

公告

最新自学视频 路飞Python免费小课
人生迷茫问题可以加Alex个人微信听鸡



汤



面向对象开发原来如此简单

16人在学 进阶 12小时



Python开发中最常用的11个模块精讲

10人在学 进阶 6小时



跟随Alex金角大王3周上手Python开发

124人在学 入门 19小时

昵称: 金角大王
园龄: 5年5个月
粉丝: 10877
关注: 5
+加关注

随笔 - 29 文章 - 64 评论 - 928

Python之路,Day4 - Python基础4 (new版)

本节内容

- 1. 迭代器&生成器
- 2. 装饰器
- 3. Json & pickle 数据序列化
- 4. 软件目录结构规范
- 5. 作业:ATM项目开发

1.列表生成式, 迭代器&生成器

列表生成式

孩子, 我现在有个需求, 看列表[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],我要求你把列表里的每个值加1, 你怎么实现? 你可能会想到2种方式

普通青年版

```
a = [1,3,4,6,7,7,8,9,11]

for index,i in enumerate(a):
    a[index] +=1
print(a)
```

原值修改

文艺青年版

其实还有一种写法, 如下

装逼青年版

这就叫做列表生成

生成器

通过列表生成式, 我们可以直接创建一个列表。但是, 受到内存限制, 列表容量肯定是有 限的。而且, 创建一个包含100万个元素的列表, 不仅占用很大的存储空间, 如果我们仅 仅需要访问前面几个元素, 那后面绝大多数元素占用的空间都白白浪费了。

所以, 如果列表元素可以按照某种算法推算出来, 那我们是否可以在循环的过程中不断推 算出后续的元素呢? 这样就不必创建完整的list, 从而节省大量的空间。在Python中, 这 种一边循环一边计算的机制, 称为生成器: generator。

要创建一个generator, 有很多种方法。第一种方法很简单, 只要把一个列表生成式的[] 改成(), 就创建了一个generator:

```
1 >>> L = [x * x for x in range(10)]
2 >>> L
3 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
4 >>> g = (x * x for x in range(10))
5 >>> g
6 <generator object <genexpr> at 0x1022ef630>
```

我的标签

职业发展(3)

创业(2)

随笔分类

□职业&生活随笔(22)

文章分类

Python全栈开发之路(12)

Python学习目录(4)

Python自动化开发之路(33)

爬虫(6)

最新评论

1. Re:Django 14天从小白到进阶- Day3

搞定Views组件

不更新了吗

--30岁老古董

2. Re:编程要自学或报班这事你都想不明白,

那必然是你智商不够

大王性格直爽,有目标,肯付出行动!偶像!我要是早几年遇见大王就好了,现在都已经三十了,浪费了大好的青春!

--Xiyue666

3. Re:python 之路, Day11 - python

mysql and ORM

ALTER mytable ADD INDEX

[indexName] ON (username(length))

这句应改为: alter table mytable add index index...

--原竹

4. Re:Python之路,Day3 - Python基础3

@我的恋人叫臭臭 淫角大王听说很厉害吧? ...

--Xiyue666

5. Re:Python之路,Day1 - Python基础1

哎 我为什么早点遇到老男孩 遇到alex 老师呢!

--Xiyue666

阅读排行榜

1. python 之路, 200行Python代码写了个打飞机游戏! (52395)

2. Django + Uwsgi + Nginx 实现生产环境部署(31757)

3. Python Select 解析(27154)

4. 为什么很多IT公司不喜欢进过培训机构的人呢? (20518)

5. 编程要自学或报班这事你都想不明白,那必然是你智商不够(16910)

推荐排行榜

1. 给一位做技术迷茫的同学回信(63)

2. 你做了哪些事,导致老板下调了对你的评价? (51)

创建L和g的区别仅在于最外层的[]和(), L是一个list, 而g是一个generator。

我们可以直接打印出list的每一个元素, 但我们怎么打印出generator的每一个元素呢?

如果要一个一个打印出来, 可以通过next() 函数获得generator的下一个返回值:

```

1  >>> next(g)
2  0
3  >>> next(g)
4  1
5  >>> next(g)
6  4
7  >>> next(g)
8  9
9  >>> next(g)
10 16
11 >>> next(g)
12 25
13 >>> next(g)
14 36
15 >>> next(g)
16 49
17 >>> next(g)
18 64
19 >>> next(g)
20 81
21 >>> next(g)
22 Traceback (most recent call last):
23   File "<stdin>", line 1, in <module>
24 StopIteration

```

我们讲过, generator保存的是算法, 每次调用next(g), 就计算出g的下一个元素的值, 直到计算到最后一个元素, 没有更多的元素时, 抛出StopIteration的错误。

当然, 上面这种不断调用next(g)实在是太变态了, 正确的方法是使用for循环, 因为generator也是可迭代对象:

```

1  >>> g = (x * x for x in range(10))
2  >>> for n in g:
3  ...     print(n)
4  ...
5  0
6  1
7  4
8  9
9  16
10 25
11 36
12 49
13 64
14 81

```

所以, 我们创建了一个generator后, 基本上永远不会调用next(), 而是通过for循环来迭代它, 并且不需要关心StopIteration的错误。

generator非常强大。如果推算的算法比较复杂, 用类似列表生成式的for循环无法实现的时候, 还可以用函数来实现。

比如, 著名的斐波拉契数列 (Fibonacci), 除第一个和第二个数外, 任意一个数都可由前两个数相加得到:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来, 但是, 用函数把它打印出来却很容易:

```

1  def fib(max):

```

3. 关于认识、格局、多维度发展的感触(46)
4. 为什么很多IT公司不喜欢进过培训机构的人呢? (37)
5. 编程要自学或报班这事你都想不明白,那必然是你智商不够(35)

```

2     n, a, b = 0, 0, 1
3     while n < max:
4         print(b)
5         a, b = b, a + b
6         n = n + 1
7     return 'done'

```

注意, 赋值语句:

```

1     a, b = b, a + b

```

相当于:

```

1     t = (b, a + b) # t是一个tuple
2     a = t[0]
3     b = t[1]

```

但不必显式写出临时变量t就可以赋值。

上面的函数可以输出斐波那契数列的前N个数:


```

1     >>> fib(10)
2     1
3     1
4     2
5     3
6     5
7     8
8     13
9     21
10    34
11    55
12    done

```

仔细观察, 可以看出, fib函数实际上是定义了斐波拉契数列的推算规则, 可以从第一个元素开始, 推算出后续任意的元素, 这种逻辑其实非常类似generator。

也就是说, 上面的函数和generator仅一步之遥。要把fib函数变成generator, 只需要把print(b)改为yield b就可以了:



```


def fib(max):
    n, a, b = 0, 0, 1

    while n < max:
        #print(b)
        yield b
        a, b = b, a+b

        n += 1

    return 'done'

```



这就是定义generator的另一种方法。如果一个函数定义中包含yield关键字, 那么这个函数就不再是一个普通函数, 而是一个generator:

```

>>> f = fib(6)
>>> f
<generator object fib at 0x104feaaa0>

```

这里, 最难理解的就是generator和函数的执行流程不一样。函数是顺序执行, 遇到return语句或者最后一行函数语句就返回。而变成generator的函数, 在每次调用next()的时候执行, 遇到yield语句返回, 再次执行时从上次返回的yield语句处继续执行。



```
data = fib(10)
print(data)

print(data.__next__())
print(data.__next__())
print("干点别的事")
print(data.__next__())
print(data.__next__())
print(data.__next__())
print(data.__next__())
print(data.__next__())

#输出
<generator object fib at 0x101be02b0>
1
1
干点别的事
2
3
5
8
13
```

在上面fib的例子，我们在循环过程中不断调用yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代：

```
>>> for n in fib(6):
...     print(n)
...
1
1
2
3
5
8
```

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

```
1 >>> g = fib(6)
2 >>> while True:
3 ...     try:
4 ...         x = next(g)
5 ...         print('g:', x)
6 ...     except StopIteration as e:
7 ...         print('Generator return value:', e.value)
8 ...         break
9 ...
10 g: 1
11 g: 1
12 g: 2
13 g: 3
14 g: 5
15 g: 8
16 Generator return value: done
```

关于如何捕获错误，后面的错误处理还会详细讲解。

还可通过yield实现在单线程的情况下实现并发运算的效果

通过生成器实现协程并行运算

迭代器

我们已经知道，可以直接作用于for循环的数据类型有以下几种：

一类是集合数据类型，如list、tuple、dict、set、str等；

一类是generator，包括生成器和带yield的generator function。

这些可以直接作用于for循环的对象统称为可迭代对象：Iterable。

可以使用isinstance()判断一个对象是否是Iterable对象：

```
1 >>> from collections import Iterable
2 >>> isinstance([], Iterable)
3 True
4 >>> isinstance({}, Iterable)
5 True
6 >>> isinstance('abc', Iterable)
7 True
8 >>> isinstance((x for x in range(10)), Iterable)
9 True
10 >>> isinstance(100, Iterable)
11 False
```

而生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。

***可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。**

可以使用isinstance()判断一个对象是否是Iterator对象：

```
1 >>> from collections import Iterator
2 >>> isinstance((x for x in range(10)), Iterator)
3 True
4 >>> isinstance([], Iterator)
5 False
6 >>> isinstance({}, Iterator)
7 False
8 >>> isinstance('abc', Iterator)
9 False
```

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

```
1 >>> isinstance(iter([]), Iterator)
2 True
3 >>> isinstance(iter('abc'), Iterator)
4 True
```

你可能会问，为什么list、dict、str等数据类型不是Iterator？

这是因为Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

小结

凡是可作用于for循环的对象都是Iterable类型;

凡是可作用于next()函数的对象都是Iterator类型, 它们表示一个惰性计算的序列;

集合数据类型如list、dict、str等是Iterable但不是Iterator, 不过可以通过iter()函数获得一个Iterator对象。

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

```
1 | for x in [1, 2, 3, 4, 5]:
2 |     pass
```

实际上完全等价于:

```
# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])
# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```

2. 装饰器

你是一家视频网站的后端开发工程师, 你们网站有以下几个版块

```
1 | def home():
2 |     print("---首页---")
3 |
4 | def america():
5 |     print("----欧美专区----")
6 |
7 | def japan():
8 |     print("----日韩专区----")
9 |
10 | def henan():
11 |     print("----河南专区----")
```

视频刚上线初期, 为了吸引用户, 你们采取了免费政策, 所有视频免费观看, 迅速吸引了一大批用户, 免费一段时间后, 每天巨大的带宽费用公司承受不了了, 所以准备对比较受欢迎的几个版块收费, 其中包括“欧美”和“河南”专区, 你拿到这个需求后, 想了想, 想收费得先让其进行用户认证, 认证通过后, 再判定这个用户是否是VIP付费会员就可以了, 是VIP就让看, 不是VIP就不让看就行了呗。你觉得这个需求很简单, 因为要对多个版块进行认证, 那应该把认证功能提取出来单独写个模块, 然后每个版块里调用 就可以了, 与你轻轻的就实现了下面的功能。

```
1 | # -*- coding: utf-8 -*-
2 |
3 |
4 | user_status = False #用户登录了就把这个改成True
5 |
6 | def login():
7 |     _username = "alex" #假装这是DB里存的用户信息
8 |     _password = "abc123" #假装这是DB里存的用户信息
9 |     global user_status
10 |
```

```

11     if user_status == False:
12         username = input("user:")
13         password = input("password:")
14
15         if username == _username and password == _password:
16             print("welcome login...")
17             user_status = True
18         else:
19             print("wrong username or password!")
20     else:
21         print("用户已登录, 验证通过...")
22
23 def home():
24     print("---首页---")
25
26 def america():
27     login() #执行前加上验证
28     print("----欧美专区----")
29
30 def japan():
31     print("----日韩专区----")
32
33 def henan():
34     login() #执行前加上验证
35     print("----河南专区----")
36
37
38
39 home()
40 america()
41 henan()

```

此时你信心满满的把这个代码提交给你的TEAM LEADER审核, 没成想, 没过5分钟, 代码就被打回来了, TEAM LEADER给你反馈是, 我现在有很多模块需要加认证模块, 你的代码虽然实现了功能, 但是需要更改需要加认证的各个模块的代码, 这直接违反了软件开发中的一个原则“开放-封闭”原则, 简单来说, 它规定已经实现的功能代码不允许被修改, 但可以被扩展, 即:

- 封闭: 已实现的功能代码块不应该被修改
- 开放: 对现有功能的扩展开放

这个原则你还是第一次听说, 我擦, 再次感受了自己这个野生程序员与正规军的差距, BUT ANYWAY, 老大要求的这个怎么实现呢? 如何在不改原有功能代码的情况下加上认证功能呢? 你一时想不出思路, 只好带着这个问题回家继续憋, 媳妇不在家, 去隔壁老王家串门了, 你正好落的清静, 一不小心就想到了解决方案, 不改源代码可以呀,

你师从沙河金角大王时, 记得他教过你, 高阶函数, 就是把一个函数当做一个参数传给另外一个函数, 当时大王说, 有一天, 你会用到它的, 没想到这时这个知识点突然从脑子里蹦出来了, 我只需要写个认证方法, 每次调用 需要验证的功能 时, 直接 把这个功能 的函数名当做一个参数 传给 我的验证模块不就行了么, 哈哈, 机智如我, 如是你啪啪啪改写了之前的代码

```

1  #_*_coding:utf-8_*_
2
3
4  user_status = False #用户登录了就把这个改成True
5
6  def login(func): #把要执行的模块从这里传进来
7      _username = "alex" #假装这是DB里存的用户信息
8      _password = "abc!23" #假装这是DB里存的用户信息
9      global user_status
10
11     if user_status == False:
12         username = input("user:")
13         password = input("password:")

```

```

14
15         if username == _username and password == _password:
16             print("welcome login...")
17             user_status = True
18         else:
19             print("wrong username or password!")
20
21     if user_status == True:
22         func() # 看这里看这里, 只要验证通过了, 就调用相应功能
23
24 def home():
25     print("---首页---")
26
27 def america():
28     #login() #执行前加上验证
29     print("----欧美专区----")
30
31 def japan():
32     print("----日韩专区----")
33
34 def henan():
35     #login() #执行前加上验证
36     print("----河南专区----")
37
38
39
40 home()
41 login(america) #需要验证就调用 login, 把需要验证的功能 当做一个参数传给login
42 # home()
43 # america()
44 login(henan)

```

你很开心, 终于实现了老板的要求, 不改变原功能代码的前提下, 给功能加上了验证, 此时, 媳妇回来了, 后面还跟着老王, 你两家关系 非常好, 老王经常来串门, 老王也是码农, 你跟他分享了你写的代码, 兴奋的等他看完 夸奖你NB, 没成想, 老王看后, 并没有夸你, 抱起你的儿子, 笑笑说, 你这个代码还是改改吧, 要不然会被开除的, WHAT? 会开除, 明明实现了功能 呀, 老王讲, 没错, 你功能 是实现了, 但是你又犯了一个大忌, 什么大忌?

你改变了调用方式呀, 想一想, 现在没每个需要认证的模块, 都必须调用你的login()方法, 并把自己的函数名传给你, 人家之前可不是这么调用 的, 试想, 如果 有100个模块需要认证, 那这100个模块都得更改调用方式, 这么多模块肯定不止是一个人写的, 让每个人再去修改调用方式 才能加上认证, 你会被骂死的。。。

你觉得老王说的对, 但问题是, 如何即不改变原功能代码, 又不改变原有调用方式, 还能加上认证呢? 你苦思了一会, 还是想不出, 老王在逗你的儿子玩, 你说, 老王呀, 快给我点思路, 实在想不出来, 老王背对着你问,

老王: 学过匿名函数没有?

你: 学过学过, 就是lambda嘛

老王: 那lambda与正常函数的区别是什么?

你: 最直接的区别是, 正常函数定义时需要写名字, 但lambda不需要

老王: 没错, 那lambda定好后, 为了多次调用, 可否也给它命名?

你: 可以呀, 可以写成plus = lambda x:x+1类似这样, 以后再调用plus就可以了, 但这样不就失去了lambda的意义了, 明明人家叫匿名函数呀, 你起了名字有什么用呢?

老王: 我不是要跟你讨论它的意义, 我想通过这个让你明白一个事实

说着, 老王拿起你儿子的画板, 在上面写了以下代码:

```

1  def plus(n):
2      return n+1
3

```



```
4 | plus2 = lambda x:x+1
```

老王：上面这两种写法是不是代表 同样的意思？

你：是的

老王：我给lambda x:x+1 起了个名字叫plus2，是不是相当于def plus2(x)？

你：我擦，你别说，还真是，但老王呀，你想说明什么呢？

老王：没啥，只想告诉你，给函数赋值变量名就像def func_name 是一样的效果，如下面的plus(n)函数，你调用时可以用plus名，还可以再起个其它名字，如

```
1 | calc = plus
2 |
3 | calc(n)
```

你明白我想传达什么意思了么？

你：。。。。。。这。。。。。。嗯。。。。。。不太。。。。明白。。

老王：。。。。这。。。。呵呵。。。。好吧。。。。，那我在给你点一下，你之前写的下面这段调用 认证的代码

```
1 | home()
2 | login(america) #需要验证就调用 login, 把需要验证的功能 当做一个参数传给login
3 | # home()
4 | # america()
5 | login(henan)
```

你之所改变了调用方式，是因为用户每次调用时需要执行login(henan)，类似的。其实稍一改就可以了呀

```
1 | home()
2 | america = login(america)
3 | henan = login(henan)
```

这样你，其它人调用henan时，其实相当于调用了login(henan)，通过login里的验证后，就会自动调用henan功能。

你：我擦，还真是唉。。。老王，还是你nb。。。不过，等等，我这样写了好，那用户调用时，应该是下面这个样子

```
1 | home()
2 | america = login(america) #你在这里相当于把america这个函数替换了
3 | henan = login(henan)
4 |
5 | #那用户调用时依然写
6 | america()
```

但问题在于，还不等用户调用，你的america = login(america)就会先自己把america执行了呀。。。你，你应该等我用户调用 的时候 再执行才对呀，不信我试给你看。。。

老王：哈哈，你说的没错，这样搞会出现这个问题？但你想有没有解决办法 呢？

你：我擦，你指思路呀，大哥。。我哪知道 下一步怎么走。。。

老王：算了，估计你也想不出来。。。学过嵌套函数没有？

你：yes,然后呢？

老王：想实现一开始你写的america = login(america)不触发你函数的执行，只需要在这个login里面再定义一层函数，第一次调用america = login(america)只调用到外层login，这个login虽然会执行，但不会触发认证了，因为认证的所有代码被封装在login里层的新定义的函数里了，login只返回 里层函数的函数名，这样下次再执行america()时，就会调用里层函数啦。。。

你：。。。。。。什么？什么个意思，我蒙逼了。。。

老王：还是给你看代码吧。。

```

1  def login(func): #把要执行的模块从这里传进来
2
3      def inner():#再定义一层函数
4          _username = "alex" #假装这是DB里存的用户信息
5          _password = "abc!23" #假装这是DB里存的用户信息
6          global user_status
7
8          if user_status == False:
9              username = input("user:")
10             password = input("password:")
11
12             if username == _username and password == _password:
13                 print("welcome login...")
14                 user_status = True
15             else:
16                 print("wrong username or password!")
17
18             if user_status == True:
19                 func() # 看这里看这里，只要验证通过了，就调用相应功能
20
21         return inner #用户调用login时，只会返回inner的内存地址，下次再调用时加上()才

```

此时你仔细看了老王写的代码，感觉老王真不是一般人呀，连这种奇淫巧技都能想出来。。。心中默默感谢上天赐你一个大牛邻居。

你：老王呀，你这个姿势很nb呀，你独创的？

此时你媳妇噗嗤的笑出声来，你也不知道她笑个球。。。

老王：呵呵，这不是我独创的呀当然，这是开发中一个常用的玩法，叫语法糖，官方名称“装饰器”，其实上面的写法，还可以更简单

可以把下面代码去掉

```

1  america = login(america) #你在这里相当于把america这个函数替换了

```

只在你要装饰的函数上面加上下面代码

```

1  @login
2  def america():
3      #login() #执行前加上验证
4      print("----欧美专区----")
5
6  def japan():
7      print("----日韩专区----")
8
9  @login
10 def henan():
11     #login() #执行前加上验证
12     print("----河南专区----")

```

效果是一样的。

你开心的玩着老王教你的新姿势，玩着玩着就手贱给你的“河南专区”版块加了个参数，然后，结果出错了。。。



```
python基础 自动化基础day4 decorator2.py
40 @login
41 def henan(style):
42     ...
43     :param style: 喜欢看什么类型的, 就传进来
44     :return:
45     ...
46 #login() #执行前加上验证 现场还原
47 print("-----河南专区-----")
48
49 home()
50 # america = login(america) #你在这里相当于把america这个函数替换了
51 # henan = login(henan)
52
53 # 那用户调用时依然写
54 # america()
55
56 henan("3p")
57
```

Run: decorator2.py

Traceback (most recent call last):

File "/Users/alex/Documents/work/PyProjects/python基础/自动化基础day4/decorator2.py", line 57, in <module>

henan("3p")

TypeError: inner() takes 0 positional arguments but 1 was given

Process finished with exit code 1

你: 老王, 老王, 怎么传个参数就不行了呢?

老王: 那必然呀, 你调用henan时, 其实是相当于调用的login, 你的henan第一次调用时 henan = login(henan), login就返回了inner的内存地址, 第2次用户自己调用 henan("3p"),实际上相当于调用的inner,但你的inner定义时并没有设置参数, 但你给他传了个参数, 所以自然就报错了呀

你: 但是我的 版块需要传参数呀, 你不让我传不行呀。。。

老王: 没说不能让你传, 稍微改动便可。。



```
user_status = False #用户登录了就把这个改成True
def login(func): #把要执行的模块从这里传进来 加个arg1参数
    def inner(arg1): #再定义一层函数
        _username = "alex" #假装这是DB里存的用户信息
        _password = "abc!23" #假装这是DB里存的用户信息
        global user_status

        if user_status == False:
            username = input("user:")
            password = input("password:")

            if username == _username and password == _password:
                print("welcome login...")
                user_status = True
            else:
                print("wrong username or password!")

        if user_status == True:
            func(arg1) #看这里看这里, 只要验证通过了, 就调用相应功能

    return inner #用户调用login时, 只会返回inner的内存地址, 下次再调用时加上()才会执行inner函数

def home():...
```

老王: 你再试试就好了。

你: 果然好使, 大神就是大神呀。。。不过, 如果有多个参数呢?

老王:。。。。老弟, 你不要什么都让我教你吧, 非固定参数你没学过么?

*args,**kwargs...

你: 噢。。。还能这么搞?,nb,我再试试。

你身陷这种新玩法中无法自拔, 竟没注意到老王已经离开, 你媳妇告诉你说了不打扰你加班, 今晚带孩子去跟她姐妹住, 你觉得媳妇真体贴, 最终, 你终于搞定了所有需求, 完全遵循开放-封闭原则, 最终代码如下。

[+ View Code](#)

此时, 你已累的不行了, 洗洗就抓紧睡了, 半夜, 上厕所, 隐隐听到隔壁老王家有微弱的女人的声音传来, 你会心一笑, 老王这家伙, 不声不响找了女朋友也不带给我看看, 改天一定要见下真人。。。

第二天早上，产品经理又提了新的需求，要允许用户选择用qq\weibo\weixin认证，此时的你，已深谙装饰器各种装逼技巧，轻松的就实现了新的需求。

带参数的装饰器

3.Json & pickle 数据序列化

参考 <http://www.cnblogs.com/alex3714/articles/5161349.html>

4.软件目录结构规范

为什么要设计好目录结构？

"设计项目目录结构"，就和"代码编码风格"一样，属于个人风格问题。对于这种风格上的规范，一直都存在两种态度：

1. 一类同学认为，这种个人风格问题"无关紧要"。理由是能让程序work就好，风格问题根本不是问题。
2. 另一类同学认为，规范化能更好的控制程序结构，让程序具有更高的可读性。

我是比较偏向于后者的，因为我是前一类同学思想行为下的直接受害者。我曾经维护过一个非常不好读的项目，其实现的逻辑并不复杂，但是却耗费了我非常长的时间去理解它想表达的意思。从此我个人对于提高项目可读性、可维护性的要求就很高了。"项目目录结构"其实也是属于"可读性和可维护性"的范畴，我们设计一个层次清晰的目录结构，就是为了达到以下两点：

1. 可读性高：不熟悉这个项目的代码的人，一眼就能看懂目录结构，知道程序启动脚本是哪个，测试目录在哪儿，配置文件在哪儿等等。从而非常快速的了解这个项目。
2. 可维护性高：定义好组织规则后，维护者就能很明确地知道，新增的哪个文件和代码应该放在什么目录之下。这个好处是，随着时间的推移，代码/配置的规模增加，项目结构不会混乱，仍然能够组织良好。

所以，我认为，保持一个层次清晰的目录结构是有必要的。更何况组织一个良好的工程目录，其实是一件很简单的事儿。

目录组织方式

关于如何组织一个较好的Python工程目录结构，已经有一些得到了共识的目录结构。在Stackoverflow的[这个问题](#)上，能看到大家对Python目录结构的讨论。

这里面说的已经很好了，我也不打算重新造轮子列举各种不同的方式，这里面我说一下我的理解和体会。

假设你的项目名为foo，我比较建议的最方便快捷目录结构这样就足够了：

```
Foo/  
|-- bin/  
|   |-- foo  
|  
|-- foo/  
|   |-- tests/
```

```
| | |__ __init__.py
| | |__ test_main.py
| |
| |__ __init__.py
| |__ main.py
|
|-- docs/
|   |-- conf.py
|   |-- abc.rst
|
|-- setup.py
|-- requirements.txt
|-- README
```

简要解释一下：

1. bin/: 存放项目的一些可执行文件，当然你可以起名script/之类的也行。
2. foo/: 存放项目的所有源代码。(1) 源代码中的所有模块、包都应该放在此目录。不要置于顶层目录。(2) 其子目录tests/存放单元测试代码；(3) 程序的入口最好命名为main.py。
3. docs/: 存放一些文档。
4. setup.py: 安装、部署、打包的脚本。
5. requirements.txt: 存放软件依赖的外部Python包列表。
6. README: 项目说明文件。

除此之外，有一些方案给出了更加多的内容。比如LICENSE.txt,ChangeLog.txt文件等，我没有列在这里，因为这些东西主要是项目开源的时候需要用到。如果你想写一个开源软件，目录该如何组织，可以参考[这篇文章](#)。

下面，再简单讲一下我对这些目录的理解和个人要求吧。

关于README的内容

这个我觉得是每个项目都应该有的一个文件，目的是能简要描述该项目的信息，让读者快速了解这个项目。

它需要说明以下几个事项：

1. 软件定位，软件的基本功能。
2. 运行代码的方法：安装环境、启动命令等。
3. 简要的使用说明。
4. 代码目录结构说明，更详细点可以说明软件的基本原理。
5. 常见问题说明。

我觉得有以上几点是比较好的一个README。在软件开发初期，由于开发过程中以上内容可能不明确或者发生变化，并不是一定要在一开始就将所有信息都补全。但是在项目完结的时候，是需要撰写这样的一个文档的。

可以参考Redis源码中Readme的写法，这里面简洁但是清晰的描述了Redis功能和源码结构。

关于requirements.txt和setup.py

setup.py

一般来说，用setup.py来管理代码的打包、安装、部署问题。业界标准的写法是用Python流行的打包工具setuptools来管理这些事情。这种方式普遍应用于开源项目中。不过这里的核心思想不是用标准化的工具来解决这些问题，而是说，**一个项目一定要有一个安装部署工具**，能快速便捷的在一台新机器上将环境装好、代码部署好和将程序运行起来。

这个我是踩过坑的。

我刚开始接触Python写项目的时候，安装环境、部署代码、运行程序这个过程全是手动完成，遇到过以下问题：

1. 安装环境时经常忘了最近又添加了一个新的Python包，结果一到线上运行，程序就出错了。
2. Python包的版本依赖问题，有时候我们程序中使用的是一个版本的Python包，但是官方的已经是最新的包了，通过手动安装就可能装错了。
3. 如果依赖的包很多的话，一个一个安装这些依赖是很费时的事情。
4. 新同学开始写项目的时候，将程序跑起来非常麻烦，因为可能经常忘了要怎么安装各种依赖。

setup.py可以将这些事情自动化起来，提高效率、减少出错的概率。"复杂的东西自动化，能自动化的东西一定要自动化。"是一个非常好的习惯。

setuptools的文档比较庞大，刚接触的话，可能不太好找到切入点。学习技术的方式就是看他人是怎么用的，可以参考一下Python的一个Web框架，flask是如何写的：[setup.py](#)当然，简单点自己写个安装脚本（deploy.sh）替代setup.py也未尝不可。

requirements.txt

这个文件存在的目的是：

1. 方便开发者维护软件的包依赖。将开发过程中新增的包添加进这个列表中，避免在setup.py安装依赖时漏掉软件包。
2. 方便读者明确项目使用了哪些Python包。

这个文件的格式是每一行包含一个包依赖的说明，通常是flask>=0.10这种格式，要求是这个格式能被pip识别，这样就可以简单的通过 `pip install -r requirements.txt`来把所有Python包依赖都装好了。具体格式说明：[点这里](#)。

关于配置文件的使用方法

注意，在上面的目录结构中，没有将conf.py放在源码目录下，而是放在docs/目录下。

很多项目对配置文件的使用做法是：

1. 配置文件写在一个或多个python文件中，比如此处的conf.py。
2. 项目中哪个模块用到这个配置文件就直接通过 `import conf` 这种形式来在代码中使用配置。

这种做法我不太赞同：

1. 这让单元测试变得困难（因为模块内部依赖了外部配置）
2. 另一方面配置文件作为用户控制程序的接口，应当可以由用户自由指定该文件的路径。
3. 程序组件可复用性太差，因为这种贯穿所有模块的代码硬编码方式，使得大部分模块都依赖conf.py这个文件。

所以，我认为配置的使用，更好的方式是，

1. 模块的配置都是可以灵活配置的，不受外部配置文件的影响。
2. 程序的配置也是可以灵活控制的。

能够佐证这个思想的是，用过nginx和mysql的同学都知道，nginx、mysql这些程序都可以自由的指定用户配置。

所以，不应当在代码中直接 `import conf` 来使用配置文件。上面目录结构中的conf.py，是给出的一个配置样例，不是在写死在程序中直接引用的配置文件。可以通过给main.py启动参数指定配置路径的方式来让程序读取配置内容。当然，这里的conf.py你可以换个类似的名字，比如settings.py。或者你也可以使用其他格式的内容来编写配置文件，比如settings.yaml之类的。

5. 本节作业

作业需求：

模拟实现一个ATM + 购物商城程序

- 1. 额度 15000或自定义
- 2. 实现购物商城，买东西加入 购物车，调用信用卡接口结账
- 3. 可以提现，手续费5%
- 4. 每月22号出账单，每月10号为还款日，过期未还，按欠款总额 万分之5 每日计息
- 5. 支持多账户登录
- 6. 支持账户间转账
- 7. 记录每月日常消费流水
- 8. 提供还款接口
- 9. ATM记录操作日志
- 10. 提供管理接口，包括添加账户、用户额度，冻结账户等。。。
- 11. 用户认证用装饰器

示例代码 https://github.com/triaquae/py3_training/tree/master/atm

简易流程图：

<https://www.processon.com/view/link/589eb841e4b0999184934329>

分类: [Python自动化开发之路](#)

好文要顶

关注我

收藏该文

金角大王

关注 - 5

粉丝 - 10877

+加关注

190

posted @ 2016-08-12 15:12 金角大王 阅读(70516) 评论(17) 编辑 收藏

评论列表

| | | | |
|--|------------------|-----------------|-------------|
| #1楼 | 2016-08-15 17:27 | freedom_dog | 回复 引用 |
| 多谢。 | | | 支持(0) 反对(0) |
| #2楼 | 2017-02-10 17:42 | super-sos | 回复 引用 |
| 打死你个龟孙儿 | | | 支持(1) 反对(0) |
| #3楼 | 2017-02-19 20:39 | great_zhi | 回复 引用 |
| 打死你个龟孙儿 | | | 支持(0) 反对(0) |
| #4楼 | 2017-06-21 09:55 | Frank_srv_world | 回复 引用 |
| 传参的过程还是有些迷糊 | | | 支持(0) 反对(0) |
| #5楼 | 2017-07-07 11:30 | 小温xy | 回复 引用 |
| 单线程实现并发效果的producer函数不用带参数name吧 | | | 支持(0) 反对(1) |
| #6楼 | 2017-07-18 15:14 | hao_xiaoyu | 回复 引用 |
| alex的小故事写的也是津津有味 | | | 支持(2) 反对(0) |
| #7楼 | 2017-09-14 14:28 | SmartMing | 回复 引用 |
| 楼主没老婆或者经常扮演隔壁老王。让人恶心的段子 | | | 支持(6) 反对(6) |
| #8楼 | 2017-12-03 20:23 | Byron-He | 回复 引用 |
| @ SmartMing 这只是个一简单的举例方式，不知道为什么你这样抨击楼主！ | | | |

支持(0) 反对(2)

#9楼 2018-01-07 00:59 彭世瑜 回复 引用

交个作业，做了3天总算做好了，分享给大家
<https://github.com/mouday/Atm>

支持(3) 反对(0)

#10楼 2018-03-15 21:28 小夕公子 回复 引用

Alex很会编故事。。。。。。

支持(0) 反对(0)

#11楼 2018-03-28 14:47 pennychenpei 回复 引用

```
# *_coding:utf-8_*
__author__ = 'Alex Li'

import time
def consumer(name):
    print("%s 准备吃包子啦!" %name)
    while True:
        baozi = yield

    print("包子[%s]来了,被[%s]吃了!" %(baozi,name))

def producer(name):
    c = consumer('A')
    c2 = consumer('B')
    c.__next__()
    c2.__next__()
    print("老子开始准备做包子啦!")
    for i in range(10):
        time.sleep(1)
        print("做了2个包子!")
        c.send(i)
        c2.send(i)

producer("alex")

这个执行报错

File "C:\Users\cwxx210659\Desktop\python learning\hanshu.py", line 75, in producer
print(c.__next__())
AttributeError: 'generator' object has no attribute '__next__'
```

支持(0) 反对(0)

#12楼 2018-03-28 15:17 pennychenpei 回复 引用

```
c.__next__()
c2.__next__() 改成next (c) 和next(c2)就是ok的。
```

支持(0) 反对(0)

#13楼 2018-04-04 10:46 alex_hrg 回复 引用

花了三天时间，终于把作业交上，发个地址大家共同学习<https://github.com/hrghrg/atm>

支持(0) 反对(0)

#14楼 2018-08-15 17:51 charles_guo 回复 引用

@ pennychenpei
改成c.next()

支持(0) 反对(0)

#15楼 2018-09-12 13:04 yoocin 回复 引用

@ pennyche换成next (c1) 跟next (c2) 就好了

支持(0) 反对(0)

#16楼 2019-05-24 05:18 我也不想这么菜 回复 引用

```
import time
def consumer(name):
    print("%s 准备吃包子啦!" %name)
    while True:
        baozi = yield

    print("包子[%s]来了,被[%s]吃了!" %(baozi,name))

def producer(name):
```



```
c = consumer('A')
c2 = consumer('B')
c.__next__()
c2.__next__()
print("老子开始准备做包子啦!")
for i in range(10):
    time.sleep(1)
    print("做了2个包子!")
    c.send(i)
    c2.send(i)

producer("alex")
```

哪位大佬能指点一下这里的
c.__next__()
c2.__next__()中的__next__是用老干啥的啊

支持(1) 反对(0)

#17楼 2020-04-22 01:12 zoey_chou

回复 引用

```
def fib(max): n, a, b = 0, 0, 1 while n < max: print(b) a, b = b, a + b n = n + 1 return 'done'
```

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。
提问：目前这个斐波那契还是一个函数,也就是按顺序执行的,为何是先打印print(b),然后计算a,b的值?挪了print b的位置值又发生了变化

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

发表评论

编辑 预览

B 🔗 <> “ ” ☒

支持 Markdown

提交评论 退出 订阅评论

[Ctrl+Enter快捷键提交]

- 【推荐】超50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】了不起的开发者，挡不住的华为，园子里的品牌专区
- 【推荐】有道智云周年庆，API服务大放送，注册即送100元体验金！
- 【推荐】史上最全 Vue 面试题汇总



相关博文：

- [【Python之路Day4】基础篇](#)
 - [python之路day4](#)
 - [python之路_day4](#)
 - [python之路, Day4-Python基础](#)
 - [Python之路,Day4 - Python基础4](#)
- » [更多推荐...](#)

最新 IT 新闻：

- [SpaceX 高管天团全披露 | 与偏执狂埃隆·马斯克共事的CFO们](#)
 - [假设太阳系内有一颗黑洞，那怎样才能找到它？](#)
 - [国产手机产业链“千里之堤”，毁于苹果？](#)
 - [蒂姆·库克等大型科技公司CEO参加的反垄断听证会改在周三举行](#)
 - [华为郭平：全球5G部署告一段落 下一重点是释放5G红利](#)
- » [更多新闻...](#)