.269827,"longitude":-103
.12419,"tags":["laborum","excepteur","veniam","reprehenderit","voluptate","labo
rum","in"],"friends":[{"id":0,"name":"DorotheaShields"},{"id":1,"name":"AnnaRos
ales"},{"id":2,"name":"GravesBryant"}],"greeting":"Hello,MasseySaunders!Youhave
8unreadmessages.","favoriteFruit":"apple"}
```


这时候你就可以使用 python 的命令行来直接美化。

```
$ python -m json.tool demo.json
{
    "_id": "5f12d319624e57e27d1291fe",
    "about": "Velitmagnanostrudexcepteurduisextemporirurefugiataliquasunt.Excep
teurvelitquiseuinexinoccaecatoccaecatveliteuet.Commodonisialiquipirureminimcons
equatminimconsecteturipsumsitex.\r\n",
    "address": "246IndianaPlace,Glenbrook,Iowa,3896",
    "age": 36,
    "balance": "$2,954.93",
    "company": "TALAE",
    "email": "masseysaunders@talae.com",
    "eyeColor": "green",
    "favoriteFruit": "apple",
    "friends": [
        {
            "id": 0,
            "name": "DorotheaShields"
        },
        {
            "id": 1,
            "name": "AnnaRosales"
        },
        {
            "id": 2,
            "name": "GravesBryant"
        }
    ],
    "gender": "male",
    "greeting": "Hello,MasseySaunders!Youhave8unreadmessages.",
    "guid": "4e482708-c6aa-4ef9-a45e-d5ce2c72c68d",
    "index": 0,
    "isActive": false,
    "latitude": -10.269827,
    "longitude": -103.12419,
    "name": "MasseySaunders",
    "phone": "+1(853)508-3237",
    "picture": "http://placehold.it/32x32",
```

```
    "registered": "2017-02-06T06:42:20-08:00",
    "tags": [
        "laborum",
        "excepteur",
        "veniam",
        "reprehenderit",
        "voluptate",
        "laborum",
        "in"
    ]
}
```


2.4 命令行式执行 Python 代码

有时候你只是想验证一小段 Python 代码是否可用时，通常有两种方法

1. 输入 python 回车，进入 console 模式，然后敲入代码进行验证
2. 将你的代码写入 demo.py 脚本中，然后使用 python demo.py 验证

其实还有一种更简单的方法，比如我要计算一个字符串的md5

```
$ python -c "import hashlib;print(hashlib.md5('hello').hexdigest())"
5d41402abc4b2a76b9719d911017c592
```

只要加 -c 参数，就可以输入你的 Python 代码了。

2.5 用调试模式执行脚本

当你使用 pdb 进行脚本的调试时，你可能会先在目标代码处输入

`import pdb;pdb.set_trace()` 来设置断点。

除此之外，还有一种方法，就是使用 `-m pdb`

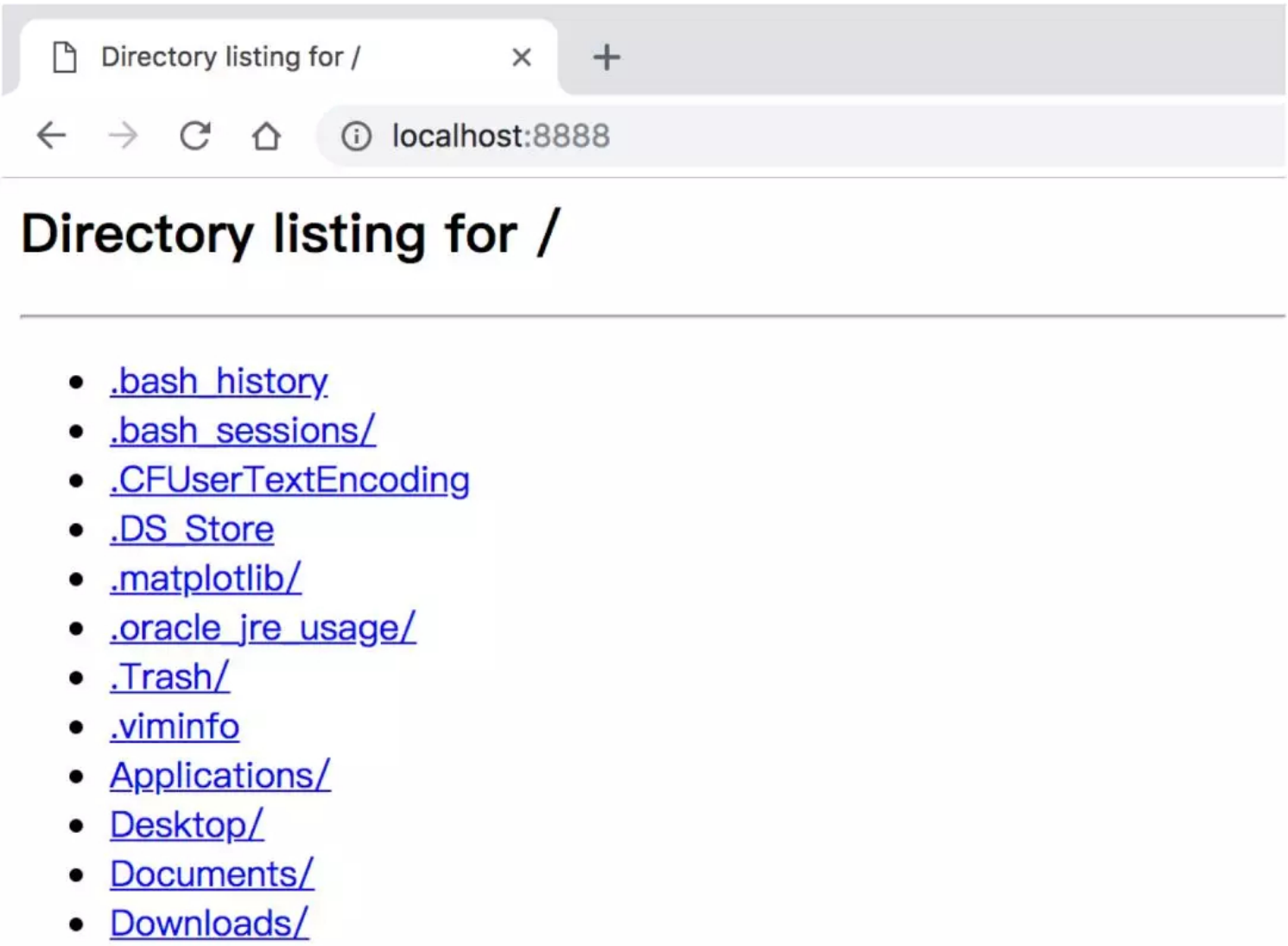
```
$ python -m pdb demo.py
> /Users/MING/demo.py(1)<module>()
-> import sys
(Pdb)
```


2.6 如何快速搭建 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
```



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。

2.7 快速构建 HTML 帮助文档

当你不知道一个内置模块如何使用时，会怎么做呢?

百度？Google?

其实完全没必要，这里教你一个离线学习 Python 模块的方法。

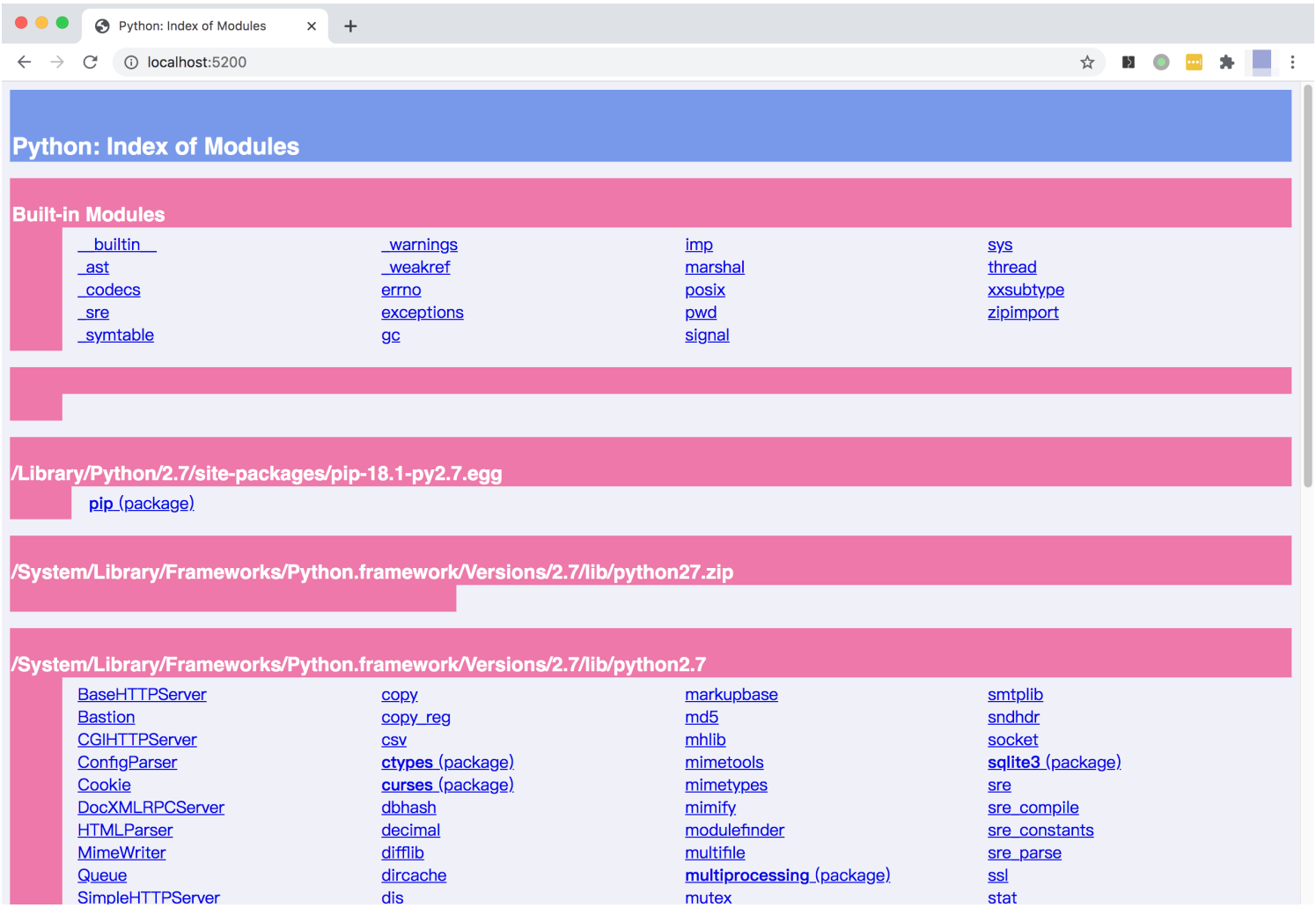
是的，你没有听错。

就算没有外网网络也能学习 Python 模块.

你只要在命令行下输入 `python -m pydoc -p xxx` 命令即可开启一个 HTTP 服务，xxx 为端口，你可以自己指定。

```
$ python -m pydoc -p 5200
pydoc server ready at http://localhost:5200/
```

帮助文档的效果如下



2.8 最正确且优雅的装包方法

当你使用 pip 来安装第三方的模块时，通常会使用这样的命令

```
$ pip install requests
```

此时如果你的环境中有 Python2 也有 Python 3，那你使用这条命令安装的包是安装 Python2 呢？还是安装到 Python 3 呢？

就算你的环境上没有安装 Python2，那也有可能存在着多个版本的 Python 吧？比如安装了 Python3.8，也安装了 Python3.9，那你安装包时就会很困惑，我到底把包安装在了哪里？

但若你使用这样的命令去安装，就没有了这样的烦恼了

```
# 在 python2 中安装
$ python -m pip install requests

# 在 python3 中安装
$ python3 -m pip install requests

# 在 python3.8 中安装
$ python3.8 -m pip install requests

# 在 python3.9 中安装
$ python3.9 -m pip install requests
```


2.9 往 Python Shell 中传入参数

往一个 Python 脚本传入参数，是一件非常简单的事情。

比如这样：

```
$ python demo.py arg1 arg2
```

我在验证一些简单的 Python 代码时，喜欢使用 Python Shell 。

那有没有办法在使用 Python Shell 时，向上面传递参数一样，传入参数呢？

经过我的摸索，终于找到了方法，具体方法如下：

```
(PythonCodingTime)
~/Github/PythonCodingTime on ?master! 19:51:31
$ Python - "欢迎关注公众号：Python编程时光"
Python 3.6.7 (v3.6.7:6ec5cf24b7, Oct 20 2018, 03:02:14)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.argv[0]
'-'
>>> sys.argv[1]
'欢迎关注公众号：Python编程时光'
>>>
```

2.10 让脚本报错后立即进入调试模式

当你在使用 `python xxx.py` 这样的方法，执行 Python 脚本时，若因为代码 bug 导致异常未捕获，那整个程序便会终止退出。

这个时候，我们通常会去排查是什么原因导致的程序崩溃。

大家都知道，排查问题的思路，第一步肯定是去查看日志，若这个 bug 隐藏的比较深，只有在特定场景下才会现身，那么还需要开发者，复现这个 bug，方能优化代码。

复现有时候很难，有时候虽然简单，但是要伪造各种数据，相当麻烦。

如果有一种方法能在程序崩溃后，立马进入调试模式该有多好啊?

明哥都这么问了，那肯定是带着解决方案来的。

只要你在执行脚本行，加上 `-i` 参数，即可在脚本执行完毕后进入 Python Shell 模式，方便你进行调试。

具体演示如下：

需要注意的是：脚本执行完毕，有两种情况：

1. 正常退出
2. 异常退出

这两种都会进入 Python Shell，如果脚本并无异常，最终也会进入 Python Shell 模式，需要你手动退出

2.11 极简模式执行 Python Shell

在终端输入 Python 就会进入 Python Shell 。

方便是挺方便，就是有点说不出的难受，谁能告诉我，为什么要多出这么大一段无关的内容。

```
(PythonCodingTime)
~/Github/PythonCodingTime on ?master! 20:27:13
$ Python
Python 3.6.7 (v3.6.7:6ec5cf24b7, Oct 20 2018, 03:02:14)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

这有点像，你上爱某艺看视频吧，都要先看个 90 秒的广告。

如果你和我一样不喜欢这种 『牛皮癣』，那么可以加个 `-q` 参数，静默进入 Python Shell，就像下面这样子，开启了极简模式，舒服多了。

2.12 在执行任意代码前自动念一段平安经

最近的"平安经"可谓是引起了不小的风波啊。

作为一个正儿八经的程序员，最害怕的就是自己的代码上线出现各种各样的 BUG。

为此明哥就研究了一下，如何在你执行任意 Python 代码前，让 Python 解释器自动念上一段平安经，保佑代码不出 BUG 。

没想到还真被我研究出来了

做好心理准备了嘛?

我要开始作妖了，噢不，是开始念经了。

```
~ on ?master! 22:15:20
$ cat demo.py
print("everything is ok")

~ on ?master! 22:16:45
$ python3 demo.py


                  _ooOoo_
                 o8888888o    _____________________________________
                 88" . "88    |  爬虫工程师平安  后端工程师平安  |
                 (| -_- |)    |  数据分析师平安  自动化运维平安  |
                  0\ = /0     <_____________________________________|
               ___/`---'\____
             .   ' \\| |// `.
              / \\||| : |||// \
            / _||||| -:- |||||- \
              | | \\\ - /// | |
            | \_| ''\---/'' | |
             \ .-\__ `-` ___/-. /
          ___`. .' /--.--\ `. . __
       ."" '< `.___\_<|>_/___.' >'"".
      | | : `- \`.;`\ _ /`;.`/ - ` : | |
        \ \ `-. \_ __\ /__ _/ .-` / /
 ======`-.____`-.___\_____/___.-`____.-'======
                    `=---='

 .............................................
          佛祖保佑            永无BUG

everything is ok
```

感谢佛祖保佑，Everything is ok，No bugs in the code.

你一定很想知道这是如何的吧?

如果你对 Linux 比较熟悉，就会知道，当你在使用 SSH 远程登陆 Linux 服务器的时候? 会读取

`.bash_profile` 文件加载一些环境变量。

`.bash_profile` 你可以视其为一个 shell 脚本，可以在这里写一些 shell 代码达到你的定制化需求。

而在 Python 中，也有类似 `.bash_profile` 的文件，这个文件一般情况下是不存在的。

我们需要新建一个用户环境目录，这个目录比较长，不需要你死记硬背，使用 site 模块的方法就可以获取，然后使用 `mkdir -p` 命令创建它。

```
~ on ?master! 22:06:47
$ python3
Python 3.9.0a4 (v3.9.0a4:6e02691f30, Feb 25 2020, 18:14:13)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import site
>>> site.getusersitepackages()
'/Users/MING/Library/Python/3.9/lib/python/site-packages'
>>> quit()

~ on ?master! 22:07:11
$ mkdir -p /Users/MING/Library/Python/3.9/lib/python/site-packages

~ on ?master! 22:07:37
$ cd /Users/MING/Library/Python/3.9/lib/python/site-packages
```

在这个目录下，新建一个 `usercustomize.py` 文件，注意名字必须是这个，换成其他的可就识别不到啦。

这个 `usercustomize.py` 的内容如下（明哥注：佛祖只保佑几个 Python 的主要应用方向，毕竟咱是 Python 攻城狮嘛...）

```
msg=r"""

                  _ooOoo_
                 o8888888o    _____________________________________
                 88" . "88    |  爬虫工程师平安  后端工程师平安  |
                 (| -_- |)    |  数据分析师平安  自动化运维平安  |
                  0\ = /0     <_____________________________________|
               ____/`---'\____
             .   ' \\| |// `.
              / \\||| : |||// \
            / _||||| -:- |||||- \
              | | \\\ - /// | |
            | \_| ''\---/'' | |
             \ .-\__ `-` ___/-. /
          ___`. .' /--.--\ `. . __
       ."" '< `.___\_<|>_/___.' >'"".
      | | : `- \`.;`\ _ /`;.`/ - ` : | |
        \ \ `-. \_ __\ /__ _/ .-` / /
 ======`-.____`-.___\_____/___.-`____.-'======
                    `=---='

 .............................................
         佛祖保佑             永无BUG
"""
print(msg)
```

这个文件我放在了我的 github 上，你可以点此前往获取。

一切都完成后，无论你是使用 `python xxx.py` 执行脚本

```
~ on ?master! 22:15:20
$ cat demo.py
print("everything is ok")

~ on ?master! 22:16:45
$ python3 demo.py
```


```
everything is ok
```

还是使用 `python` 进入 Python Shell ， 都会先念一下平安经保平安。

2.13 启动 Python Shell 前自动执行某脚本

上面我们介绍了一种，只要运行解释器就会自动触发执行 Python 脚本的方法。

除此之外，可还有其他方法呢?

当然是有，只不过相对来说，会麻烦一点了。

先来看一下效果，在 `~/Library/Python/3.9/lib/python/site-packages` 目录下并没有 `usercustomize.py` 文件，但是在执行 python 进入 Python Shell 模式后，还是会打印平安经。

这是如何做到的呢?

很简单，只要做两件事

第一件事，在任意你喜欢的目录下，新建 一个Python 脚本，名字也随意，比如我叫 `startup.py`，内容还是和上面一样

```
msg=r"""

                  _ooOoo_
                 o8888888o    ___________________________________
                 88" . "88    |  爬虫工程师平安  后端工程师平安  |
                 (| -_- |)    |  数据分析师平安  自动化运维平安  |
                  0\ = /0     <___________________________________|
               ____/`---'\____
             .   ' \\| |// `.
              / \\||| : ||// \
            / _||||| -:- |||||- \
              | | \\\ - /// | |
            | \_| ''\---/'' | |
             \ .-\__ `-` ___/-. /
          ___`. .' /--.--\ `. . __
       ."" '< `.___\_<|>_/___.' >'"".
      | | : `- \`.;`\ _ /`;.`/ - ` : | |
        \ \ `-. \_ __\ /__ _/ .-` / /
 ======`-.____`-.___\_____/___.-`____.-'======
                    `=---='

 .............................................
         佛祖保佑             永无BUG
"""
print(msg)
```

第二件事，设置一个环境变量 PYTHONSTARTUP，指向你的脚本路径

```
$ export PYTHONSTARTUP=/Users/MING/startup.py
```

这样就可以了。

但是这种方法只适用于 Python Shell ， 只不适合 Python 执行脚本的方法。

```
~ on ?master! 23:02:15
$ python3 demo.py
everything is ok
```

如果要在脚本中实现这种效果，我目前想到最粗糙我笨拙的方法了 -- 手动加载执行

```
~ on master! 23:04:42
$ cat demo.py
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)

print("everything is ok")

~ on master! 23:04:44
$ python3 demo.py


                  _ooOoo_
                 o8888888o    ___________________________________
                 88" . "88    |  爬虫工程师平安  后端工程师平安  |
                 (| -_- |)    |  数据分析师平安  自动化运维平安  |
                  0\ = /0     <___________________________________|
               ____/`---'\____
             .   ' \\| |// `.
              / \\||| : |||// \
            / _||||| -:- |||||- \
              | | \\\ - /// | |
            | \_| ''\---/'' | |
             \ .-\__ `-` ___/-. /
          ___`. .' /--.--\ `. . __
       ."" '< `.___\_<|>_/___.' >'"".
      | | : `- \`.;`\ _ /`;.`/ - ` : | |
        \ \ `-. \_ __\ /__ _/ .-` / /
======`-.____`-.___\_____/___.-`____.-'======
                   `=---='

 ..........................................
         佛祖保佑             永无BUG

everything is ok
```

> 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。

第三章：炫技魔法操作

3.1 八种连接列表的方式

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 Pythonnic 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 这个方法来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]
>>
```


3.2 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 Pythonnic 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。

3.3 花式导包的八种方法

1. 直接 import

人尽皆知的方法，直接导入即可

```
>>> import os
>>> os.getcwd()
'/home/wangbm'
```

与此类似的还有，不再细讲

```
import ...
import ... as ...
from ... import ...
from ... import ... as ...
```

一般情况下，使用 `import` 语句导入模块已经够用的。

但是在一些特殊场景中，可能还需要其他的导入方式。

下面我会一一地给你介绍。

2. 使用 __import__

`__import__` 函数可用于导入模块，import 语句也会调用函数。其定义为：

```
__import__(name[, globals[, locals[, fromlist[, level]]]])
```

参数介绍：

- name (required): 被加载 module 的名称
- globals (optional): 包含全局变量的字典，该选项很少使用，采用默认值 global()
- locals (optional): 包含局部变量的字典，内部标准实现未用到该变量，采用默认值 – local()
- fromlist (Optional): 被导入的 submodule 名称
- level (Optional): 导入路径选项，Python 2 中默认为 –1，表示同时支持 absolute import 和 relative import。Python 3 中默认为 0，表示仅支持 absolute import。如果大于 0，则表示相对导入的父目录的级数，即 1 类似于 '.'，2 类似于 '..'。

使用示例如下：

```
>>> os = __import__('os')
>>> os.getcwd()
'/home/wangbm'
```

如果要实现 `import xx as yy` 的效果，只要修改左值即可

如下示例，等价于 `import os as myos`：

```
>>> myos = __import__('os')
>>> myos.getcwd()
'/home/wangbm'
```

上面说过的 `__import__` 是一个内建函数，既然是内建函数的话，那么这个内建函数必将存在于 `__buildins__` 中，因此我们还可以这样导入 os 的模块：

```
>>> __builtins__.__dict__['__import__']('os').getcwd()
'/home/wangbm'
```

3. 使用 importlib 模块

importlib 是 Python 中的一个标准库，importlib 能提供的功能非常全面。

它的简单示例：

```
>>> import importlib
>>> myos=importlib.import_module("os")
>>> myos.getcwd()
'/home/wangbm'
```

如果要实现 `import xx as yy` 效果，可以这样

```
>>> import importlib
>>>
>>> myos = importlib.import_module("os")
>>> myos.getcwd()
'/home/wangbm'
```

4. 使用 imp 模块

`imp` 模块提供了一些 import 语句内部实现的接口。例如模块查找（find_module）、模块加载（load_module）等等（模块的导入过程会包含模块查找、加载、缓存等步骤）。可以用该模块来简单实现内建的 `__import__` 函数功能：

```
>>> import imp
>>> file, pathname, desc = imp.find_module('os')
>>> myos = imp.load_module('sep', file, pathname, desc)
>>> myos
<module 'sep' from '/usr/lib64/python2.7/os.pyc'>
>>> myos.getcwd()
'/home/wangbm'
```

从 python 3 开始，内建的 reload 函数被移到了 imp 模块中。而从 Python 3.4 开始，imp 模块被否决，不再建议使用，其包含的功能被移到了 importlib 模块下。即从 Python 3.4 开始，importlib 模块是之前 imp 模块和 importlib 模块的合集。

5. 使用 execfile

在 Python 2 中有一个 execfile 函数，利用它可以用来执行一个文件。

语法如下：

```
execfile(filename[, globals[, locals]])
```

参数有这么几个：

- filename：文件名。
- globals：变量作用域，全局命名空间，如果被提供，则必须是一个字典对象。
- locals：变量作用域，局部命名空间，如果被提供，可以是任何映射对象。

```
>>> execfile("/usr/lib64/python2.7/os.py")
>>>
>>> getcwd()
'/home/wangbm'
```

6. 使用 exec 执行

`execfile` 只能在 Python2 中使用，Python 3.x 里已经删除了这个函数。

但是原理值得借鉴，你可以使用 open ... read 读取文件内容，然后再用 exec 去执行模块。

示例如下：

```
>>> with open("/usr/lib64/python2.7/os.py", "r") as f:
...     exec(f.read())
```

```
...
>>> getcwd()
'/home/wangbm'
```

7. import_from_github_com

有一个包叫做 **import_from_github_com**，从名字上很容易得知，它是一个可以从 github 下载安装并导入的包。为了使用它，你需要做的就是按照如下命令使用pip 先安装它。

```
$ python3 -m pip install import_from_github_com
```

这个包使用了PEP 302中新的引入钩子，允许你可以从github上引入包。这个包实际做的就是安装这个包并将它添加到本地。你需要 Python 3.2 或者更高的版本，并且 git 和 pip 都已经安装才能使用这个包。

pip 要保证是较新版本，如果不是请执行如下命令进行升级。

```
$ python3 -m pip install --upgrade pip
```

确保环境 ok 后，你就可以在 Python shell 中使用 import_from_github_com

示例如下

```
>>> from github_com.zzzeek import sqlalchemy
Collecting git+https://github.com/zzzeek/sqlalchemy
Cloning https://github.com/zzzeek/sqlalchemy to /tmp/pip-acfv7t06-build
Installing collected packages: SQLAlchemy
Running setup.py install for SQLAlchemy ... done
Successfully installed SQLAlchemy-1.1.0b1.dev0
>>> locals()
{'__builtins__': <module 'builtins' (built-in)>, '__spec__': None,
'__package__': None, '__doc__': None, '__name__': '__main__',
'sqlalchemy': <module 'sqlalchemy' from '/usr/local/lib/python3.5/site-packages
/\
sqlalchemy/__init__.py'>,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>}
>>>
```

看了 import_from_github_com的源码后，你会注意到它并没有使用importlib。实际上，它的原理就是使用 pip 来安装那些没有安装的包，然后使用Python的 `__import__()` 函数来引入新安装的模块。

8、远程导入模块

我在这篇文章里（深入探讨 Python 的 import 机制：实现远程导入模块），深入剖析了导入模块的内部原理，并在最后手动实现了从远程服务器上读取模块内容，并在本地成功将模块导入的导入器。

具体内容非常的多，你可以点击这个链接进行深入学习。

示例代码如下：

```
# 新建一个 py 文件（my_importer.py），内容如下
import sys
import importlib
import urllib.request as urllib2

class UrlMetaFinder(importlib.abc.MetaPathFinder):
    def __init__(self, baseurl):
        self._baseurl = baseurl


    def find_module(self, fullname, path=None):
        if path is None:
            baseurl = self._baseurl
        else:
            # 不是原定义的url就直接返回不存在
            if not path.startswith(self._baseurl):
                return None
            baseurl = path

        try:
            loader = UrlMetaLoader(baseurl)
            return loader
        except Exception:
            return None

class UrlMetaLoader(importlib.abc.SourceLoader):
    def __init__(self, baseurl):
        self.baseurl = baseurl

    def get_code(self, fullname):
        f = urllib2.urlopen(self.get_filename(fullname))
        return f.read()

    def get_data(self):
        pass
```

```
    def get_filename(self, fullname):
        return self.baseurl + fullname + '.py'

def install_meta(address):
    finder = UrlMetaFinder(address)
    sys.meta_path.append(finder)
```

并且在远程服务器上开启 http 服务（为了方便，我仅在本地进行演示），并且手动编辑一个名为 my_info 的 python 文件，如果后面导入成功会打印 `ok` 。

```
$ mkdir httpserver && cd httpserver
$ cat>my_info.py<EOF
name='wangbm'
print('ok')
EOF
$ cat my_info.py
name='wangbm'
print('ok')
$
$ python3 -m http.server 12800
Serving HTTP on 0.0.0.0 port 12800 (http://0.0.0.0:12800/) ...
...
```

一切准备好，验证开始。

```
>>> from my_importer import install_meta
>>> install_meta('http://localhost:12800/') # 往 sys.meta_path 注册 finder
>>> import my_info  # 打印ok，说明导入成功
ok
>>> my_info.name  # 验证可以取得到变量
'wangbm'
```

好了，8 种方法都给大家介绍完毕，对于普通开发者来说，其实只要掌握 import 这种方法足够了，而对于那些想要自己开发框架的人来说，深入学习 `__import__` 以及 importlib 是非常有必要的。

3.4 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。

3.5 判断是否包含子串的七种方法

1、使用 in 和 not in

`in` 和 `not in` 在 Python 中是很常用的关键字，我们将它们归类为 `成员运算符` 。

使用这两个成员运算符，可以很让我们很直观清晰的判断一个对象是否在另一个对象中，示例如下：

```
>>> "llo" in "hello, python"
True
>>>
>>> "lol" in "hello, python"
False
```

2、使用 find 方法

使用 字符串 对象的 find 方法，如果有找到子串，就可以返回指定子串在字符串中的出现位置，如果没有找到，就返回 `-1`

```
>>> "hello, python".find("llo") != -1
True
>>> "hello, python".find("lol") != -1
False
>>
```

3、使用 index 方法

字符串对象有一个 index 方法，可以返回指定子串在该字符串中第一次出现的索引，如果没有找到会抛出异常，因此使用时需要注意捕获。

```
def is_in(full_str, sub_str):
    try:
        full_str.index(sub_str)
        return True
    except ValueError:
        return False

print(is_in("hello, python", "llo"))  # True
print(is_in("hello, python", "lol"))  # False
```

4、使用 count 方法

利用和 index 这种曲线救国的思路，同样我们可以使用 count 的方法来判断。

只要判断结果大于 0 就说明子串存在于字符串中。

```
def is_in(full_str, sub_str):
    return full_str.count(sub_str) > 0

print(is_in("hello, python", "llo"))  # True
print(is_in("hello, python", "lol"))  # False
```

5、通过魔法方法

在第一种方法中，我们使用 in 和 not in 判断一个子串是否存在于另一个字符中，实际上当你使用 in 和 not in 时，Python 解释器会先去检查该对象是否有 `__contains__` 魔法方法。

若有就执行它，若没有，Python 就自动会迭代整个序列，只要找到了需要的一项就返回 True 。

示例如下；

```
>>> "hello, python".__contains__("llo")
True
>>>
>>> "hello, python".__contains__("lol")
False
>>>
```

这个用法与使用 in 和 not in 没有区别，但不排除有人会特意写成这样来增加代码的理解难度。

6、借助 operator

operator模块是python中内置的操作符函数接口，它定义了一些算术和比较内置操作的函数。operator模块是用c实现的，所以执行速度比 python 代码快。

在 operator 中有一个方法 `contains` 可以很方便地判断子串是否在字符串中。

```
>>> import operator
>>>
>>> operator.contains("hello, python", "llo")
True
>>> operator.contains("hello, python", "lol")
False
>>>
```

7、使用正则匹配

说到查找功能，那正则绝对可以说是专业的工具，多复杂的查找规则，都能满足你。

对于判断字符串是否存在于另一个字符串中的这个需求，使用正则简直就是大材小用。

```
import re

def is_in(full_str, sub_str):
    if re.findall(sub_str, full_str):
        return True
    else:
        return False

print(is_in("hello, python", "llo"))  # True
print(is_in("hello, python", "lol"))  # False
```


3.6 海象运算符的三种用法

Python 版本发展非常快，如今最新的版本已经是 Pyhton 3.9，即便如此，有很多人甚至还停留在 3.6 或者 3.7，连 3.8 还没用上。

很多 Python 3.8 的特性还没来得及了解，就已经成为旧知识了，比如今天要说的海象运算符。

海象运算符是在 PEP 572 被提出的，直到 3.8 版本合入发布。

它的英文原名叫 `Assignment Expressions`，翻译过来也就是 `赋值表达式`，不过现在大家更普遍地称之为海象运算符，就是因为它长得真的太像海象了。

第一个用法：if/else

可能有朋友是第一次接触这个新特性，所以还是简单的介绍一下这个海象运算符有什么用?

在 Golang 中的条件语句可以直接在 if 中运算变量的获取后直接对这个变量进行判断，可以让你少写一行代码

```
import "fmt"

func main() {
    if age := 20;age > 18 {
        fmt.Println("已经成年了")
    }
}
```

若在 Python 3.8 之前，Python 必须得这样子写

```
age = 20
```

```
if age > 18:
    print("已经成年了")
```

但有了海象运算符之后，你可以和 Golang 一样（如果你没学过 Golang，那这里要注意，Golang 中的 `:=` 叫短变量声明，意思是声明并初始化，它和 Python 中的 `:=` 不是一个概念）

```
if (age:= 20) > 18:
    print("已经成年了")
```

第二个用法：while

在不使用 海象运算符之前，使用 while 循环来读取文件的时候，你也许会这么写

```
file = open("demo.txt", "r")
while True:
    line = file.readline()
    if not line:
        break
    print(line.strip())
```

但有了海象运算符之后，你可以这样

```
file = open("demo.txt", "r")
while (line := file.readline()):
    print(line.strip())
```

使用它替换以往的无限 while 循环写法更为惊艳

比如，实现一个需要命令行交互输入密码并检验的代码，你也许会这样子写

```
while True:
   p = input("Enter the password: ")
   if p == "youpassword":
      break
```

有了海象运算符之后，这样子写更为舒服

```
while (p := input("Enter the password: ")) != "youpassword":
   continue
```

第三个用法：推导式

这个系列的文章，几乎每篇都能看到推导式的身影，这一篇依旧如此。

在编码过程中，我很喜欢使用推导式，在简单的应用场景下，它简洁且不失高效。

如下这段代码中，我会使用列表推导式得出所有会员中过于肥胖的人的 bmi 指数

```
members = [
    {"name": "小五", "age": 23, "height": 1.75, "weight": 72},
    {"name": "小李", "age": 17, "height": 1.72, "weight": 63},
    {"name": "小陈", "age": 20, "height": 1.78, "weight": 82},
]

count = 0

def get_bmi(info):
    global count
    count += 1

    print(f"执行了 {count} 次")

    height = info["height"]
    weight = info["weight"]

    return weight / (height**2)

# 查出所有会员中过于肥胖的人的 bmi 指数
fat_bmis = [get_bmi(m) for m in members if get_bmi(m) > 24]

print(fat_bmis)
```

输出如下

```
执行了 1 次
执行了 2 次
执行了 3 次
执行了 4 次
[25.88057063502083]
```

可以看到，会员数只有 3 个，但是 get_bmi 函数却执行了 4 次，原因是在判断时执行了 3 次，而在构造新的列表时又重复执行了一遍。

如果所有会员都是过于肥胖的，那最终将执行 6 次，这种在大量的数据下是比较浪费性能的，因此对于这种结构，我通常会使用传统的for 循环 + if 判断。

```
fat_bmis = []

# 查出所有会员中过于肥胖的人的 bmi 指数
for m in members:
    bmi = get_bmi(m)
    if bmi > 24:
        fat_bmis.append(bmi)
```

在有了海象运算符之后，你就可以不用在这种场景下做出妥协。

```
# 查出所有会员中过于肥胖的人的 bmi 指数
fat_bmis = [bmi for m in members if (bmi := get_bmi(m)) > 24]
```

最终从输出结果可以看出，只执行了 3 次

```
执行了 1 次
执行了 2 次
执行了 3 次
[25.88057063502083]
```

这里仅介绍了列表推导式，但在字典推导式和集合推导式中同样适用。不再演示。

海象运算符，是一个新奇的特性，有不少人觉得这样这种特性会破坏代码的可读性。确实在一个新鲜事物刚出来时是会这样，但我相信经过时间的沉淀后，越来越多的人使用它并享受它带来的便利时，这种争议也会慢慢消失在历史的长河中。

第四章：魔法进阶扫盲

4.1 精通装饰器八大用法

Hello，装饰器

装饰器的使用方法很固定

- 先定义一个装饰器（帽子）
- 再定义你的业务函数或者类（人）
- 最后把这装饰器（帽子）扣在这个函数（人）头上

就像下面这样子

```
def decorator(func):
    def wrapper(*args, **kw):
        return func()
    return wrapper

@decorator
def function():
    print("hello, decorator")
```

实际上，装饰器并不是编码必须性，意思就是说，你不使用装饰器完全可以，它的出现，应该是使我们的代码

- 更加优雅，代码结构更加清晰
- 将实现特定的功能代码封装成装饰器，提高代码复用率，增强代码可读性

接下来，我将以实例讲解，如何编写出各种简单及复杂的装饰器。

入门：日志打印器

首先是**日志打印器**。
实现的功能：

- 在函数执行前，先打印一行日志告知一下主人，我要执行函数了。
- 在函数执行完，也不能拍拍屁股就走人了，咱可是有礼貌的代码，再打印一行日志告知下主人，我执行完啦。

```
# 这是装饰器函数，参数 func 是被装饰的函数
def logger(func):
    def wrapper(*args, **kw):
        print('主人，我准备开始执行：{} 函数了:'.format(func.__name__))

        # 真正执行的是这行。
        func(*args, **kw)
```

```
        print('主人，我执行完啦。')
    return wrapper
```

假如，我的业务函数是，计算两个数之和。写好后，直接给它带上帽子。

```
@logger
def add(x, y):
    print('{} + {} = {}'.format(x, y, x+y))
```

然后执行一下 add 函数。

```
add(200, 50)
```

来看看输出了什么?

```
主人，我准备开始执行：add 函数了：
200 + 50 = 250
主人，我执行完啦。
```

入门：时间计时器

再来看看 **时间计时器**
实现功能：顾名思义，就是计算一个函数的执行时长。

```
# 这是装饰函数
def timer(func):
    def wrapper(*args, **kw):
        t1=time.time()
        # 这是函数真正执行的地方
        func(*args, **kw)
        t2=time.time()

        # 计算下时长
        cost_time = t2-t1
        print("花费时间：{}秒".format(cost_time))
    return wrapper
```

假如，我们的函数是要睡眠10秒。这样也能更好的看出这个计算时长到底靠不靠谱。

```
import time

@timer
def want_sleep(sleep_time):
    time.sleep(sleep_time)

want_sleep(10)
```

来看看输出，如预期一样，输出10秒。

```
花费时间：10.007380008697509 8秒
```

> 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。

进阶：带参数的函数装饰器

通过上面两个简单的入门示例，你应该能体会到装饰器的工作原理了。

不过，装饰器的用法还远不止如此，深究下去，还大有文章。今天就一起来把这个知识点学透。

回过头去看看上面的例子，装饰器是不能接收参数的。其用法，只能适用于一些简单的场景。不传参的装饰器，只能对被装饰函数，执行固定逻辑。

装饰器本身是一个函数，做为一个函数，如果不能传参，那这个函数的功能就会很受限，只能执行固定的逻辑。这意味着，如果装饰器的逻辑代码的执行需要根据不同场景进行调整，若不能传参的话，我们就要写两个装饰器，这显然是不合理的。

比如我们要实现一个可以定时发送邮件的任务（一分钟发送一封），定时进行时间同步的任务（一天同步一次），就可以自己实现一个 periodic_task （定时任务）的装饰器，这个装饰器可以接收一个时间间隔的参数，间隔多长时间执行一次任务。

可以这样像下面这样写，由于这个功能代码比较复杂，不利于学习，这里就不贴了。

```
@periodic_task(spacing=60)
def send_mail():
     pass

@periodic_task(spacing=86400)
```

```
def ntp()
    pass
```

那我们来自己创造一个伪场景，可以在装饰器里传入一个参数，指明国籍，并在函数执行前，用自己国家的母语打一个招呼。

```
# 小明，中国人
@say_hello("china")
def xiaoming():
    pass

# jack，美国人
@say_hello("america")
def jack():
    pass
```

那我们如果实现这个装饰器，让其可以实现 `传参` 呢?

会比较复杂，需要两层嵌套。

```
def say_hello(contry):
    def wrapper(func):
        def deco(*args, **kwargs):
            if contry == "china":
                print("你好!")
            elif contry == "america":
                print('hello.')
            else:
                return

            # 真正执行函数的地方
            func(*args, **kwargs)
        return deco
    return wrapper
```

来执行一下

```
xiaoming()
print("------------")
jack()
```

看看输出结果。

```
你好!
------------
hello.
```

高阶：不带参数的类装饰器

以上都是基于函数实现的装饰器，在阅读别人代码时，还可以时常发现还有基于类实现的装饰器。

基于类装饰器的实现，必须实现 `__call__` 和 `__init__` 两个内置函数。
`__init__` ：接收被装饰函数
`__call__` ：实现装饰逻辑。

还是以日志打印这个简单的例子为例

```
class logger(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("[INFO]: the function {func}() is running..."\
            .format(func=self.func.__name__))
        return self.func(*args, **kwargs)

@logger
def say(something):
    print("say {}!".format(something))

say("hello")
```

执行一下，看看输出

```
[INFO]: the function say() is running...
say hello!
```

高阶：带参数的类装饰器

上面不带参数的例子，你发现没有，只能打印 `INFO` 级别的日志，正常情况下，我们还需要打印 `DEBUG` `WARNING` 等级别的日志。 这就需要给类装饰器传入参数，给这个函数指定级别了。

带参数和不带参数的类装饰器有很大的不同。

`__init__` ：不再接收被装饰函数，而是接收传入参数。
`__call__` ：接收被装饰函数，实现装饰逻辑。

```
class logger(object):
    def __init__(self, level='INFO'):
        self.level = level

    def __call__(self, func): # 接受函数
        def wrapper(*args, **kwargs):
            print("[{level}]: the function {func}() is running..."\
                .format(level=self.level, func=func.__name__))
            func(*args, **kwargs)
        return wrapper  #返回函数

@logger(level='WARNING')
def say(something):
    print("say {}!".format(something))

say("hello")
```

我们指定 `WARNING` 级别，运行一下，来看看输出。

```
[WARNING]: the function say() is running...
say hello!
```

使用偏函数与类实现装饰器

绝大多数装饰器都是基于函数和闭包实现的，但这并非制造装饰器的唯一方式。

事实上，Python 对某个对象是否能通过装饰器（ `@decorator` ）形式使用只有一个要求：**decorator 必须是一个“可被调用（callable）的对象。**

对于这个 callable 对象，我们最熟悉的就是函数了。

除函数之外，类也可以是 callable 对象，只要实现了 `__call__` 函数（上面几个例子已经接触过了）。

还有容易被人忽略的偏函数其实也是 callable 对象。

接下来就来说说，如何使用 类和偏函数结合实现一个与众不同的装饰器。

如下所示，DelayFunc 是一个实现了 `__call__` 的类，delay 返回一个偏函数，在这里 delay 就可以做为一个装饰器。（以下代码摘自 Python工匠：使用装饰器的小技巧）

```
import time
import functools

class DelayFunc:
    def __init__(self,  duration, func):
        self.duration = duration
        self.func = func

    def __call__(self, *args, **kwargs):
        print(f'Wait for {self.duration} seconds...')
        time.sleep(self.duration)
        return self.func(*args, **kwargs)

    def eager_call(self, *args, **kwargs):
        print('Call without delay')
        return self.func(*args, **kwargs)

def delay(duration):
    """
    装饰器：推迟某个函数的执行。
    同时提供 .eager_call 方法立即执行
    """
    # 此处为了避免定义额外函数，
    # 直接使用 functools.partial 帮助构造 DelayFunc 实例
    return functools.partial(DelayFunc, duration)
```

我们的业务函数很简单，就是相加

```
@delay(duration=2)
def add(a, b):
    return a+b
```

来看一下执行过程

```
>>> add    # 可见 add 变成了 Delay 的实例
<__main__.DelayFunc object at 0x107bd0be0>
```

```
>>>
>>> add(3,5)  # 直接调用实例，进入 __call__
Wait for 2 seconds...
8
>>>
>>> add.func # 实现实例方法
<function add at 0x107bef1e0>
```

> 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。

如何写能装饰类的装饰器?

用 Python 写单例模式的时候，常用的有三种写法。其中一种，是用装饰器来实现的。

以下便是我自己写的装饰器版的单例写法。

```
instances = {}

def singleton(cls):
    def get_instance(*args, **kw):
        cls_name = cls.__name__
        print('===== 1 ====')
        if not cls_name in instances:
            print('===== 2 ====')
            instance = cls(*args, **kw)
            instances[cls_name] = instance
        return instances[cls_name]
    return get_instance

@singleton
class User:
    _instance = None

    def __init__(self, name):
        print('===== 3 ====')
        self.name = name
```

可以看到我们用singleton 这个装饰函数来装饰 User 这个类。装饰器用在类上，并不是很常见，但只要熟悉装饰器的实现过程，就不难以实现对类的装饰。在上面这个例子中，装饰器就只是实

现对类实例的生成的控制而已。

其实例化的过程，你可以参考我这里的调试过程，加以理解。

```
91    instances = {}
92
93    def singleton(cls):
94        def get_instance(*args, **kw):
95            cls_name = cls.__name__
96            print('===== 1 ====')
97            if not cls_name in instances:
98                print('===== 2 ====')
99                instance = cls(*args, **kw)
100               instances[cls_name] = instance
101           return instances[cls_name]
102       return get_instance
103
104   @singleton
105   class User:
106       _instance = None
107
108       def __init__(self, name):
109           print('===== 3 ====')
110           self.name = name
```

singleton()

mytest(3)

```
>>> u1 = User('wangbm1')
===== 1 ====
===== 2 ====
===== 3 ====
>>> u1.age = 20
>>> u2 = User('wangbm2')
===== 1 ====
>>> u2.age
20
>>> u1 is u2
True
>>>
```


wraps 装饰器有啥用?

在 functools 标准库中有提供一个 wraps 装饰器，你应该也经常见过，那他有啥用呢?

先来看一个例子

```
def wrapper(func):
    def inner_function():
        pass
    return inner_function

@wrapper
def wrapped():
    pass

print(wrapped.__name__)
#inner_function
```

为什么会这样子？不是应该返回 `func` 吗？

这也不难理解，因为上边执行 `func` 和下边 `decorator(func)` 是等价的，所以上面 `func.__name__` 是等价于下面 `decorator(func).__name__` 的，那当然名字是 `inner_function`

```
def wrapper(func):
    def inner_function():
        pass
    return inner_function

def wrapped():
    pass

print(wrapper(wrapped).__name__)
#inner_function
```

那如何避免这种情况的产生？方法是使用 functools .wraps 装饰器，它的作用就是将 **被修饰的函数(wrapped)** 的一些属性值赋值给 **修饰器函数(wrapper)**，最终让属性的显示更符合我们的直觉。

```
from functools import wraps

def wrapper(func):
    @wraps(func)
    def inner_function():
        pass
```

```
    return inner_function

@wrapper
def wrapped():
    pass

print(wrapped.__name__)
# wrapped
```

准确点说，wraps 其实是一个偏函数对象（partial），源码如下

```
def wraps(wrapped,
          assigned = WRAPPER_ASSIGNMENTS,
          updated = WRAPPER_UPDATES):
    return partial(update_wrapper, wrapped=wrapped,
                   assigned=assigned, updated=updated)
```

可以看到wraps其实就是调用了一个函数 `update_wrapper`，知道原理后，我们改写上面的代码，在不使用 wraps的情况下，也可以让 `wrapped.__name__` 打印出 wrapped，代码如下：

```
from functools import update_wrapper

WRAPPER_ASSIGNMENTS = ('__module__', '__name__', '__qualname__', '__doc__',
                       '__annotations__')

def wrapper(func):
    def inner_function():
        pass

    update_wrapper(inner_function, func, assigned=WRAPPER_ASSIGNMENTS)
    return inner_function

@wrapper
def wrapped():
    pass

print(wrapped.__name__)
```

4.2 深入理解描述符

学习 Python 这么久了，说起 Python 的优雅之处，能让我脱口而出的， Descriptor（描述符）特性可以排得上号。

描述符 是Python 语言独有的特性，它不仅在应用层使用，在语言的基础设施中也有涉及。

我可以大胆地猜测，你对于描述符的了解是始于诸如 Django ORM 和 SQLAlchemy 中的字段对象，是的，它们都是描述符。你的它的认识，可能也止步于此，如果你没有去深究，它为何要如此设计？也就加体会不到 Python 给我们带来的便利与优雅。

由于 描述符的内容较多，长篇大论，容易让你倦怠，所以我打算分几篇来讲。

为什么要使用描述符?

假想你正在给学校写一个成绩管理系统，并没有太多编码经验的你，可能会这样子写。

```
class Student:
    def __init__(self, name, math, chinese, english):
        self.name = name
        self.math = math
        self.chinese = chinese
        self.english = english

    def __repr__(self):
        return "<Student: {}, math:{}, chinese: {}, english:{}>".format(
                self.name, self.math, self.chinese, self.english
            )
```

看起来一切都很合理

```
>>> std1 = Student('小明', 76, 87, 68)
>>> std1
<Student: 小明, math:76, chinese: 87, english:68>
```

但是程序并不像人那么智能，不会自动根据使用场景判断数据的合法性，如果老师在录入成绩的

时候，不小心录入了将成绩录成了负数，或者超过100，程序是无法感知的。

聪明的你，马上在代码中加入了判断逻辑。

```
class Student:
    def __init__(self, name, math, chinese, english):
        self.name = name
        if 0 <= math <= 100:
            self.math = math
        else:
            raise ValueError("Valid value must be in [0, 100]")

        if 0 <= chinese <= 100:
            self.chinese = chinese
        else:
            raise ValueError("Valid value must be in [0, 100]")

        if 0 <= chinese <= 100:
            self.english = english
        else:
            raise ValueError("Valid value must be in [0, 100]")


    def __repr__(self):
        return "<Student: {}, math:{}, chinese: {}, english:{}>".format(
                self.name, self.math, self.chinese, self.english
            )
```

这下程序稍微有点人工智能了，能够自己明辨是非了。

```
>>> std1 = Student('小明', -76, 87, 68)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "F:/VMP-Code/Python Script/test/mytest.py", line 26, in __init__
    self.math = math
  File "F:/VMP-Code/Python Script/test/mytest.py", line 9, in __set__
    raise ValueError('Valid value must be in [0, 100]')
ValueError: Valid value must be in [0, 100]
>>> std2 = Student('小明', 76, 120, 68)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "F:/VMP-Code/Python Script/test/mytest.py", line 27, in __init__
    self.chinese = chinese
  File "F:/VMP-Code/Python Script/test/mytest.py", line 9, in __set__
    raise ValueError('Valid value must be in [0, 100]')
ValueError: Valid value must be in [0, 100]
```

程序是智能了，但在 `__init__` 里有太多的判断逻辑，很影响代码的可读性。巧的是，你刚好学过 Property 特性，可以很好的应用在这里。于是你将代码修改成如下，代码的可读性瞬间提升了不少

```python
class Student:
    def __init__(self, name, math, chinese, english):
        self.name = name
        self.math = math
        self.chinese = chinese
        self.english = english

    @property
    def math(self):
        return self._math

    @math.setter
    def math(self, value):
        if 0 <= value <= 100:
            self._math = value
        else:
            raise ValueError("Valid value must be in [0, 100]")

    @property
    def chinese(self):
        return self._chinese
```

```
    @chinese.setter
    def chinese(self, value):
        if 0 <= value <= 100:
            self._chinese = value
        else:
            raise ValueError("Valid value must be in [0, 100]")

    @property
    def english(self):
        return self._english

    @english.setter
    def english(self, value):
        if 0 <= value <= 100:
            self._english = value
        else:
            raise ValueError("Valid value must be in [0, 100]")

    def __repr__(self):
        return "<Student: {}, math:{}, chinese: {}, english:{}>".format(
                self.name, self.math, self.chinese, self.english
            )
```

程序还是一样的人工智能，非常好。

```
>>> std1 = Student('小明', -76, 87, 68)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "F:/VMP-Code/Python Script/test/mytest.py", line 26, in __init__
    self.math = math
  File "F:/VMP-Code/Python Script/test/mytest.py", line 9, in __set__
    raise ValueError('Valid value must be in [0, 100]')
ValueError: Valid value must be in [0, 100]
>>> std2 = Student('小明', 76, 120, 68)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "F:/VMP-Code/Python Script/test/mytest.py", line 27, in __init__
    self.chinese = chinese
  File "F:/VMP-Code/Python Script/test/mytest.py", line 9, in __set__
    raise ValueError('Valid value must be in [0, 100]')
ValueError: Valid value must be in [0, 100]
```

你以为你写的代码，已经非常优秀，无懈可击了。

> 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。

没想到，人外有天，你的主管看了你的代码后，深深地叹了口气：类里的三个属性，math、chinese、english，都使用了 Property 对属性的合法性进行了有效控制。功能上，没有问题，但就是太啰嗦了，三个变量的合法性逻辑都是一样的，只要大于0，小于100 就可以，代码重复率太高了，这里三个成绩还好，但假设还有地理、生物、历史、化学等十几门的成绩呢，这代码简直没法忍。去了解一下 Python 的描述符吧。

经过主管的指点，你知道了「描述符」这个东西。怀着一颗敬畏之心，你去搜索了下关于 描述符的用法。

其实也很简单，一个实现了 `描述符协议` 的类就是一个描述符。

什么描述符协议：实现了 `__get__()` 、 `__set__()` 、 `__delete__()` 其中至少一个方法的类，就是一个描述符。

- `__get__` ： 用于访问属性。它返回属性的值，若属性不存在、不合法等都可以抛出对应的异常。
- `__set__` ：将在属性分配操作中调用。不会返回任何内容。
- `__delete__` ：控制删除操作。不会返回内容。

对描述符有了大概的了解后，你开始重写上面的方法。

如前所述，Score 类是一个描述器，当从 Student 的实例访问 math、chinese、english这三个属性的时候，都会经过 Score 类里的三个特殊的方法。这里的 Score 避免了 使用Property 出现大量的代码无法复用的尴尬。

```
class Score:
    def __init__(self, default=0):
        self._score = default

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Score must be integer')
        if not 0 <= value <= 100:
            raise ValueError('Valid value must be in [0, 100]')

        self._score = value
```

```
    def __get__(self, instance, owner):
        return self._score

    def __delete__(self):
        del self._score

class Student:
    math = Score(0)
    chinese = Score(0)
    english = Score(0)

    def __init__(self, name, math, chinese, english):
        self.name = name
        self.math = math
        self.chinese = chinese
        self.english = english


    def __repr__(self):
        return "<Student: {}, math:{}, chinese: {}, english:{}>".format(
                self.name, self.math, self.chinese, self.english
            )
```

实现的效果和前面的一样，可以对数据的合法性进行有效控制（字段类型、数值区间等）

```
>>> std1 = Student('小明', 76, 120, 68)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "F:/VMP-Code/Python Script/test/mytest.py", line 27, in __init__
    self.chinese = chinese
  File "F:/VMP-Code/Python Script/test/mytest.py", line 9, in __set__
    raise ValueError('Valid value must be in [0, 100]')
ValueError: Valid value must be in [0, 100]
>>> std1 = Student('小明', 76, "九十", 68)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "F:/VMP-Code/Python Script/test/mytest.py", line 27, in __init__
    self.chinese = chinese
  File "F:/VMP-Code/Python Script/test/mytest.py", line 7, in __set__
>>>
```

以上，我举了下具体的实例，从最原始的编码风格到 Property ， 最后引出描述符。由浅入深，一步一步带你感受到描述符的优雅之处。

通过此文，你需要记住的只有一点，就是描述符给我们带来的编码上的便利，它在实现 `保护属性不受修改` 、 `属性类型检查` 的基本功能，同时有大大提高代码的复用率。

> 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。

描述符的访问规则

描述符分两种：

- 数据描述符：实现了 `__get__` 和 `__set__` 两种方法的描述符
- 非数据描述符：只实现了 `__get__` 一种方法的描述符

你一定会问，他们有什么区别呢？网上的讲解，我看过几个，很多都把一个简单的东西讲得复杂了。

其实就一句话，**数据描述器和非数据描述器的区别在于：它们相对于实例的字典的优先级不同。**

如果实例字典中有与描述器同名的属性，如果描述器是数据描述器，优先使用数据描述器，如果是非数据描述器，优先使用字典中的属性。

这边还是以上节的成绩管理的例子来说明，方便你理解。

```
# 数据描述符
class DataDes:
    def __init__(self, default=0):
        self._score = default

    def __set__(self, instance, value):
        self._score = value

    def __get__(self, instance, owner):
        print("访问数据描述符里的 __get__")
        return self._score

# 非数据描述符
class NoDataDes:
    def __init__(self, default=0):
        self._score = default

    def __get__(self, instance, owner):
        print("访问非数据描述符里的 __get__")
```

```
        return self._score


class Student:
    math = DataDes(0)
    chinese = NoDataDes(0)

    def __init__(self, name, math, chinese):
        self.name = name
        self.math = math
        self.chinese = chinese

    def __getattribute__(self, item):
        print("调用 __getattribute__")
        return super(Student, self).__getattribute__(item)

    def __repr__(self):
        return "<Student: {}, math:{}, chinese: {},>".format(
                self.name, self.math, self.chinese)
```

需要注意的是，math 是数据描述符，而 chinese 是非数据描述符。从下面的验证中，可以看出，当实例属性和数据描述符同名时，会优先访问数据描述符（如下面的math），而当实例属性和非数据描述符同名时，会优先访问实例属性（`__getattribute__`）

```
>>> std = Student('xm', 88, 99)
>>>
>>> std.math
调用 __getattribute__
访问数据描述符里的 __get__
88
>>> std.chinese
调用 __getattribute__
99
```

讲完了数据描述符和非数据描述符，我们还需要了解的对象属性的查找规律。

当我们对一个实例属性进行访问时，Python 会按 `obj.__dict__` → `type(obj).__dict__` → `type(obj)的父类.__dict__` 顺序进行查找，如果查找到目标属性并发现是一个描述符，Python 会调用描述符协议来改变默认的控制行为。

基于描述符如何实现property

经过上面的讲解，我们已经知道如何定义描述符，且明白了描述符是如何工作的。

正常人所见过的描述符的用法就是上篇文章提到的那些，我想说的是那只是描述符协议最常见的应用之一，或许你还不知道，其实有很多 Python 的特性的底层实现机制都是基于 `描述符协议` 的，比如我们熟悉的 `@property` 、 `@classmethod` 、 `@staticmethod` 和 `super` 等。

先来说说 `property` 吧。

有了第一篇的基础，我们知道了 property 的基本用法。这里我直接切入主题，从第一篇的例子里精简了一下。

```
class Student:
    def __init__(self, name):
        self.name = name

    @property
    def math(self):
        return self._math

    @math.setter
    def math(self, value):
        if 0 <= value <= 100:
            self._math = value
        else:
            raise ValueError("Valid value must be in [0, 100]")
```

不防再简单回顾一下它的用法，通过property装饰的函数，如例子中的 math 会变成 Student 实例的属性。而对 math 属性赋值会进入 使用 `math.setter` 装饰函数的逻辑代码块。

为什么说 property 底层是基于描述符协议的呢？通过 PyCharm 点击进入 property 的源码，很可惜，只是一份类似文档一样的伪源码，并没有其具体的实现逻辑。

不过，从这份伪源码的魔法函数结构组成，可以大体知道其实现逻辑。

这里我自己通过模仿其函数结构，结合「描述符协议」来自己实现类 `property` 特性。

代码如下：

```
class TestProperty(object):

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc
```

```
    def __get__(self, obj, objtype=None):
        print("in __get__")
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError
        return self.fget(obj)

    def __set__(self, obj, value):
        print("in __set__")
        if self.fset is None:
            raise AttributeError
        self.fset(obj, value)

    def __delete__(self, obj):
        print("in __delete__")
        if self.fdel is None:
            raise AttributeError
        self.fdel(obj)


    def getter(self, fget):
        print("in getter")
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        print("in setter")
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        print("in deleter")
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

然后 Student 类，我们也相应改成如下

```
class Student:
    def __init__(self, name):
        self.name = name

    # 其实只有这里改变
    @TestProperty
    def math(self):
        return self._math

    @math.setter
```

```
    def math(self, value):
        if 0 <= value <= 100:
            self._math = value
        else:
            raise ValueError("Valid value must be in [0, 100]")
```

> 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。

为了尽量让你少产生一点疑惑，我这里做两点说明：

1. 使用 `TestProperty` 装饰后， `math` 不再是一个函数，而是 `TestProperty` 类的一个实例。所以第二个math函数可以使用 `math.setter` 来装饰，本质是调用 `TestProperty.setter` 来产生一个新的 `TestProperty` 实例赋值给第二个 `math` 。
2. 第一个 `math` 和第二个 `math` 是两个不同 `TestProperty` 实例。但他们都属于同一个描述符类（TestProperty），当对 math 对于赋值时，就会进入 `TestProperty.__set__` ，当对math 进行取值里，就会进入 `TestProperty.__get__` 。仔细一看，其实最终访问的还是Student实例的 `_math` 属性。

说了这么多，还是运行一下，更加直观一点。

```
# 运行后，会直接打印这一行，这是在实例化 TestProperty 并赋值给第二个math
in setter
>>>
>>> s1.math = 90
in __set__
>>> s1.math
in __get__
90
```

对于以上理解 `property` 的运行原理有困难的同学，请务必参照我上面写的两点说明。如有其他疑问，可以加微信与我进行探讨。

基于描述符如何实现staticmethod

说完了 `property` ，这里再来讲讲 `@classmethod` 和 `@staticmethod` 的实现原理。

我这里定义了一个类，用了两种方式来实现静态方法。

```
class Test:
    @staticmethod
    def myfunc():
        print("hello")

# 上下两种写法等价

class Test:
    def myfunc():
        print("hello")
    # 重点：这就是描述符的体现
    myfunc = staticmethod(myfunc)
```

这两种写法是等价的，就好像在 `property` 一样，其实以下两种写法也是等价的。

```
@TestProperty
def math(self):
    return self._math

math = TestProperty(fget=math)
```

话题还是转回到 `staticmethod` 这边来吧。

由上面的注释，可以看出 `staticmethod` 其实就相当于一个描述符类，而 `myfunc` 在此刻变成了一个描述符。关于 `staticmethod` 的实现，你可以参照下面这段我自己写的代码，加以理解。

```
class staticmethod:
    def __init__(self, f):
        self.f = f

    def __get__(self, obj, objtype=None):
        print("in staticmethod __get__")
        return self.f


class Test:
    def myfunc():
        print("hello")

    myfunc = staticmethod(myfunc)
```

PyCharm 虽会检测出语法错误
但并不会影响实际代码运行。

调用这个方法可以知道，每调用一次，它都会经过描述符类的 `__get__` 。

```
>>> Test.myfunc()
in staticmethod __get__
hello
>>> Test().myfunc()
in staticmethod __get__
hello
```

基于描述符如何实现classmethod

同样的 `classmethod` 也是一样。

```
class classmethod(object):
    def __init__(self, f):
        self.f = f

    def __get__(self, instance, owner=None):
        print("in classmethod __get__")

        def newfunc(*args):
            return self.f(owner, *args)
        return newfunc

class Test:
    def myfunc(cls):
        print("hello")

    # 重点：这就是描述符的体现
    myfunc = classmethod(myfunc)
```

验证结果如下

```
>>> Test.myfunc()
in classmethod __get__
hello
>>> Test().myfunc()
in classmethod __get__
hello
```

讲完了 `property` 、 `staticmethod` 和 `classmethod` 与 描述符的关系。我想你应该对描述符在 Python 中的应用有了更深的理解。对于 super 的实现原理，就交由你来自己完成。

所有实例共享描述符

若要合理使用描述符，利用描述符给我们带来的编码上的便利。有一个坑，需要注意，比如下面这个Student我们没有定义构造函数

```
class Score:
    def __init__(self, default=0):
        self._value = default

    def __get__(self, instance, owner):
        return self._value

    def __set__(self, instance, value):
        if 0 <= value <= 100:
            self._value = value
        else:
            raise ValueError


class Student:
    math = Score(0)
    chinese = Score(0)
    english = Score(0)

    def __repr__(self):
        return "<Student math:{}, chinese:{}, english:{}>".format(self.math, se
lf.chinese, self.english)
```

看一下会出现什么样的问题，std2 居然共享了std1 的属性值，因为它被当成了一个类变量了，而每个实例都没有自己的实例变量，自然访问的是同一个变量。这样是很难达到我们使用描述符的初衷。

```
>>> std1 = Student()
>>> std1
<Student math:0, chinese:0, english:0>
>>> std1.math = 85
>>> std1
<Student math:85, chinese:0, english:0>
>>> std2 = Student()
>>> std2 # std2 居然共享了std1 的属性值
<Student math:85, chinese:0, english:0>
>>> std2.math = 100
>>> std1 # std2 也会改变std1 的属性值
<Student math:100, chinese:0, english:0>
```

而正确的做法应该是，所有的实例数据只属于该实例本身（通过实例初始化传入），具体写法可参考上一节。

> 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。

4.3 精通上下文管理器

`with` 这个关键字，对于每一学习Python的人，都不会陌生。

操作文本对象的时候，几乎所有的人都会让我们要用 `with open` ，这就是一个上下文管理的例子。你一定已经相当熟悉了，我就不再废话了。

```
with open('test.txt') as f:
    print f.readlines()
```

what context manager?

基本语法

```
with EXPR as VAR:
    BLOCK
```

先理清几个概念

```
1. 上下文表达式：with open('test.txt') as f:
2. 上下文管理器：open('test.txt')
3. f 不是上下文管理器，应该是资源对象。
```

how context manager?

要自己实现这样一个上下文管理，要先知道上下文管理协议。

简单点说，就是在一个类里，实现了 `__enter__` 和 `__exit__` 的方法，这个类的实例就是一个上下文管理器。

例如这个示例：

```
class Resource():
    def __enter__(self):
        print('===connect to resource===')
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print('===close resource connection===')

    def operate(self):
        print('===in operation===')

with Resource() as res:
    res.operate()
```

我们执行一下，通过日志的打印顺序。可以知道其执行过程。

```
===connect to resource===
===in operation===
===close resource connection===
```

从这个示例可以很明显的看出，在编写代码时，可以将资源的连接或者获取放在 `__enter__` 中，而将资源的关闭写在 `__exit__` 中。

why context manager?

学习时多问自己几个为什么，养成对一些细节的思考，有助于加深对知识点的理解。

为什么要使用上下文管理器?

在我看来，这和 Python 崇尚的优雅风格有关。

1. 可以以一种更加优雅的方式，操作（创建/获取/释放）资源，如文件操作、数据库连接；
2. 可以以一种更加优雅的方式，处理异常；

第一种，我们上面已经以资源的连接为例讲过了。

而第二种，会被大多数人所忽略。这里会重点讲一下。

大家都知道，处理异常，通常都是使用 `try...execept..` 来捕获处理的。这样做一个不好的地方是，在代码的主逻辑里，会有大量的异常处理代理，这会很大的影响我们的可读性。

好一点的做法呢，可以使用 `with` 将异常的处理隐藏起来。

仍然是以上面的代码为例，我们将 `1/0` 这个 `一定会抛出异常的代码` 写在 `operate` 里

```
class Resource():
    def __enter__(self):
        print('===connect to resource===')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('===close resource connection===')
        return True

    def operate(self):
        1/0

with Resource() as res:
    res.operate()
```

运行一下，惊奇地发现，居然不会报错。

这就是上下文管理协议的一个强大之处，异常可以在 `__exit__` 进行捕获并由你自己决定如何处理，是抛出呢还是在这里就解决了。在 `__exit__` 里返回 `True` （没有return 就默认为 return False），就相当于告诉 Python解释器，这个异常我们已经捕获了，不需要再往外抛了。

在 写 `__exit__` 函数时，需要注意的事，它必须要有这三个参数：

- exc_type：异常类型
- exc_val：异常值
- exc_tb：异常的错误栈信息

当主逻辑代码没有报异常时，这三个参数将都为None。

how contextlib?

在上面的例子中，我们只是为了构建一个上下文管理器，却写了一个类。如果只是要实现一个简单的功能，写一个类未免有点过于繁杂。这时候，我们就想，如果只写一个函数就可以实现上下文管理器就好了。

这个点Python早就想到了。它给我们提供了一个装饰器，你只要按照它的代码协议来实现函数内容，就可以将这个函数对象变成一个上下文管理器。

我们按照 contextlib 的协议来自己实现一个打开文件（with open）的上下文管理器。

```
import contextlib

@contextlib.contextmanager
def open_func(file_name):
```

```
    # __enter__方法
    print('open file:', file_name, 'in __enter__')
    file_handler = open(file_name, 'r')

    # 【重点】：yield
    yield file_handler

    # __exit__方法
    print('close file:', file_name, 'in __exit__')
    file_handler.close()
    return

with open_func('/Users/MING/mytest.txt') as file_in:
    for line in file_in:
        print(line)
```

在被装饰函数里，必须是一个生成器（带有yield），而yield之前的代码，就相当于 `__enter__` 里的内容。yield 之后的代码，就相当于 `__exit__` 里的内容。

上面这段代码只能实现上下文管理器的第一个目的（管理资源），并不能实现第二个目的（处理异常）。

如果要处理异常，可以改成下面这个样子。

```
import contextlib

@contextlib.contextmanager
def open_func(file_name):
    # __enter__方法
    print('open file:', file_name, 'in __enter__')
    file_handler = open(file_name, 'r')

    try:
        yield file_handler
    except Exception as exc:
        # deal with exception
        print('the exception was thrown')
    finally:
        print('close file:', file_name, 'in __exit__')
        file_handler.close()

        return

with open_func('/Users/MING/mytest.txt') as file_in:
    for line in file_in:
```

```
        1/0
        print(line)
```

好像只要讲到上下文管理器，大多数人都会谈到打开文件这个经典的例子。

但是在实际开发中，可以使用到上下文管理器的例子也不少。我这边举个我自己的例子。

在OpenStack中，给一个虚拟机创建快照时，需要先创建一个临时文件夹，来存放这个本地快照镜像，等到本地快照镜像创建完成后，再将这个镜像上传到Glance。然后删除这个临时目录。

这段代码的主逻辑是 `创建快照` ，而 `创建临时目录` ，属于前置条件， `删除临时目录` ，是收尾工作。

虽然代码量很少，逻辑也不复杂，但是“ `创建临时目录，使用完后再删除临时目录` ”这个功能，在一个项目中很多地方都需要用到，如果可以将这段逻辑处理写成一个工具函数作为一个上下文管理器，那代码的复用率也大大提高。

代码是这样的

```
@contextlib.contextmanager
def tempdir(**kwargs):
    argdict = kwargs.copy()
    if 'dir' not in argdict:
        argdict['dir'] = CONF.tempdir
    tmpdir = tempfile.mkdtemp(**argdict)
    try:
        yield tmpdir
    finally:
        try:
            shutil.rmtree(tmpdir)
        except OSError as e:
            LOG.error(_LE('Could not remove tmpdir: %s'), e)
```

总结起来，使用上下文管理器有三个好处：

1. 提高代码的复用率；
2. 提高代码的优雅度；

3. 提高代码的可读性；

第五章：魔法开发技巧

5.1 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 ============')
```

5.2 将嵌套 for 循环写成单行

我们经常会如下这种嵌套的 for 循环代码

```
list1 = range(1,3)
list2 = range(4,6)
list3 = range(7,9)
for item1 in list1:
    for item2 in list2:
        for item3 in list3:
              print(item1+item2+item3)
```

这里仅仅是三个 for 循环，在实际编码中，有可能会有更层。

这样的代码，可读性非常的差，很多人不想这么写，可又没有更好的写法。

这里介绍一种我常用的写法，使用 itertools 这个库来实现更优雅易读的代码。

```
from itertools import product
list1 = range(1,3)
list2 = range(4,6)
list3 = range(7,9)
for item1,item2,item3 in product(list1, list2, list3):
    print(item1+item2+item3)
```

输出如下

```
$ python demo.py
12
13
13
14
13
14
14
15
```

5.3 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。

5.4 如何关闭异常自动关联上下文?

当你在处理异常时，由于处理不当或者其他问题，再次抛出另一个异常时，往外抛出的异常也会携带原始的异常信息。

就像这样子。

```
try:
    print(1 / 0)
except Exception as exc:
    raise RuntimeError("Something bad happened")
```

从输出可以看到两个异常信息

```
Traceback (most recent call last):
  File "demo.py", line 2, in <module>
    print(1 / 0)
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "demo.py", line 4, in <module>
    raise RuntimeError("Something bad happened")
RuntimeError: Something bad happened
```

如果在异常处理程序或 finally 块中引发异常，默认情况下，异常机制会隐式工作会将先前的异常附加为新异常的 `__context__` 属性。这就是 Python 默认开启的自动关联异常上下文。

如果你想自己控制这个上下文，可以加个 from 关键字（`from` 语法会有个限制，就是第二个表达式必须是另一个异常类或实例。），来表明你的新异常是直接由哪个异常引起的。

```
try:
    print(1 / 0)
except Exception as exc:
```

```
    raise RuntimeError("Something bad happened") from exc
```

输出如下

```
Traceback (most recent call last):
  File "demo.py", line 2, in <module>
    print(1 / 0)
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "demo.py", line 4, in <module>
    raise RuntimeError("Something bad happened") from exc
RuntimeError: Something bad happened
```

当然，你也可以通过 `with_traceback()` 方法为异常设置上下文 `__context__` 属性，这也能在 `traceback` 更好的显示异常信息。

```
try:
    print(1 / 0)
except Exception as exc:
    raise RuntimeError("bad thing").with_traceback(exc)
```

最后，如果我想彻底关闭这个自动关联异常上下文的机制？有什么办法呢？

可以使用 `raise...from None`，从下面的例子上看，已经没有了原始异常

```
$ cat demo.py
try:
    print(1 / 0)
except Exception as exc:
    raise RuntimeError("Something bad happened") from None
$
$ python demo.py
Traceback (most recent call last):
  File "demo.py", line 4, in <module>
    raise RuntimeError("Something bad happened") from None
RuntimeError: Something bad happened
(PythonCodingTime)
```

5.5 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
```


5.6 如何流式读取数G超大文件

使用 with...open... 可以从一个文件中读取数据，这是所有 Python 开发者都非常熟悉的操作。

但是如果你使用不当，也会带来很大的麻烦。

比如当你使用了 read 函数，其实 Python 会将文件的内容一次性的全部载入内存中，如果文件有 10 个G甚至更多，那么你的电脑就要消耗的内存非常巨大。

```
# 一次性读取
with open("big_file.txt", "r") as fp:
    content = fp.read()
```

对于这个问题，你也许会想到使用 readline 去做一个生成器来逐行返回。

```
def read_from_file(filename):
    with open(filename, "r") as fp:
        yield fp.readline()
```

可如果这个文件内容就一行呢，一行就 10个G，其实你还是会一次性读取全部内容。

最优雅的解决方法是，在使用 read 方法时，指定每次只读取固定大小的内容，比如下面的代码中，每次只读取 8kb 返回。

```
def read_from_file(filename, block_size = 1024 * 8):
    with open(filename, "r") as fp:
        while True:
            chunk = fp.read(block_size)
```

```
            if not chunk:
                break

            yield chunk
```

上面的代码，功能上已经没有问题了，但是代码看起来代码还是有些臃肿。

借助偏函数 和 iter 函数可以优化一下代码

```
from functools import partial

def read_from_file(filename, block_size = 1024 * 8):
    with open(filename, "r") as fp:
        for chunk in iter(partial(fp.read, block_size), ""):
            yield chunk
```

> 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。

5.7 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
```


5.8 如何快速计算函数运行时间

计算一个函数的运行时间，你可能会这样子做

```
import time

start = time.time()

# run the function

end = time.time()
print(end-start)
```

你看看你为了计算函数运行时间，写了几行代码了。

有没有一种方法可以更方便的计算这个运行时间呢?

有。

有一个内置模块叫 timeit

使用它，只用一行代码即可

```
import time
import timeit

def run_sleep(second):
    print(second)
    time.sleep(second)

# 只用这一行
print(timeit.timeit(lambda :run_sleep(2), number=5))
```

运行结果如下

```
2
2
2
2
2
10.020059824
```


5.9 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()
```


5.10 快速定位错误进入调试模式

当你在写一个程序时，最初的程序一定遇到不少零零散散的错误，这时候就免不了调试一波。

如果你和我一样，习惯使用 pdb 进行调试的话，一定有所体会，通常我们都要先把 `pdb.set_trace()` 去掉，让程序畅通无阻，直到它把异常抛出来。

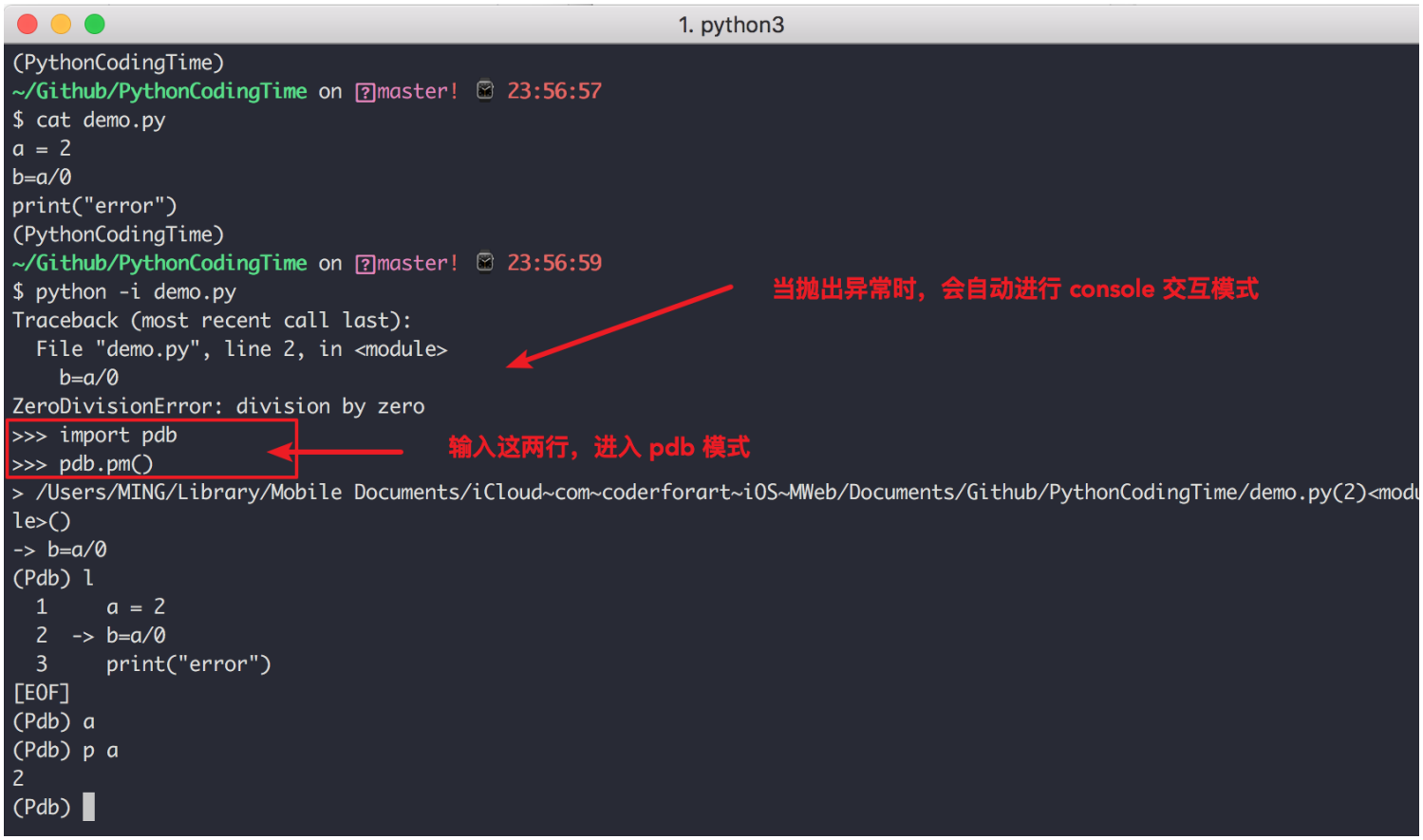
出现异常后，再使用 vim 跳转到抛出异常的位置，敲入 `import pdb;pdb.set_trace()` ，然后再到运行，进入调试模式，找到问题并修改代码后再去掉我们加上的那行 pdb 的代码。

如此反复这样一个过程，直到最后程序没有异常。

你应该能够感受到这个过程有多繁锁，令人崩溃。

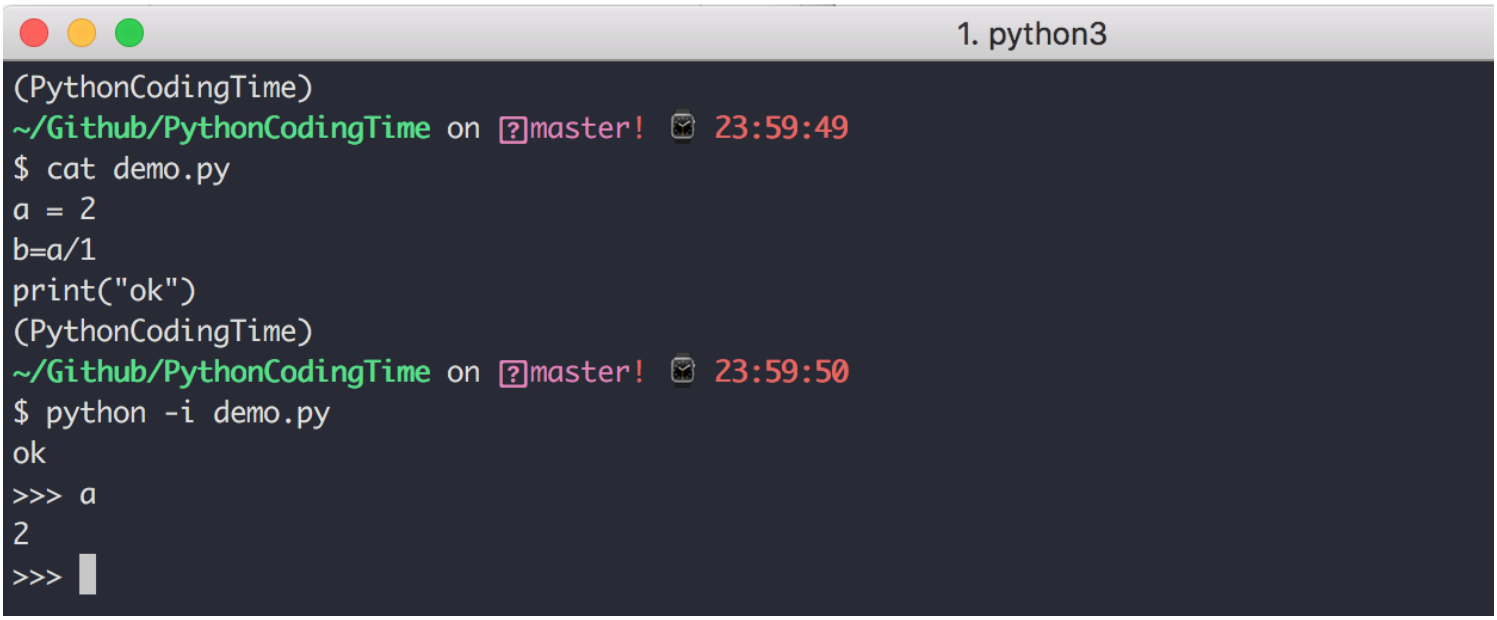
接下来介绍一种，可以让你不需要修改源代码，就可以在异常抛出时，快速切换到调试模式，进入 『案发现场』排查问题。

方法很简单，只需要你在执行脚本时，加入 `-i` 参考



如果你的程序没有任何问题，加上 `-i` 后又会有什么不一样呢?

从下图可以看出，程序执行完成后会自动进入 console 交互模式。

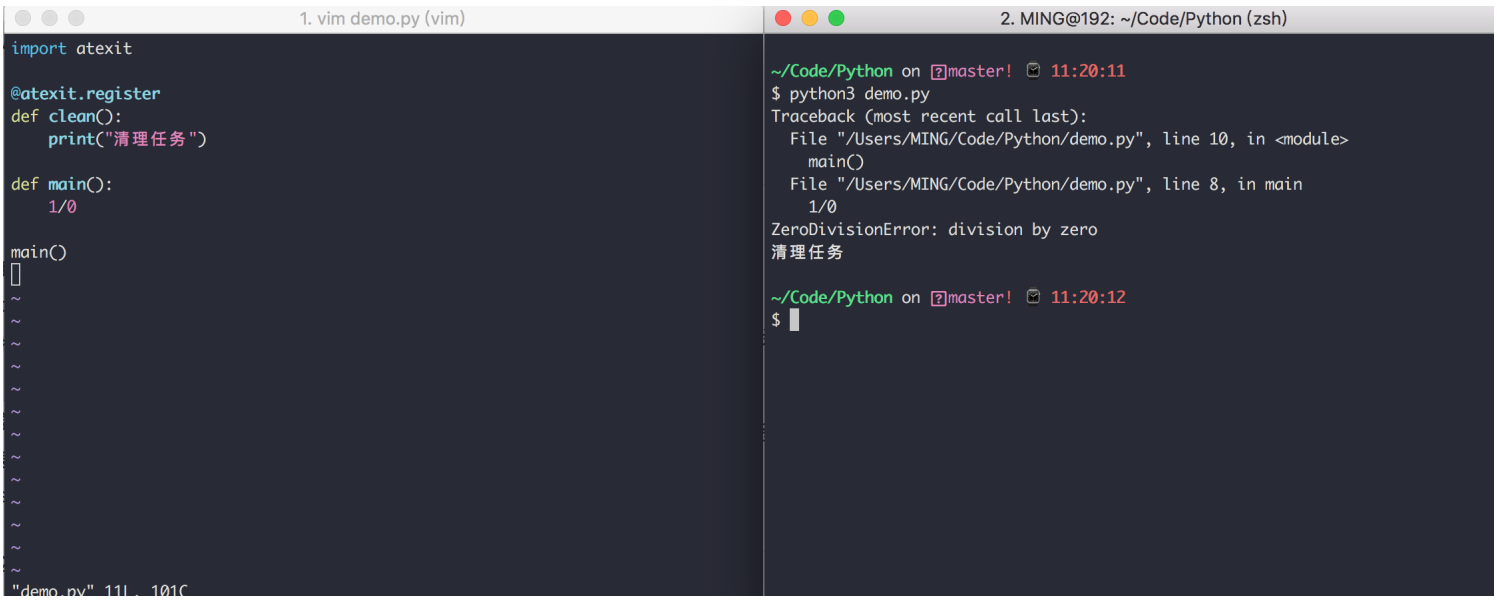


5.11 在程序退出前执行代码的技巧

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



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。

5.12 逗号也有它的独特用法

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r

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。

5.13 如何在运行状态查看源代码?

查看函数的源代码，我们通常会使用 IDE 来完成。

比如在 PyCharm 中，你可以 Ctrl + 鼠标点击 进入函数的源代码。

那如果没有 IDE 呢?

当我们想使用一个函数时，如何知道这个函数需要接收哪些参数呢?

当我们在使用函数时出现问题的时候，如何通过阅读源代码来排查问题所在呢?

这时候，我们可以使用 inspect 来代替 IDE 帮助你完成这些事

```
# demo.py
import inspect


def add(x, y):
    return x + y

print("====================")
print(inspect.getsource(add))
```

运行结果如下

```
$ python demo.py
====================
def add(x, y):
```

```
    return x + y
```

> 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。

5.14 少为人知的重试机制

为了避免由于一些网络或等其他不可控因素，而引起的功能问题。

比如在发送请求时，会因为网络问题，而导致请求超时的问题。

这种情况下，我们往往会在代码中加入重试的代码，代码本身不难实现，但如何写得优雅、易用，是我们要考虑的问题。

这里要给大家介绍的是一个第三方库 – `Tenacity` ，它实现了几乎我们可以使用到的所有重试场景，比如：

1. 在什么情况下才进行重试?
2. 重试几次呢?
3. 重试多久后结束?
4. 每次重试的间隔多长呢?
5. 重试失败后的回调?

在使用它之前 ，先要安装它

```
$ pip install tenacity
```

最基本的重试

无条件重试，重试之间无间隔

```
from tenacity import retry

@retry
def test_retry():
    print("等待重试，重试无间隔执行...")
    raise Exception

test_retry()
```

无条件重试，但是在重试之前要等待 2 秒

```
from tenacity import retry, wait_fixed

@retry(wait=wait_fixed(2))
def test_retry():
    print("等待重试...")
    raise Exception

test_retry()
```

设置停止基本条件

只重试7 次

```
from tenacity import retry, stop_after_attempt

@retry(stop=stop_after_attempt(7))
def test_retry():
    print("等待重试...")
    raise Exception

test_retry()
```

重试 10 秒后不再重试

```
from tenacity import retry, stop_after_delay

@retry(stop=stop_after_delay(10))
def test_retry():
    print("等待重试...")
    raise Exception

test_retry()
```

或者上面两个条件满足一个就结束重试

```
from tenacity import retry, stop_after_delay, stop_after_attempt

@retry(stop=(stop_after_delay(10) | stop_after_attempt(7)))
def test_retry():
```

```
    print("等待重试...")
    raise Exception

test_retry()
```

设置何时进行重试

在出现特定错误/异常（比如请求超时）的情况下，再进行重试

```
from requests import exceptions
from tenacity import retry, retry_if_exception_type

@retry(retry=retry_if_exception_type(exceptions.Timeout))
def test_retry():
    print("等待重试...")
    raise exceptions.Timeout

test_retry()
```

在满足自定义条件时，再进行重试。

如下示例，当 `test_retry` 函数返回值为 False 时，再进行重试

```
from tenacity import retry, stop_after_attempt, retry_if_result

def is_false(value):
    return value is False

@retry(stop=stop_after_attempt(3),
       retry=retry_if_result(is_false))
def test_retry():
    return False

test_retry()
```

重试后错误重新抛出

当出现异常后，tenacity 会进行重试，若重试后还是失败，默认情况下，往上抛出的异常会变成 RetryError，而不是最根本的原因。

因此可以加一个参数（`reraise=True`），使得当重试失败后，往外抛出的异常还是原来的那个。

```
from tenacity import retry, stop_after_attempt

@retry(stop=stop_after_attempt(7), reraise=True)
def test_retry():
    print("等待重试...")
    raise Exception

test_retry()
```

设置回调函数

当最后一次重试失败后，可以执行一个回调函数

```
from tenacity import *

def return_last_value(retry_state):
    print("执行回调函数")
    return retry_state.outcome.result()  # 表示返回原函数的返回值

def is_false(value):
    return value is False

@retry(stop=stop_after_attempt(3),
       retry_error_callback=return_last_value,
       retry=retry_if_result(is_false))
def test_retry():
    print("等待重试中...")
    return False

print(test_retry())
```

输出如下

```
等待重试中...
等待重试中...
等待重试中...
执行回调函数
False
```

5.15 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 ~]$
```

5.16 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']
```

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。

5.17 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。*

> 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。

5.18 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
```


5.19 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。

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
```

5.20 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。

if else 早已烂大街，但我相信仍然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。

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。

再来看看，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。

> 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。

5.21 单分派泛函数如何写?

泛型，如果你尝过java，应该对他不陌生吧。但你可能不知道在 Python 中（3.4+ ），也可以实现 简单的泛型函数。

在Python中只能实现基于单个（第一个）参数的数据类型来选择具体的实现方式，官方名称 是 `single-dispatch` 。你或许听不懂，说人话，就是可以实现第一个参数的数据类型不同，其调用的函数也就不同。

`singledispatch` 是 PEP443 中引入的，如果你对此有兴趣，PEP443 应该是最好的学习文档：https://www.python.org/dev/peps/pep-0443/

它使用方法极其简单，只要被 `singledispatch` 装饰的函数，就是一个 `single-dispatch` 的泛函数（ `generic functions` ）。

- **单分派**：根据一个参数的类型，以不同方式执行相同的操作的行为。
- **多分派**：可根据多个参数的类型选择专门的函数的行为。
- **泛函数**：多个函数绑在一起组合成一个泛函数。

这边举个简单的例子。

```
from functools import singledispatch

@singledispatch
def age(obj):
    print('请传入合法类型的参数！')

@age.register(int)
def _(age):
    print('我已经{}岁了。'.format(age))

@age.register(str)
def _(age):
    print('I am {} years old.'.format(age))


age(23)  # int
age('twenty three')  # str
age(['23'])  # list
```

执行结果

```
我已经23岁了。
I am twenty three years old.
请传入合法类型的参数！
```

说起泛型，其实在 Python 本身的一些内建函数中并不少见，比如 `len()` ， `iter()` ，`copy.copy()` ， `pprint()` 等

你可能会问，它有什么用呢？实际上真没什么用，你不用它或者不认识它也完全不影响你编码。

我这里举个例子，你可以感受一下。

大家都知道，Python 中有许许多的数据类型，比如 str，list， dict， tuple 等，不同数据类型的拼接方式各不相同，所以我这里我写了一个通用的函数，可以根据对应的数据类型对选择对应的拼接方式拼接，而且不同数据类型我还应该提示无法拼接。以下是简单的实现。

```
def check_type(func):
    def wrapper(*args):
        arg1, arg2 = args[:2]
        if type(arg1) != type(arg2):
            return '【错误】：参数类型不同，无法拼接!!'
        return func(*args)
    return wrapper
```

```python
@singledispatch
def add(obj, new_obj):
    raise TypeError

@add.register(str)
@check_type
def _(obj, new_obj):
    obj += new_obj
    return obj


@add.register(list)
@check_type
def _(obj, new_obj):
    obj.extend(new_obj)
    return obj

@add.register(dict)
@check_type
def _(obj, new_obj):
    obj.update(new_obj)
    return obj

@add.register(tuple)
@check_type
def _(obj, new_obj):
    return (*obj, *new_obj)

print(add('hello',', world'))
print(add([1,2,3], [4,5,6]))
print(add({'name': 'wangbm'}, {'age':25}))
print(add(('apple', 'huawei'), ('vivo', 'oppo')))

# list 和 字符串 无法拼接
print(add([1,2,3], '4,5,6'))
```

输出结果如下

```
hello, world
[1, 2, 3, 4, 5, 6]
{'name': 'wangbm', 'age': 25}
('apple', 'huawei', 'vivo', 'oppo')
【错误】: 参数类型不同，无法拼接!!
```

如果不使用singledispatch 的话，你可能会写出这样的代码。

```
def check_type(func):
    def wrapper(*args):
        arg1, arg2 = args[:2]
        if type(arg1) != type(arg2):
            return '【错误】：参数类型不同，无法拼接!!'
        return func(*args)
    return wrapper

@check_type
def add(obj, new_obj):
    if isinstance(obj, str) :
        obj += new_obj
        return obj

    if isinstance(obj, list) :
        obj.extend(new_obj)
        return obj

    if isinstance(obj, dict) :
        obj.update(new_obj)
        return obj

    if isinstance(obj, tuple) :
        return (*obj, *new_obj)

print(add('hello',', world'))
print(add([1,2,3], [4,5,6]))
print(add({'name': 'wangbm'}, {'age':25}))
print(add(('apple', 'huawei'), ('vivo', 'oppo')))

# list 和 字符串 无法拼接
print(add([1,2,3], '4,5,6'))
```

输出如下

```
hello, world
[1, 2, 3, 4, 5, 6]
{'name': 'wangbm', 'age': 25}
('apple', 'huawei', 'vivo', 'oppo')
【错误】：参数类型不同，无法拼接!!
```

第六章：良好编码习惯

6.1 不要直接调用类的私有方法

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()
```

6.2 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）

6.3 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 __add__ 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。

6.4 别再使用 pprint 打印了

1. 吐槽问题

pprint 你应该很熟悉了吧?

随便在搜索引擎上搜索如何打印漂亮的字典或者格式化字符串时，大部分人都会推荐你使用这货。

比如这下面这个 json 字符串或者说字典（我随便在网上找的），如果不格式化美化一下，根本无法阅读。

```
[{"id":1580615,"name":"皮的嘛","packageName":"com.renren.mobile.android","iconUrl":"app/com.renren.mobile.android/icon.jpg","stars":2,"size":21803987,"downloadUrl":"app/com.renren.mobile.android/com.renren.mobile.android.apk","des":"2011-2017 你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青"},{"id":1540629,"name":"不存在的","packageName":"com.ct.client","iconUrl":"app/com.ct.client/icon.jpg","stars":2,"size":4794202,"downloadUrl":"app/com.ct.client/com.ct.client.apk","des":"斗鱼271934 走过路过不要错过，这里有最好的鸡儿"}]
```

如果你不想看到一堆密密麻麻的字，那就使用大伙都极力推荐的 pprint 看下什么效果（以下在 Python 2 中演示，Python 3 中是不一样的效果）。

```
>>> info=[{"id":1580615,"name":"皮的嘛","packageName":"com.renren.mobile.android","iconUrl":"app/com.renren.mobile.android/icon.jpg","stars":2,"size":21803987,"downloadUrl":"app/com.renren.mobile.android/com.renren.mobile.android.apk","des":"2011-2017 你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青"},{"id":1540629,"name":"不存在的","packageName":"com.ct.client","iconUrl":"app/com.ct.client/icon.jpg","stars":2,"size":4794202,"downloadUrl":"app/com.ct.client/com.ct.client.apk","des":"斗鱼271934 走过路过不要错过，这里有最好的鸡儿"}]
>>>
>>> from pprint import pprint
>>> pprint(info)
[{'des': '2011-2017 \xe4\xbd\xa0\xe7\x9a\x84\xe9\x93\x81\xe5\xa4\xb4\xe5\xa8\x83\xe4\xb8\x80\xe7\x9b\xb4\xe5\x9c\xa8\xe8\xbf\x99\xe5\x84\xbf\xe3\x80\x82\xe4\xb8\xad\xe5\x9b\xbd\xe6\x9c\x80\xe5\xa4\xa7\xe7\x9a\x84\xe5\xae\x9e\xe5\x90\x8d\xe5\x88\xb6SNS\xe7\xbd\x91\xe7\xbb\x9c\xe5\xb9\xb3\xe5\x8f\xb0\xef\xbc\x8c\xe5\xab\xa9\xe5\xa4\xb4\xe9\x9d\x92',
  'downloadUrl': 'app/com.renren.mobile.android/com.renren.mobile.android.apk',
  'iconUrl': 'app/com.renren.mobile.android/icon.jpg',
  'id': 1580615,
  'name': '\xe7\x9a\xae\xe7\x9a\x84\xe5\x98\x9b',
  'packageName': 'com.renren.mobile.android',
  'size': 21803987,
  'stars': 2},
 {'des': '\xe6\x96\x97\xe9\xb1\xbc271934 \xe8\xb5\xb0\xe8\xbf\x87\xe8\xb7\xaf\xe8\xbf\x87\xe4\xb8\x8d\xe8\xa6\x81\xe9\x94\x99\xe8\xbf\x87\xef\xbc\x8c\xe8\xbf\x99\xe9\x87\x8c\xe6\x9c\x89\xe6\x9c\x80\xe5\xa5\xbd\xe7\x9a\x84\xe9\xb8\xa1\xe5\x84\xbf',
  'downloadUrl': 'app/com.ct.client/com.ct.client.apk',
  'iconUrl': 'app/com.ct.client/icon.jpg',
  'id': 1540629,
  'name': '\xe4\xb8\x8d\xe5\xad\x98\xe5\x9c\xa8\xe7\x9a\x84',
```

```
  'packageName': 'com.ct.client',
  'size': 4794202,
  'stars': 2}]
```

好像有点效果，真的是“神器”呀。

但是你告诉我， **\xe4\xbd\xa0\xe7\x9a** 这些是什么玩意？本来想提高可读性的，现在变成完全不可读了。

好在我懂点 Python 2 的编码，知道 Python 2 中默认（不带u）的字符串格式都是 str 类型，也是 bytes 类型，它是以 byte 存储的。

行吧，好像是我错了，我改了下，使用 unicode 类型来定义中文字符串吧。

```
>>> info = [{"id":1580615,"name":u"皮的嘛","packageName":"com.renren.mobile.andr
oid","iconUrl":"app/com.renren.mobile.android/icon.jpg","stars":2,"size":218039
87,"downloadUrl":"app/com.renren.mobile.android/com.renren.mobile.android.apk",
"des":u"2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青"},{"id":15406
29,"name":u"不存在的","packageName":"com.ct.client","iconUrl":"app/com.ct.client
/icon.jpg","stars":2,"size":4794202,"downloadUrl":"app/com.ct.client/com.ct.cli
ent.apk","des":u"斗鱼271934走过路过不要错过，这里有最好的鸡儿"}]
>>>
>>> from pprint import pprint
>>> pprint(info)
[{'des': u'2011-2017\u4f60\u7684\u94c1\u5934\u5a03\u4e00\u76f4\u5728\u8fd9\u513
f\u3002\u4e2d\u56fd\u6700\u5927\u7684\u5b9e\u540d\u5236SNS\u7f51\u7edc\u5e73\u5
3f0\uff0c\u5ae9\u5934\u9752',
  'downloadUrl': 'app/com.renren.mobile.android/com.renren.mobile.android.apk',
  'iconUrl': 'app/com.renren.mobile.android/icon.jpg',
  'id': 1580615,
  'name': u'\u76ae\u7684\u561b',
  'packageName': 'com.renren.mobile.android',
  'size': 21803987,
  'stars': 2},
 {'des': u'\u6597\u9c7c271934\u8d70\u8fc7\u8def\u8fc7\u4e0d\u8981\u9519\u8fc7\u
ff0c\u8fd9\u91cc\u6709\u6700\u597d\u7684\u9e21\u513f',
  'downloadUrl': 'app/com.ct.client/com.ct.client.apk',
  'iconUrl': 'app/com.ct.client/icon.jpg',
  'id': 1540629,
  'name': u'\u4e0d\u5b58\u5728\u7684',
  'packageName': 'com.ct.client',
  'size': 4794202,
  'stars': 2}]
```

确实是有好点了，但是看到下面这些，我崩溃了，我哪里知道这是什么鬼，难道是我太菜了吗？当我是计算机呀？

```
u'\u6597\u9c7c271934\u8d70\u8fc7\u8def\u8fc7\u4e0d\u8981\u9519\u8fc7\uff0c\u8fd9\u91cc\u6709\u6700\u597d\u7684\u9e21\u513f'
```

除此之外，我们知道 json 的严格要求必须使用 **双引号**，而我定义字典时，也使用了双引号了，为什么打印出来的为什么是 **单引号**？我也太难了吧，我连自己的代码都无法控制了吗？

到这里，我们知道了 pprint 带来的两个问题：

1. 没法在 Python 2 下正常打印中文
2. 没法输出 JSON 标准格式的格式化内容（双引号）

2. 解决问题

打印中文

如果你是在 Python 3 下使用，你会发现中文是可以正常显示的。

```
# Python3.7
>>> info = [{"id":1580615,"name":u"皮的嘛","packageName":"com.renren.mobile.android","iconUrl":"app/com.renren.mobile.android/icon.jpg","stars":2,"size":21803987,"downloadUrl":"app/com.renren.mobile.android/com.renren.mobile.android.apk","des":u"2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青"},{"id":1540629,"name":u"不存在的","packageName":"com.ct.client","iconUrl":"app/com.ct.client/icon.jpg","stars":2,"size":4794202,"downloadUrl":"app/com.ct.client/com.ct.client.apk","des":u"斗鱼271934走过路过不要错过，这里有最好的鸡儿"}]
>>>
>>> from pprint import pprint
>>> pprint(info)
[{'des': '2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青',
  'downloadUrl': 'app/com.renren.mobile.android/com.renren.mobile.android.apk',
  'iconUrl': 'app/com.renren.mobile.android/icon.jpg',
  'id': 1580615,
  'name': '皮的嘛',
  'packageName': 'com.renren.mobile.android',
  'size': 21803987,
  'stars': 2},
 {'des': '斗鱼271934走过路过不要错过，这里有最好的鸡儿',
  'downloadUrl': 'app/com.ct.client/com.ct.client.apk',
  'iconUrl': 'app/com.ct.client/icon.jpg',
  'id': 1540629,
```

```
  'name': '不存在的',
  'packageName': 'com.ct.client',
  'size': 4794202,
  'stars': 2}]
>>>
```

但是很多时候（在公司的一些服务器）你无法选择自己使用哪个版本的 Python，本来我可以选择不用的，因为有更好的替代方案（**这个后面会讲**）。

但是我出于猎奇，正好前两天不是写过一篇关于 编码 的文章吗，我自认为自己对于 编码还是掌握比较熟练的，就想着来解决一下这个问题。

索性就来看下 pprint 的源代码，还真被我找到了解决方法，如果你也想挑战一下，不防在这里停住，自己研究一下如何实现，我相信对你阅读源码会有帮助。

以下是我的解决方案，供你参考：

写一个自己的 printer 对象，继承自 PrettyPrinter （pprint 使用的printer）

并且复写 format 方法，判断传进来的字符串对象是否 str 类型，如果不是 str 类型，而是 unicode 类型，就用 uft8 编码成 str 类型。

```
# coding: utf-8
from pprint import PrettyPrinter

# 继承 PrettyPrinter，复写 format 方法
class MyPrettyPrinter(PrettyPrinter):
    def format(self, object, context, maxlevels, level):
        if isinstance(object, unicode):
            return (object.encode('utf8'), True, False)
        return PrettyPrinter.format(self, object, context, maxlevels, level)

info = [{"id":1580615,"name":u"皮的嘛","packageName":"com.renren.mobile.android"
,"iconUrl":"app/com.renren.mobile.android/icon.jpg","stars":2,"size":21803987,"
downloadUrl":"app/com.renren.mobile.android/com.renren.mobile.android.apk","des
":u"2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青"},{"id":1540629,"
name":u"不存在的","packageName":"com.ct.client","iconUrl":"app/com.ct.client/ico
n.jpg","stars":2,"size":4794202,"downloadUrl":"app/com.ct.client/com.ct.client.
apk","des":u"斗鱼271934走过路过不要错过，这里有最好的鸡儿"}]

MyPrettyPrinter().pprint(info)
```

输出如下，已经解决了中文的显示问题：

```
[wangbm@35ha01 ~]$ python2 demo.py
[{'des': 2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青,
  'downloadUrl': 'app/com.renren.mobile.android/com.renren.mobile.android.apk',
  'iconUrl': 'app/com.renren.mobile.android/icon.jpg',
  'id': 1580615,
  'name': 皮的嘛,
  'packageName': 'com.renren.mobile.android',
  'size': 21803987,
  'stars': 2},
 {'des': 斗鱼271934走过路过不要错过，这里有最好的鸡儿,
  'downloadUrl': 'app/com.ct.client/com.ct.client.apk',
  'iconUrl': 'app/com.ct.client/icon.jpg',
  'id': 1540629,
  'name': 不存在的,
  'packageName': 'com.ct.client',
  'size': 4794202,
  'stars': 2}]
[wangbm@35ha01 ~]$
[wangbm@35ha01 ~]$
```

打印双引号

解决了中文问题后，再来看看如何让 pprint 打印双引号。

在实例化 PrettyPrinter 对象的时候，可以接收一个 stream 对象，它表示你要将内容输出到哪里，默认是使用 sys.stdout 这个 stream，也就是标准输出。

现在我们要修改输出的内容，也就是将输出的单引号替换成双引号。

那我们完全可以自己定义一个 stream 类型的对象，该对象不需要继承任何父类，只要你实现 write 方法就可以。

有了思路，就可以开始写代码了，如下：

```
# coding: utf-8
from pprint import PrettyPrinter

class MyPrettyPrinter(PrettyPrinter):
    def format(self, object, context, maxlevels, level):
        if isinstance(object, unicode):
            return (object.encode('utf8'), True, False)
        return PrettyPrinter.format(self, object, context, maxlevels, level)

class MyStream():
    def write(self, text):
        print text.replace('\'', '"')

info = [{"id":1580615,"name":u"皮的嘛","packageName":"com.renren.mobile.android"
,"iconUrl":"app/com.renren.mobile.android/icon.jpg","stars":2,"size":21803987,"
```

```
downloadUrl":"app/com.renren.mobile.android/com.renren.mobile.android.apk","des
":u"2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青"},{"id":1540629,"
name":u"不存在的","packageName":"com.ct.client","iconUrl":"app/com.ct.client/ico
n.jpg","stars":2,"size":4794202,"downloadUrl":"app/com.ct.client/com.ct.client.
apk","des":u"斗鱼271934走过路过不要错过，这里有最好的鸡儿"}]
MyPrettyPrinter(stream=MyStream()).pprint(info)
```

尝试执行了下，我的天，怎么是这样子的。

```
[
{
"des"
:
2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青
,
  "downloadUrl":
"app/com.renren.mobile.android/com.renren.mobile.android.apk"
,
  "iconUrl":
"app/com.renren.mobile.android/icon.jpg"
,
  "id":
1580615
,
  "name":
皮的嘛
,
  "packageName":
"com.renren.mobile.android"
,
  "size":
21803987
,
  "stars":
2
}
,

{
"des"
:
斗鱼271934走过路过不要错过，这里有最好的鸡儿
,
  "downloadUrl":
"app/com.ct.client/com.ct.client.apk"
```

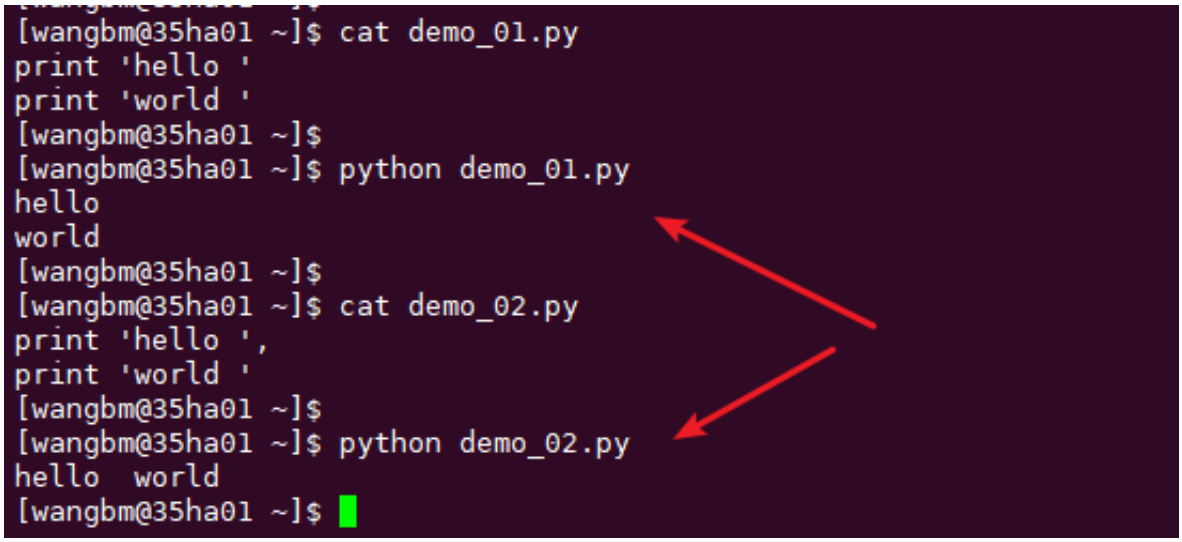
```
,
  "iconUrl":
"app/com.ct.client/icon.jpg"
,
  "id":
1540629
,
  "name":
不存在的
,
  "packageName":
"com.ct.client"
,
  "size":
4794202
,
  "stars":
2
}
]
```

经过一番研究，才知道是因为 print 函数默认会将打印的内容后面加个 **换行符**。

那如何将使 print 函数打印的内容，不进行换行呢?

方法很简单，但是我相信很多人都不知道，只要在 print 的内容后加一个 **逗号** 就行。

就像下面这样。



知道了问题所在，再修改下代码

```
# coding: utf-8
```

```
from pprint import PrettyPrinter

class MyPrettyPrinter(PrettyPrinter):
    def format(self, object, context, maxlevels, level):
        if isinstance(object, unicode):
            return (object.encode('utf8'), True, False)
        return PrettyPrinter.format(self, object, context, maxlevels, level)

class MyStream():
    def write(self, text):
        print text.replace('\'', '"'),

info = [{"id":1580615,"name":u"皮的嘛","packageName":"com.renren.mobile.android"
,"iconUrl":"app/com.renren.mobile.android/icon.jpg","stars":2,"size":21803987,"
downloadUrl":"app/com.renren.mobile.android/com.renren.mobile.android.apk","des
":u"2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青"},{"id":1540629,"
name":u"不存在的","packageName":"com.ct.client","iconUrl":"app/com.ct.client/ico
n.jpg","stars":2,"size":4794202,"downloadUrl":"app/com.ct.client/com.ct.client.
apk","des":u"斗鱼271934走过路过不要错过，这里有最好的鸡儿"}]

MyPrettyPrinter(stream=MyStream()).pprint(info)
```

终于成功了，太不容易了吧。

```
[wangbm@35ha01 ~]$ python demo.py
[ { "des" :  2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青 ,
  "downloadUrl":  "app/com.renren.mobile.android/com.renren.mobile.android.apk" ,
  "iconUrl":  "app/com.renren.mobile.android/icon.jpg" ,
  "id":  1580615 ,
  "name":  皮的嘛 ,
  "packageName":  "com.renren.mobile.android" ,
  "size":  21803987 ,
  "stars":  2 } ,
  { "des" :  斗鱼271934走过路过不要错过，这里有最好的鸡儿 ,
  "downloadUrl":  "app/com.ct.client/com.ct.client.apk" ,
  "iconUrl":  "app/com.ct.client/icon.jpg" ,
  "id":  1540629 ,
  "name":  不存在的 ,
  "packageName":  "com.ct.client" ,
  "size":  4794202 ,
  "stars":  2 } ]
[wangbm@35ha01 ~]$
```

3. 何必折腾

通过上面的一番折腾，我终于实现了我 **梦寐以求** 的需求。

代价就是我整整花费了两个小时，才得以实现，而对于小白来说，可能没有信心，也没有耐心去做这样的事情。

所以我想说的是，Python 2 下的 pprint ，真的不要再用了。

为什么我要用这么 说，因为明明有更好的替代品，人生苦短，既然用了 Python ， 当然是怎么简单怎么来咯，何必为难自己呢，一行代码可以解决的事情，偏偏要去写两个类，那不是自讨苦吃吗？ （我这是在骂自己吗？

如果你愿意抛弃 pprint ， 那我推荐你用 json.dumps ， 我保证你再也不想用 pprint 了。

打印中文

其实无法打印中文，是 Python 2 引来的大坑，并不能全怪 pprint 。

但是同样的问题，在 json.dumps 这里，却只要加个参数就好了，可比 pprint 简单得不要太多。

具体的代码示例如下：

```
>>> info = [{"id":1580615,"name":"皮的嘛","packageName":"com.renren.mobile.andro
id","iconUrl":"app/com.renren.mobile.android/icon.jpg","stars":2,"size":2180398
7,"downloadUrl":"app/com.renren.mobile.android/com.renren.mobile.android.apk","
des":"2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青"},{"id":1540629
,"name":"不存在的","packageName":"com.ct.client","iconUrl":"app/com.ct.client/ic
on.jpg","stars":2,"size":4794202,"downloadUrl":"app/com.ct.client/com.ct.client
.apk","des":"斗鱼271934走过路过不要错过，这里有最好的鸡儿"}]
>>>
>>> import json
>>>
>>>
>>> print json.dumps(info, indent=4, ensure_ascii=False)
[
    {
        "downloadUrl": "app/com.renren.mobile.android/com.renren.mobile.android
.apk",
        "iconUrl": "app/com.renren.mobile.android/icon.jpg",
        "name": "皮的嘛",
        "stars": 2,
        "packageName": "com.renren.mobile.android",
        "des": "2011-2017你的铁头娃一直在这儿。中国最大的实名制SNS网络平台，嫩头青",
        "id": 1580615,
        "size": 21803987
    },
    {
        "downloadUrl": "app/com.ct.client/com.ct.client.apk",
        "iconUrl": "app/com.ct.client/icon.jpg",
        "name": "不存在的",
        "stars": 2,
```

```
        "packageName": "com.ct.client",
        "des": "斗鱼271934走过路过不要错过，这里有最好的鸡儿",
        "id": 1540629,
        "size": 4794202
    }
]
>>>
```

json.dumps 的关键参数有两个：

- **indent=4**：以 4 个空格缩进单位
- **ensure_ascii=False**：接收非 ASCII 编码的字符，这样才能使用中文

与 pprint 相比 json.dumps 可以说完胜：

1. 两个参数就能实现所有我的需求（打印中文与双引号）
2. 就算在 Python 2 下，使用中文也不需要用 `u'中文'` 这种写法
3. Python2 和 Python3 的写法完全一致，对于这一点不需要考虑兼容问题

4. 总结一下

本来很简单的一个观点，我为了证明 pprint 实现那两个需求有多么困难，花了很多的时间去研究了 pprint 的源码（各种处理其实还是挺复杂的），不过好在最后也能有所收获。

本文的分享就到这里，阅读本文，我认为你可以获取到三个知识点

1. 核心观点：Python2 下不要再使用 pprint
2. 若真要使用，且有和一样的改造需求，可以参考我的实现
3. Python 2 中的 print 语句后居然可以加 逗号

第七章：神奇的魔法模块

7.1 远程登陆服务器的最佳利器

在使用 Python 写一些脚本的时候，在某些情况下，我们需要频繁登陆远程服务去执行一次命令，并返回一些结果。

在 shell 环境中，我们是这样子做的。

```
$ sshpass -p ${passwd} ssh -p ${port} -l ${user} -o StrictHostKeyChecking=no xx
.xx.xx.xx "ls -l"
```

然后你会发现，你的输出有很多你并不需要，但是又不去不掉的一些信息（也许有方法，请留言交流），类似这样

```
host: xx.xx.xx.xx, port: xx
Warning: Permanently added '[xx.xx.xx.xx]:xx' (RSA) to the list of known hosts.
Login failure: [Errno 1] This server is not registered to rmp platform, please
confirm whether cdn server.
total 4
-rw-r--r-- 1 root root 239 Mar 30  2018 admin-openrc
```

对于直接使用 shell 命令，来执行命令的，可以直接使用管道，或者将标准输出重定向到文件的方法取得执行命令返回的结果

1. 使用 subprocess

若是使用 Python 来做这件事，通常我们会第一时间，想到使用 os.popen，os.system，commands，subprocess 等一些命令执行库来间接获取 。

但是据我所知，这些库获取的 output 不仅只有标准输出，还包含标准错误（也就是上面那些多余的信息）

所以每次都要对 output 进行的数据清洗，然后整理格式化，才能得到我们想要的数据。

用 subprocess 举个例子，就像这样子

```
import subprocess
ssh_cmd = "sshpass -p ${passwd} ssh -p 22 -l root -o StrictHostKeyChecking=no x
x.xx.xx.xx  'ls -l'"
status, output = subprocess.getstatusoutput(ssh_cmd)

# 数据清理，格式化的就不展示了
<code...>
```

通过以上的文字 + 代码的展示 ， 可以感觉到 ssh 登陆的几大痛点

- **痛点一**：需要额外安装 sshpass（如果不免密的话）
- **痛点二**：干扰信息太多，数据清理、格式化相当麻烦

- **痛点三**：代码实现不够优雅（有点土），可读性太差
- **痛点四**：ssh 连接不能复用，一次连接仅能执行一次
- **痛点五**：代码无法全平台，仅能在 Linux 和 OSX 上使用

为了解决这几个问题，我搜索了全网关于 Python ssh 的文章，没有看到有完整介绍这方面的技巧的。

> 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。

为此，我就翻阅了一个很火的 Github 项目： awesome-python-cn（https://github.com/BingmingWong/awesome-python-cn）。

期望在这里，找到有一些关于 远程连接 的一些好用的库。

还真的被我找到了两个

- sh.ssh
- Paramiko

2. 使用 sh.ssh

首先来介绍第一个， `sh.ssh`

`sh` 是一个可以让你通过函数的调用来完成 Linxu/OSX 系统命令的一个库，非常好用，关于它有机会也写篇介绍。

```
$ python3 -m pip install sh
```

今天只介绍它其中的一个函数： `ssh`

通常两台机器互访，为了方便，可设置免密登陆，这样就不需要输入密码。

这段代码可以实现免密登陆，并执行我们的命令 `ls -l`

```
from sh import ssh
output=ssh("root@xx.xx.xx.xx", "-p 22", "ls -l")
print(output)
```

但有可能 ，我们并不想设置互信免密，为了使这段代码更通用，我假定我们没有设置免密，只能

使用密码进行登陆。

问题就来了，要输入密码，必须得使用交互式的方法来输入呀，在 Python 中要如何实现呢?

原来 ssh 方法接收一个 `_out` 参数，这个参数可以为一个字符串，表示文件路径，也可以是一个文件对象（或者类文件对象），还可以是一个回调函数，意思是当有标准输出时，就会调用将输出内容传给这个函数。

这就好办了呀。

我只要识别到有 `password:` 字样，就往标准输入写入我的密码就好了呀。

完整代码如下：

```
import sys
from sh import ssh

aggregated = ""
def ssh_interact(char, stdin):
    global aggregated
    sys.stdout.write(char.encode())
    sys.stdout.flush()
    aggregated += char
    if aggregated.endswith("password: "):
        stdin.put("you_password\n")

output=ssh("root@xx.xx.xx.xx", "-p 22", "ls -l",_tty_in=True, _out_bufsize=0, _
out=ssh_interact)
print(output)
```

这是官方文档（http://amoffat.github.io/sh/tutorials/interacting_with_processes.html?highlight=ssh）给的一些信息，写的一个demo。

尝试运行后，发现程序会一直在运行中，永远不会返回，不会退出，回调函数也永远不会进入。

通过调试查看源代码，仍然查不到问题所在，于是去 Github 上搜了下，原来在 2017 年就已经存在这个问题了，到现在 2020 年了还没有修复，看来使用 `sh.ssh` 的人并不多，于是我又“追问”了下，期望能得到回复。

ssh callback example is broken under v1.12 #393

Open acceleratorguy opened this issue on 22 Jun 2017 · 3 comments

acceleratorguy commented on 22 Jun 2017

The ssh example, found at http://amoffat.github.io/sh/tutorials/interacting_with_processes.html#, works under v1.11 but doesn't work under v1.12.
Symptoms:
The the callback function, ssh_interact, is never called.

Environment:
Python 2.7.5
sh (1.12.14)
Underlying OS: Redhat 7.3 derivative.

amoffat commented on 22 Jun 2017 Owner

Thanks for bringing this to my attention **@acceleratorguy**. I will look into it

tkossak commented on 30 Jun 2017

same problem here (sh 1.12.14, Python 3.6.1, Linux Mint)

BingmingWong commented 1 minute ago

> Thanks for bringing this to my attention **@acceleratorguy**. I will look into it

I have the same problem（Python 2.7.5 and Python 3.7.1, CentOS 7.2）, did you solve it?

以上这个问题，只有在需要输入密码才会出现，如果设置了机器互信是没有问题的。

为了感受 `sh.ssh` 的使用效果，我设置了机器互信免密，然后使用如下这段代码。

```
from sh import ssh

my_server=ssh.bake("root@xx.xx.xx.xx", "-p 22")

# 相当于执行登陆一次执行一次命令，执行完就退出登陆
print(my_server.ls())
```

```
# 可在 sleep 期间，手动登陆服务器，使用 top ，查看当前有多少终端在连接
time.sleep(5)

# 再次执行这条命令时，登陆终端数将 +1，执行完后，又将 -1
print(my_server.ifconfig())
```

惊奇地发现使用 `bake` 这种方式， `my_server.ls()` 和 `my_server.ifconfig()` 这种看似是通过同一个ssh连接，执行两次命令，可实际上，你可以在远程机器上，执行 top 命令看到已连接的终端的变化，会先 `+1` 再 `-1` ，说明两次命令的执行是通过两次连接实现的。

如此看来，使用 `sh.ssh` 可以解决痛点一（如果上述问题能得到解决）、痛点二、痛点三。

但是它仍然无法复用 ssh 连接，还是不太方便，不是我理想中的最佳方案。

最重要的一点是， `sh` 这个模块，仅支持 Linxu/OSX ，在 Windows 你得使用它的兄弟库 – `pbs` ，然后我又去 pypi 看了一眼 pbs，已经“年久失修”，没人维护了。

至此，我离“卒”，就差最后一根稻草了。

> 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。

3. 使用 paramiko

带着最后一丝希望，我尝试使用了 `paramiko` 这个库，终于在 `paramiko` 这里，找回了本应属于 Python 的那种优雅。

你可以通过如下命令去安装它

```
$ python3 -m pip install paramiko
```

然后接下来，就介绍几种常用的 ssh 登陆的方法

方法1：基于用户名和密码的 sshclient 方式登录

然后你可以参考如下这段代码，在 Linux/OSX 系统下进行远程连接

```
import paramiko

ssh = paramiko.SSHClient()
# 允许连接不在know_hosts文件中的主机
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

# 建立连接
ssh.connect("xx.xx.xx.xx", username="root", port=22, password="you_password")

# 使用这个连接执行命令
ssh_stdin, ssh_stdout, ssh_stderr = ssh.exec_command("ls -l")

# 获取输出
print(ssh_stdout.read())

# 关闭连接
ssh.close()
```

方法2：基于用户名和密码的 transport 方式登录

方法1 是传统的连接服务器、执行命令、关闭的一个操作，多个操作需要连接多次，无法复用连接[**痛点四**]。

有时候需要登录上服务器执行多个操作，比如执行命令、上传/下载文件，方法1 则无法实现，那就可以使用 transport 的方法。

```
import paramiko

# 建立连接
trans = paramiko.Transport(("xx.xx.xx.xx", 22))
trans.connect(username="root", password="you_passwd")

# 将sshclient的对象的transport指定为以上的trans
ssh = paramiko.SSHClient()
ssh._transport = trans

# 剩下的就和上面一样了
```

```
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh_stdin, ssh_stdout, ssh_stderr = ssh.exec_command("ls -l")
print(ssh_stdout.read())

# 关闭连接
trans.close()
```

方法3：基于公钥密钥的 SSHClient 方式登录

```
import paramiko

# 指定本地的RSA私钥文件
# 如果建立密钥对时设置的有密码，password为设定的密码，如无不用指定password参数
pkey = paramiko.RSAKey.from_private_key_file('/home/you_username/.ssh/id_rsa',
password='12345')

# 建立连接
ssh = paramiko.SSHClient()
ssh.connect(hostname='xx.xx.xx.xx',
            port=22,
            username='you_username',
            pkey=pkey)

# 执行命令
stdin, stdout, stderr = ssh.exec_command('ls -l')

# 结果放到stdout中，如果有错误将放到stderr中
print(stdout.read())

# 关闭连接
ssh.close()
```

方法4：基于密钥的 Transport 方式登录

```
import paramiko

# 指定本地的RSA私钥文件
# 如果建立密钥对时设置的有密码，password为设定的密码，如无不用指定password参数
pkey = paramiko.RSAKey.from_private_key_file('/home/you_username/.ssh/id_rsa',
password='12345')

# 建立连接
trans = paramiko.Transport(('xx.xx.xx.xx', 22))
```

```
trans.connect(username='you_username', pkey=pkey)

# 将sshclient的对象的transport指定为以上的trans
ssh = paramiko.SSHClient()
ssh._transport = trans

# 执行命令，和传统方法一样
stdin, stdout, stderr = ssh.exec_command('df -hl')
print(stdout.read().decode())

# 关闭连接
trans.close()
```

以上四种方法，可以帮助你实现远程登陆服务器执行命令，如果需要复用连接：一次连接执行多次命令，可以使用 **方法二** 和 **方法四**

用完后，记得关闭连接。

实现 sftp 文件传输

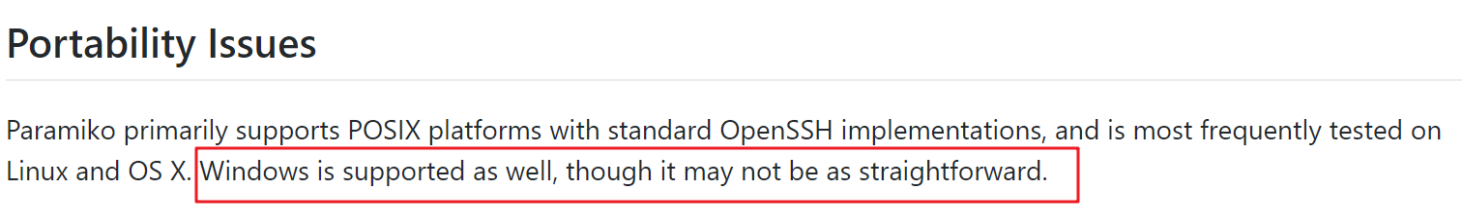
同时，paramiko 做为 ssh 的完美解决方案，它非常专业，利用它还可以实现 sftp 文件传输。

```
import paramiko

# 实例化一个trans对象# 实例化一个transport对象
trans = paramiko.Transport(('xx.xx.xx.xx', 22))

# 建立连接
trans.connect(username='you_username', password='you_passwd')

# 实例化一个 sftp对象,指定连接的通道
sftp = paramiko.SFTPClient.from_transport(trans)

# 发送文件
sftp.put(localpath='/tmp/11.txt', remotepath='/tmp/22.txt')

# 下载文件
sftp.get(remotepath='/tmp/22.txt', localpath='/tmp/33.txt')
trans.close()
```

到这里，Paramiko 已经完胜了，但是仍然有一个痛点我们没有提及，就是多平台，说的就是 Windows，这里就有一件好事，一件坏事了，。

好事就是：paramiko 支持 windows

坏事就是：你需要做很多复杂的准备，你可 google 解决，但是我建议你直接放弃，坑太深了。



注意事项

使用 paramiko 的时候，有一点需要注意一下，这个也是我自己 "踩坑" 后才发现的，其实我觉得这个设计挺好的，如果你不需要等待它返回数据，可以直接实现异步效果，只不过对于不知道这个设计的人，确实是个容易掉坑的点

就是在执行 `ssh.exec_command(cmd)` 时，这个命令并不是同步阻塞的。

比如下面这段代码，执行时，你会发现 脚本立马就结束退出了，并不会等待 5 s 后，再 执行 ssh.close()

```
import paramiko

trans = paramiko.Transport(("172.20.42.1", 57891))
trans.connect(username="root", password="youpassword")
ssh = paramiko.SSHClient()
ssh._transport = trans
stdin, stdout, stderr = ssh.exec_command("sleep 5;echo ok")
ssh.close()
```

但是如果改成这样，加上一行 stdout.read()， paramiko 就知道，你需要这个执行的结果，就会在 read() 进行阻塞。

```
import paramiko

trans = paramiko.Transport(("172.20.42.1", 57891))
trans.connect(username="root", password="youpassword")
ssh = paramiko.SSHClient()
ssh._transport = trans
stdin, stdout, stderr = ssh.exec_command("sleep 5;echo ok")

# 加上一行 read()
print(stdout.read())
ssh.close()
```

4. 写在最后

经过了一番对比，和一些实例的展示，可以看出 Paramiko 是一个专业、让人省心的 ssh 利器，个人认为 Paramiko 模块是运维人员必学模块之一，如果你恰好需要在 Python 代码中实现 ssh 到远程服务器去获取一些信息，那么我把 Paramiko 推荐给你。

当我们写的一个脚本或程序发生各种不可预知的异常时，如果我们没有进行捕获处理的时候，通常都会致使程序崩溃退出，并且会在终端打印出一堆 **密密麻麻** 的 traceback 堆栈信息来告诉我们，是哪个地方出了问题。

就像这样子，天呐，密集恐惧症要犯了都

```
Traceback (most recent call last):
  File "C:\Python27\Scripts\pserve-script.py", line 9, in <module>
    load_entry_point('pyramid==1.4b1', 'console_scripts', 'pserve')()
  File "C:\Python27\lib\site-packages\pyramid-1.4b1-py2.7.egg\pyramid\scripts\pserve.py", line 50, in main
    return command.run()
  File "C:\Python27\lib\site-packages\pyramid-1.4b1-py2.7.egg\pyramid\scripts\pserve.py", line 301, in run
    relative_to=base, global_conf=vars)
  File "C:\Python27\lib\site-packages\pyramid-1.4b1-py2.7.egg\pyramid\scripts\pserve.py", line 332, in loadserver
    server_spec, name=name, relative_to=relative_to, **kw)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 255, in loadserver
    return loadobj(SERVER, uri, name=name, **kw)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 271, in loadobj
    global_conf=global_conf)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 296, in loadcontext
    global_conf=global_conf)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 320, in _loadconfig
    return loader.get_context(object_type, name, global_conf)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 454, in get_context
    section)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 476, in _context_from_use
    object_type, name=use, global_conf=global_conf)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 406, in get_context
    global_conf=global_conf)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 296, in loadcontext
    global_conf=global_conf)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 328, in _loadegg
    return loader.get_context(object_type, name, global_conf)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 620, in get_context
    object_type, name=name)
  File "C:\Python27\lib\site-packages\pastedeploy-1.5.0-py2.7.egg\paste\deploy\loadwsgi.py", line 640, in find_egg_entry_point
    pkg_resources.require(self.spec)
  File "C:\Python27\lib\site-packages\distribute-0.6.32-py2.7.egg\pkg_resources.py", line 690, in require
    needed = self.resolve(parse_requirements(requirements))
  File "C:\Python27\lib\site-packages\distribute-0.6.32-py2.7.egg\pkg_resources.py", line 588, in resolve
    raise DistributionNotFound(req)
pkg_resources.DistributionNotFound: Paste
```

上面这段 traceback

- 只有黑白两个颜色，无法像代码高亮那样，对肉眼实现太不友好了
- 无法直接显示报错的代码，排查问题慢人一步，效率太低

那有没有一种办法，可以解决这些问题呢?

当然有了，在 Python 中，没有什么问题是一个库解决不了的，如果有，那就等你去开发这个库。

今天要介绍的这个库呢，叫做 `pretty-errors` ，从名字上就可以知道它的用途，是用来美化错误信息的。

通过这条命令你可以安装它

```
$ python3 -m pip install pretty-errors
```

> 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。

7.2 代码 BUG 变得酷炫的利器

1. 环境要求

由于使用了 `pretty-errors` 后，你的 traceback 信息输出，会有代码高亮那样的效果，因此当你在使用测试使用 `pretty-error` 时，请确保你使用的终端可以输出带有颜色的字体。

在 windows 上你可以使用 Powershell，cmder 等

在 Mac 上你可以使用自带的终端，或者安装一个更好用的 iTerm2

2. 效果对比

随便写一个没有使用 pretty-errors ，并且报错了的程序，是这样子的。

```
~ on ?master! 21:26:48
$ cat mytest.py

def foo():
    1/0

if __name__ == "__main__":
    foo()

~ on ?master! 21:26:51
$ python3 mytest.py
Traceback (most recent call last):
  File "mytest.py", line 6, in <module>
    foo()
  File "mytest.py", line 3, in foo
    1/0
ZeroDivisionError: division by zero
```

而使用了 pretty_errors 后，报错信息被美化成这样了。

```
~ on ?master! 21:33:44
$ python mytest.py

------------------------------------------------------------------------------

mytest.py 6 <module>
foo()

mytest.py 3 foo
1/0

ZeroDivisionError:
division by zero
```

是不是感觉清楚了不少，那种密密麻麻带来的焦虑感是不是都消失了呢?

当然这段代码少，你可能还没感受到，那就来看下 该项目在 Github上的一张效果对比图吧

3. 配置全局可用

可以看到使用了 pretty_errors 后，无非就是把过滤掉了一些干扰我们视线的无用信息，然后把有用的关键信息给我们高亮显示。

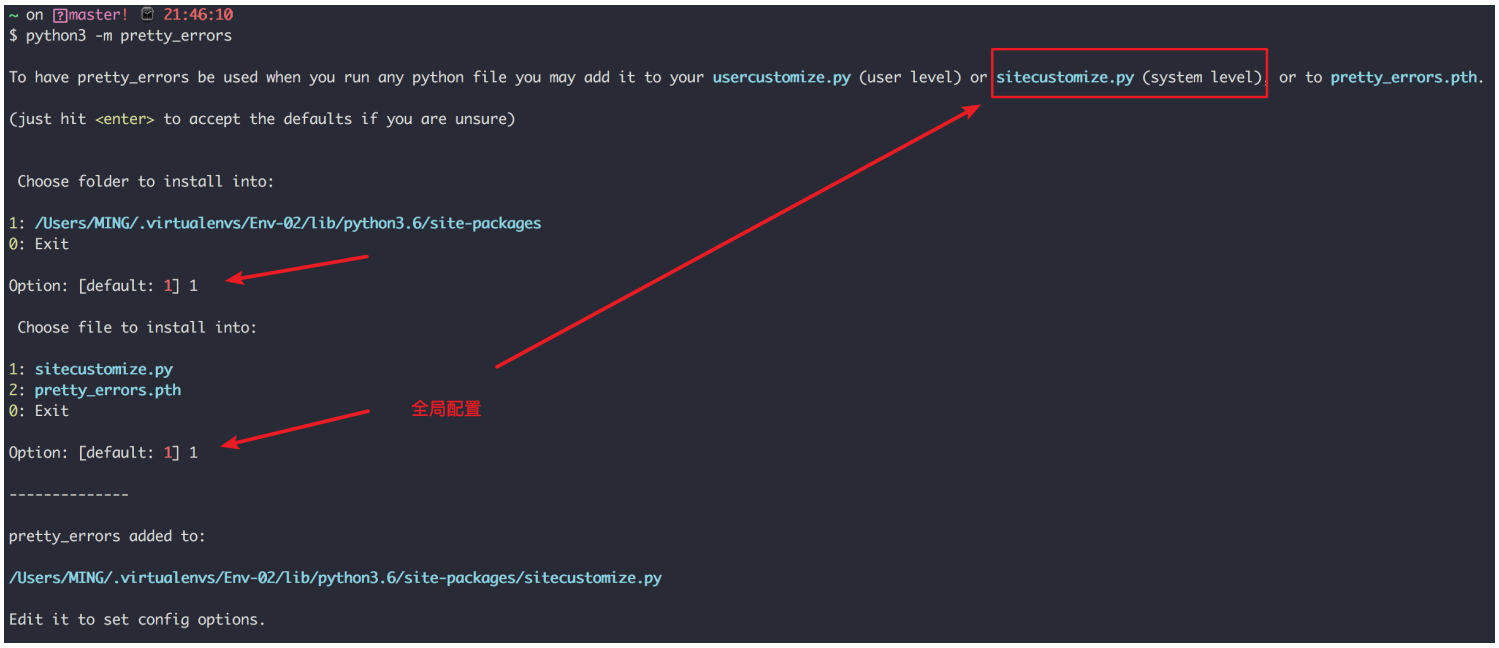
既然既然这样，那 pretty_errors 应该也能支持我们如何自定义我们选用什么样的颜色，怎么排版吧?

答案是显而易见的。

pretty_errors 和其他库不太一样，在一定程度上（如果你使用全局配置的话），它并不是开箱即用的，你在使用它之前可能需要做一下配置。

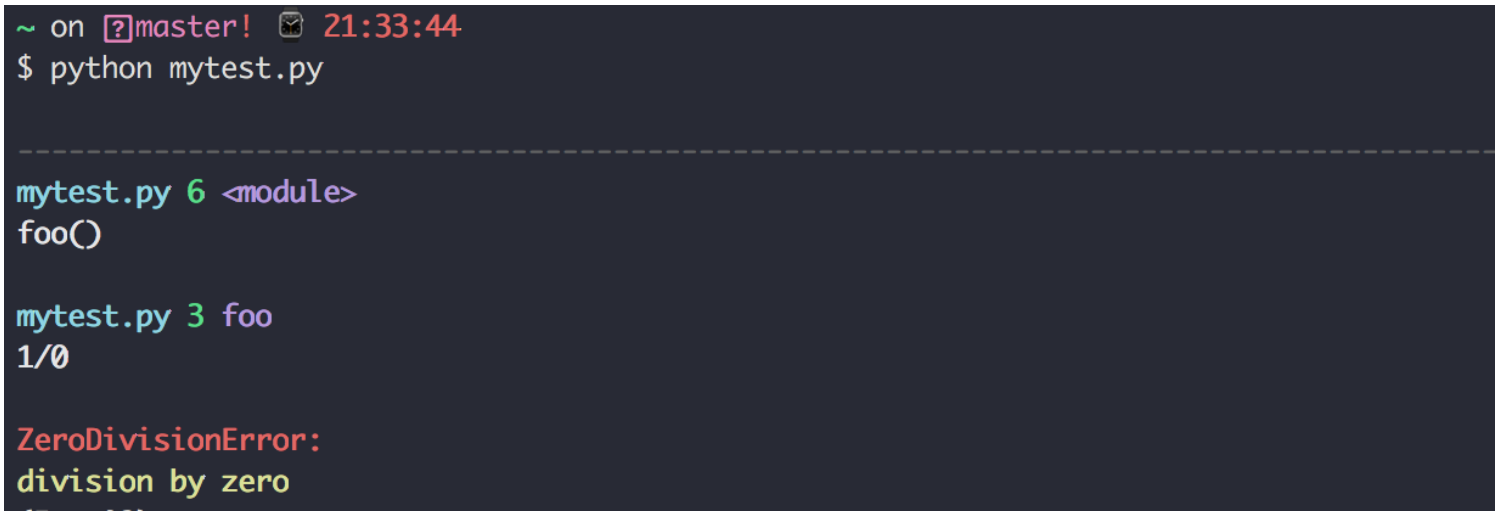
使用这一条命令，会让你进行配置，可以让你在该环境中运行其他脚本时的 traceback 输出都自动美化。

```
$ python3 -m pretty_errors
```



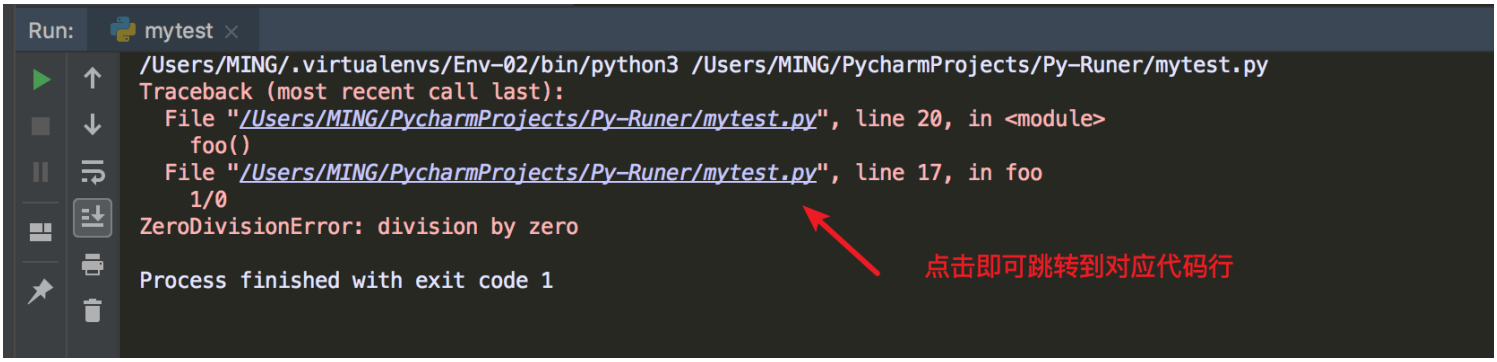
配置完成后，你再运行任何脚本，traceback 都会自动美化了。

不仅是在我的 iTerm 终端下



在 PyCharm 中也会

唯一的缺点就是，原先在 PyCharm 中的 traceback 可以直接点击 `文件路径` 直接跳转到对应错误文件代码行，而你如果是在 VSCode 可以使用 下面自定义配置的方案解决这个问题（下面会讲到，参数是：`display_link`）。



因此，有些情况下，你并不想设置 `pretty_errors` 全局可用。

那怎么取消之前的配置呢?

只需要再次输出 `python -m pretty_errors`，输出入 `C` 即可清除。

4. 单文件中使用

取消全局可用后，你可以根据自己需要，在你需要使用 `pretty-errors` 的脚本文件中导入 `pretty_errors`，即可使用

```
import pretty_errors
```

就像这样

```
import pretty_errors

def foo():
    1/0

if __name__ == "__main__":
    foo()
```

值得一提的是，使用这种方式，若是你的脚本中，出现语法错误，则输出的异常信息还是按照之前的方式展示，并不会被美化。

因此，为了让美化更彻底，官方推荐你使用 `python -m pretty_errors`

5. 自定义设置

上面的例子里，我们使用的都是 `pretty_errors` 的默认美化格式，展示的信息并没有那么全。

比如

- 它并没有展示报错文件的绝对路径，这将使我们很难定位到是哪个文件里的代码出现错误。
- 如果能把具体报错的代码，给我们展示在终端屏幕上，就不需要我们再到源码文件中排查原因了。

如果使用了 `pretty_errors` 导致异常信息有丢失，那还不如不使用 `pretty_errors` 呢。

不过，可以告诉你的是， `pretty_errors` 并没有你想象的那么简单。

它足够开放，支持自定义配置，可以由你选择你需要展示哪些信息，怎么展示?

这里举一个例子

```
import pretty_errors

# 【重点】进行配置
```

```
pretty_errors.configure(
    separator_character = '*',
    filename_display    = pretty_errors.FILENAME_EXTENDED,
    line_number_first   = True,
    display_link        = True,
    lines_before        = 5,
    lines_after         = 2,
    line_color          = pretty_errors.RED + '> ' + pretty_errors.default_conf
ig.line_color,
    code_color          = '  ' + pretty_errors.default_config.line_color,
)

# 原来的代码
def foo():
    1/0

if __name__ == "__main__":
    foo()
```

在你像上面这样使用 `pretty_errrs.configure` 进行配置时，抛出的的异常信息就变成这样了。

```
/Users/MING/.virtualenvs/Env-02/bin/python3 /Users/MING/PycharmProjects/Py-Runer/mytest.py

********************************************************************************
34 <module> /Users/MING/PycharmProjects/Py-Runer/mytest.py
"/Users/MING/PycharmProjects/Py-Runer/mytest.py", line 34
> foo()

31 foo /Users/MING/PycharmProjects/Py-Runer/mytest.py
"/Users/MING/PycharmProjects/Py-Runer/mytest.py", line 31
      truncate_code       = False,
      display_locals      = False
  )                                                    1. 显示报错文件的绝对路径
                                                       2. 显示产生报错的具体代码
  def foo():
>     1/0

  if __name__ == "__main__":

ZeroDivisionError:
division by zero

Process finished with exit code 1
```

当然了，`pretty_errors.configure()` 还可以接收很多的参数，你可以根据你自己的需要进行配置。

5.1 设置颜色

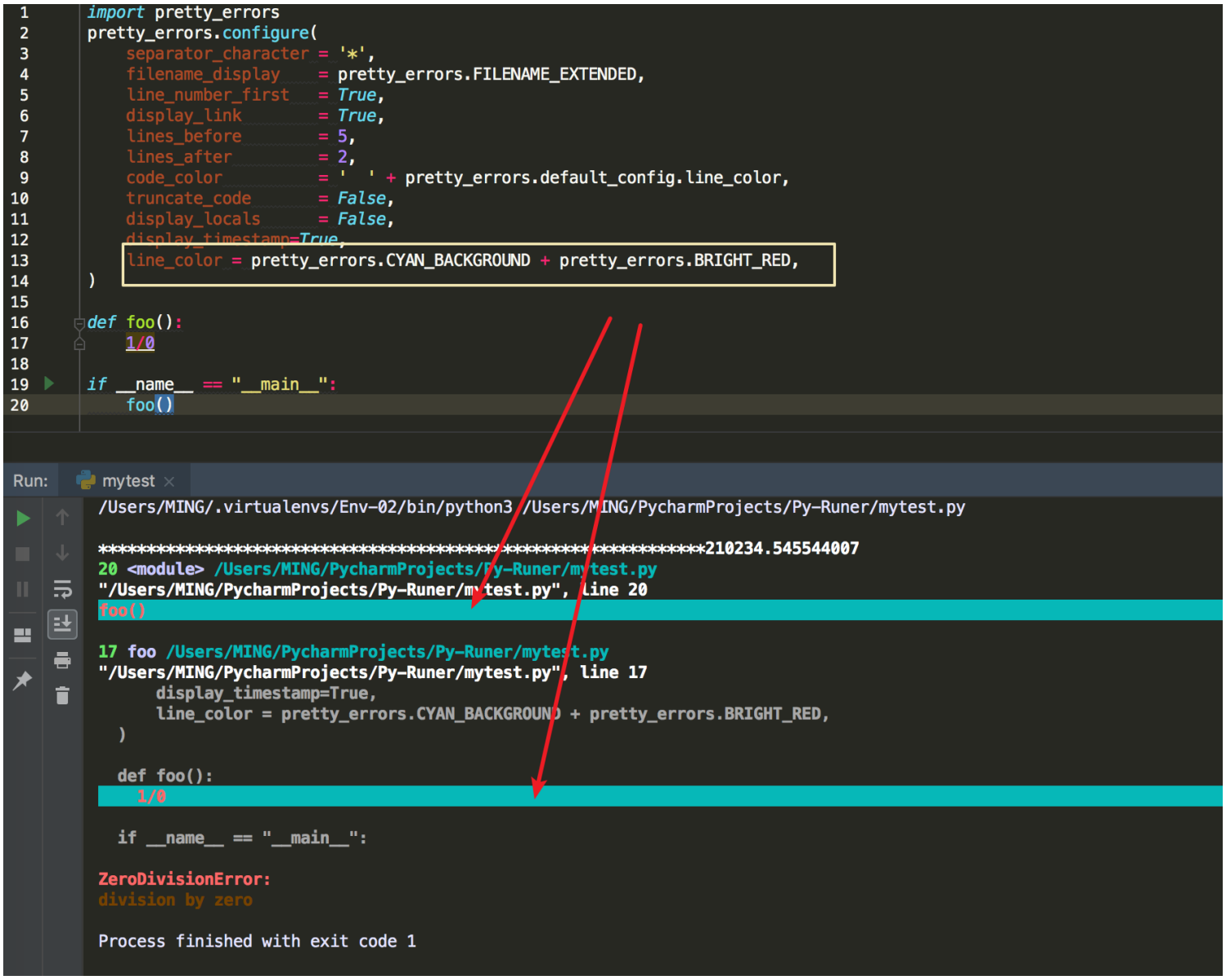
- `header_color`： 设置标题行的颜色。
- `timestamp_color`： 设置时间戳颜色
- `default_color`： 设置默认的颜色
- `filename_color`： 设置文件名颜色
- `line_number_color`： 设置行号颜色。
- `function_color`： 设置函数颜色。
- `link_color`： 设置链接的颜色。

在设置颜色的时候， `pretty_errors` 提供了一些常用的 颜色常量供你直接调取。

- `BLACK`： 黑色
- `GREY`： 灰色
- `RED`： 红色
- `GREEN`： 绿色
- `YELLOW`： 黄色
- `BLUE`： 蓝色
- `MAGENTA`： 品红色
- `CYAN`： 蓝绿色
- `WHITE`： 白色

而每一种颜色，都相应的匹配的 `BRIGHT_` 变体 和 `_BACKGROUND` 变体，

其中， `_BACKGROUND` 用于设置背景色，举个例子如下。



5.2 设置显示内容

- `line_number_first` 启用后，将首先显示行号，而不是文件名。
- `lines_before` ： 显示发生异常处的前几行代码
- `lines_after` ： 显示发生异常处的后几行代码
- `display_link` ：启用后，将在错误位置下方写入链接，VScode将允许您单击该链接。
- `separator_character` ：用于创建标题行的字符。默认情况下使用连字符。如果设置为 `''` 或者 `None` ，标题将被禁用。
- `display_timestamp` ：启用时，时间戳将写入回溯头中。

- `display_locals`
  启用后，将显示在顶部堆栈框架代码中的局部变量及其值。

- `display_trace_locals`
  启用后，其他堆栈框架代码中出现的局部变量将与它们的值一起显示。

5.3 设置怎么显示

- `line_length`：设置每行的长度，默认为0，表示每行的输出将与控制台尺寸相匹配，如果你设置的长度将好与控制台宽度匹配，则可能需要禁用 `full_line_newline`，以防止出现明显的双换行符。
- `full_line_newline`：当输出的字符满行时，是否要插入换行符。
- `timestamp_function`
  调用该函数以生成时间戳。默认值为 `time.perf_counter`。
- `top_first`
  启用后，堆栈跟踪将反转，首先显示堆栈顶部。
- `display_arrow`
  启用后，将针对语法错误显示一个箭头，指向有问题的令牌。
- `truncate_code`
  启用后，每行代码将被截断以适合行长。
- `stack_depth`
  要显示的堆栈跟踪的最大条目数。什么时候 `0` 将显示整个堆栈，这是默认值。
- `exception_above`
  启用后，异常将显示在堆栈跟踪上方。
- `exception_below`：
  启用后，异常显示在堆栈跟踪下方。
- `reset_stdout`
  启用后，重置转义序列将写入stdout和stderr；如果您的控制台留下错误的颜色，请启用此选项。
- `filename_display`

  设置文件名的展示方式，有三个选项：`pretty_errors.FILENAME_COMPACT`、`pretty_errors.FILENAME_EXTENDED`，或者 `pretty_errors.FILENAME_FULL`

以上，就是我对 `pretty_errors` 的使用体验，总的来说，这个库功能非常强大，使用效果也特别酷炫，它就跟 PEP8 规范一样，没有它是可以，但是有了它会更好一样。对于某些想自定义错误输出场景的人，`pretty_errors` 会是一个不错的解决方案，明哥把它推荐给你。

7.3 少有人知的 Python "重试机制"

为了避免由于一些网络或等其他不可控因素，而引起的功能性问题。比如在发送请求时，会因为网络不稳定，往往会有请求超时的问题。

这种情况下，我们通常会在代码中加入重试的代码。重试的代码本身不难实现，但如何写得优雅、易用，是我们要考虑的问题。

这里要给大家介绍的是一个第三方库 – `Tenacity` ，它实现了几乎我们可以使用到的所有重试场景，比如：

1. 在什么情况下才进行重试?
2. 重试几次呢?
3. 重试多久后结束?
4. 每次重试的间隔多长呢?
5. 重试失败后的回调?

在使用它之前 ，先要安装它

```
$ pip install tenacity
```

最基本的重试

无条件重试，重试之间无间隔

```
from tenacity import retry

@retry
def test_retry():
    print("等待重试，重试无间隔执行...")
    raise Exception

test_retry()
```

无条件重试，但是在重试之前要等待 2 秒

```
from tenacity import retry, wait_fixed

@retry(wait=wait_fixed(2))
def test_retry():
    print("等待重试...")
    raise Exception

test_retry()
```

设置停止基本条件

只重试7 次

```
from tenacity import retry, stop_after_attempt

@retry(stop=stop_after_attempt(7))
def test_retry():
    print("等待重试...")
    raise Exception

test_retry()
```

重试 10 秒后不再重试

```
from tenacity import retry, stop_after_delay

@retry(stop=stop_after_delay(10))
def test_retry():
    print("等待重试...")
    raise Exception

test_retry()
```

或者上面两个条件满足一个就结束重试

```
from tenacity import retry, stop_after_delay, stop_after_attempt

@retry(stop=(stop_after_delay(10) | stop_after_attempt(7)))
def test_retry():
    print("等待重试...")
    raise Exception

test_retry()
```

设置何时进行重试

在出现特定错误/异常（比如请求超时）的情况下，再进行重试

```
from requests import exceptions
```

```
from tenacity import retry, retry_if_exception_type

@retry(retry=retry_if_exception_type(exceptions.Timeout))
def test_retry():
    print("等待重试...")
    raise exceptions.Timeout

test_retry()
```

在满足自定义条件时，再进行重试。

如下示例，当 `test_retry` 函数返回值为 False 时，再进行重试

```
from tenacity import retry, stop_after_attempt, retry_if_result

def is_false(value):
    return value is False

@retry(stop=stop_after_attempt(3),
       retry=retry_if_result(is_false))
def test_retry():
    return False

test_retry()
```

重试后错误重新抛出

当出现异常后，tenacity 会进行重试，若重试后还是失败，默认情况下，往上抛出的异常会变成 RetryError，而不是最根本的原因。

因此可以加一个参数（`reraise=True`），使得当重试失败后，往外抛出的异常还是原来的那个。

```
from tenacity import retry, stop_after_attempt

@retry(stop=stop_after_attempt(7), reraise=True)
def test_retry():
    print("等待重试...")
    raise Exception

test_retry()
```

设置回调函数

当最后一次重试失败后，可以执行一个回调函数

```
from tenacity import *

def return_last_value(retry_state):
    print("执行回调函数")
    return retry_state.outcome.result()  # 表示返回原函数的返回值

def is_false(value):
    return value is False

@retry(stop=stop_after_attempt(3),
       retry_error_callback=return_last_value,
       retry=retry_if_result(is_false))
def test_retry():
    print("等待重试中...")
    return False

print(test_retry())
```

输出如下

```
等待重试中...
等待重试中...
等待重试中...
执行回调函数
False
```

学习 Python 扫这个

Python编程时光

学习 Golang 扫这个

Go 编程时光