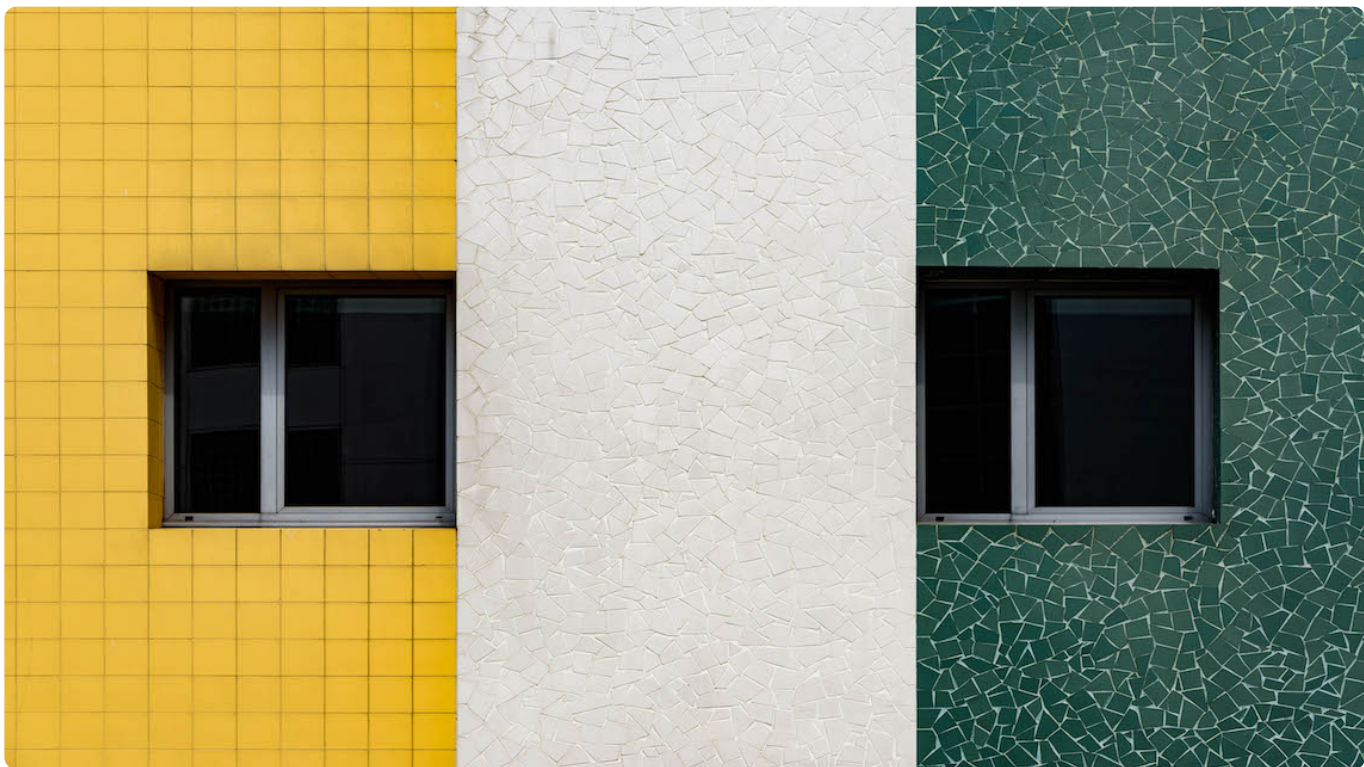


13 | 搭建积木：Python 模块化

2019-06-07 景霄

Python核心技术与实战

[进入课程 >](#)



讲述：冯永吉

时长 11:05 大小 10.16M



你好，我是景霄。


这是基础版块的最后一节。到目前为止，你已经掌握了 Python 这一门当代武功的基本招式和套路，走出了新手村，看到了更远的世界，有了和这个世界过过招的冲动。

于是，你可能开始尝试写一些不那么简单的系统性工程，或者代码量较大的应用程序。这时候，简单的一个 py 文件已经过于臃肿，无法承担一个重量级软件开发的重任。


今天这节课的主要目的，就是化繁为简，将功能模块化、文件化，从而可以像搭积木一样，将不同的功能，组件在大型工程中搭建起来。

简单模块化


说到最简单的模块化方式，你可以把函数、类、常量拆分到不同的文件，把它们放在同一个文件夹，然后使用 `from your_file import function_name, class_name` 的方式调用。之后，这些函数和类就可以在文件内直接使用了。

 复制代码

```
1 # utils.py
2
3 def get_sum(a, b):
4     return a + b
```

 复制代码

```
1 # class_utils.py
2
3 class Encoder(object):
4     def encode(self, s):
5         return s[::-1]
6
7 class Decoder(object):
8     def decode(self, s):
9         return ''.join(reverse(list(s)))
```

 复制代码


```
1 # main.py
2
3 from utils import get_sum
4 from class_utils import *
5
6 print(get_sum(1, 2))
7
8 encoder = Encoder()
9 decoder = Decoder()
10
11 print(encoder.encode('abcde'))
12 print(encoder.encode('edcba'))
13
14 ##### 输出 #####
15
16 3
17 edcba
18 abcde
```

我们来看这种方式的代码：get_sum() 函数定义在 utils.py，Encoder 和 Decoder 类则在 class_utils.py，我们在 main 函数直接调用 from import，就可以将我们需要的东西 import 过来。


非常简单。

但是这就足够了吗？当然不，慢慢地，你会发现，所有文件都堆在一个文件夹下也并不是办法。


于是，我们试着建一些子文件夹：

 复制代码


```
1 # utils/utils.py
2
3 def get_sum(a, b):
4     return a + b
```

 复制代码

```
1 # utils/class_utils.py
2
3 class Encoder(object):
4     def encode(self, s):
5         return s[::-1]
6
7 class Decoder(object):
8     def decode(self, s):
9         return ''.join(reverse(list(s)))
```


 复制代码

```
1 # main.py
2
3 from utils.utils import get_sum
4
5 print(get_sum(1, 2))
6
7 ##### 输出 #####
8
9 3
```

 复制代码

```
1 # src/sub_main.py
2
3 import sys
4 sys.path.append("..")
5
6 from utils.class_utils import *
7
8 encoder = Encoder()
9 decoder = Decoder()
10
11 print(encoder.encode('abcde'))
12 print(encoder.encode('edcba'))
13
14 ##### 输出 #####
15
16 edcba
17 abcde
```

而这一次，我们的文件结构是下面这样的：

 复制代码

```
1 .
2 |— utils
3 |   |— util.py
4 |   |— class_util.py
5 |— src
6 |   |— sub_main.py
7 |— main.py
```

很容易看出，main.py 调用子目录的模块时，只需要使用 `.` 代替 `/` 来表示子目录，`utils.utils` 表示 `utils` 子文件夹下的 `utils.py` 模块就行。

那如果我们想调用上层目录呢？注意，`sys.path.append("..")` 表示将当前程序所在位置**向上**提了一级，之后就能调用 `utils` 的模块了。

同时要注意一点，`import` 同一个模块只会被执行一次，这样就可以防止重复导入模块出现问题。当然，良好的编程习惯应该杜绝代码多次导入的情况。**在 Facebook 的编程规范中，除了一些极其特殊的情况，`import` 必须位于程序的最前端。**

最后我想再提一下版本区别。你可能在许多教程中看到过这样的要求：我们还需要在模块所在的文件夹新建一个 `__init__.py`，内容可以为空，也可以用来表述包对外暴露的模块接口。不过，事实上，这是 Python 2 的规范。在 Python 3 规范中，`__init__.py` 并不是必须的，很多教程里没提过这一点，或者没讲明白，我希望你还是能注意到这个地方。

整体而言，这就是最简单的模块调用方式了。在我初用 Python 时，这种方式已经足够我完成大学期间的项目了，毕竟，很多学校项目的文件数只有个位数，每个文件代码也只有几百行，这种组织方式能帮我顺利完成任务。

但是在我来到 Facebook 后，我发现，一个项目组的 workspace 可能有上千个文件，有几十万到几百万行代码。这种调用方式已经完全不够用了，学会新的组织方式迫在眉睫。

接下来，我们就系统学习下，模块化的科学组织方式。

项目模块化

我们先来回顾下相对路径和绝对路径的概念。

在 Linux 系统中，每个文件都有一个绝对路径，以 `/` 开头，来表示从根目录到叶子节点的路径，例如 `/home/ubuntu/Desktop/my_project/test.py`，这种表示方法叫作绝对路径。

另外，对于任意两个文件，我们都有一条通路可以从一个文件走到另一个文件，例如 `/home/ubuntu/Downloads/example.json`。再如，我们从 `test.py` 访问到 `example.json`，需要写成 `'../../Downloads/example.json'`，其中 `..` 表示上一层目录。这种表示方法，叫作相对路径。

通常，一个 Python 文件在运行的时候，都会有一个运行时位置，最开始时即为这个文件所在的文件夹。当然，这个运行路径以后可以被改变。运行 `sys.path.append("../")`，则可以改变当前 Python 解释器的位置。不过，一般而言我并不推荐，固定一个确定路径对大型工程来说是非常必要的。

理清楚这些概念后，我们就很容易搞懂，项目中如何设置模块的路径。

首先，你会发现，相对位置是一种很不好的选择。因为代码可能会迁移，相对位置会使得重构既不雅观，也易出错。因此，在大型工程中尽可能使用绝对位置是第一要义。对于一个独立的项目，所有的模块的追寻方式，最好从项目的根目录开始追溯，这叫做相对的绝对路径。

事实上，在 Facebook 和 Google，整个公司都只有一个代码仓库，全公司的代码都放在这个库里。我刚加入 Facebook 时对此感到很困惑，也很新奇，难免会有些担心：

这样做似乎会增大项目管理的复杂度吧？

是不是也会有不同组代码隐私泄露的风险呢？

后来，随着工作的深入，我才发现了这种代码仓库独有的几个优点。

第一个优点，简化依赖管理。整个公司的代码模块，都可以被你写的任何程序所调用，而你写的库和模块也会被其他人调用。调用的方式，都是从代码的根目录开始索引，也就是前面提到过的相对的绝对路径。这样极大地提高了代码的分享共用能力，你不需要重复造轮子，只需要在写之前，去搜一下有没有已经实现好的包或者框架就可以了。

第二个优点，版本统一。不存在使用了一个新模块，却导致一系列函数崩溃的情况；并且所有的升级都需要通过单元测试才可以继续。


第三个优点，代码追溯。你可以很容易追溯，一个 API 是从哪里被调用的，它的历史版本是怎样迭代开发，产生变化的。

如果你有兴趣，可以参考这篇论文：


<https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext>

在做项目的时候，虽然你不可能把全世界的代码都放到一个文件夹下，但是类似模块化的思想还是要有的——那就是以项目的根目录作为最基本的目录，所有的模块调用，都要通过根目录一层层向下索引的方式来 import。


明白了这一点后，这次我们使用 PyCharm 来创建一个项目。这个项目结构如下所示：

 复制代码


```
1 .
2 |— proto
3 |   |— mat.py
4 |— utils
5 |   |— mat_mul.py
6 |— src
7     |— main.py
```

 复制代码

```
1 # proto/mat.py
2
3 class Matrix(object):
4     def __init__(self, data):
5         self.data = data
6         self.n = len(data)
7         self.m = len(data[0])
```

 复制代码

```
1 # utils/mat_mul.py
2
3 from proto.mat import Matrix
4
5 def mat_mul(matrix_1: Matrix, matrix_2: Matrix):
6     assert matrix_1.m == matrix_2.n
7     n, m, s = matrix_1.n, matrix_1.m, matrix_2.m
8     result = [[0 for _ in range(n)] for _ in range(s)]
9     for i in range(n):
10         for j in range(s):
11             for k in range(m):
12                 result[i][k] += matrix_1.data[i][j] * matrix_2.data[j][k]
13
14     return Matrix(result)
```

 复制代码

```
1 # src/main.py
2
3 from proto.mat import Matrix
4 from utils.mat_mul import mat_mul
5
```

```
6
7 a = Matrix([[1, 2], [3, 4]])
8 b = Matrix([[5, 6], [7, 8]])
9
10 print(mat_mul(a, b).data)
11
12 ##### 输出 #####
13
14 [[19, 22], [43, 50]]
```


这个例子和前面的例子长得很像，但请注意 `utils/mat_mul.py`，你会发现，它 `import Matrix` 的方式是 `from proto.mat`。这种做法，直接从项目根目录中导入，并依次向下导入模块 `mat.py` 中的 `Matrix`，而不是使用 `..` 导入上一级文件夹。

是不是很简单呢？对于接下来的所有项目，你都能直接使用 Pycharm 来构建。把不同模块放在不同子文件夹里，跨模块调用则是从顶层直接索引，一步到位，非常方便。

我猜，这时你的好奇心来了。你尝试使用命令行进入 `src` 文件夹，直接输入 `Python main.py`，报错，找不到 `proto`。你不甘心，退回到上一级目录，输入 `Python src/main.py`，继续报错，找不到 `proto`。

Pycharm 用了什么黑魔法呢？


实际上，Python 解释器在遇到 `import` 的时候，它会在一个特定的列表中寻找模块。这个特定的列表，可以用下面的方式拿到：

 复制代码

```
1 import sys
2
3 print(sys.path)
4
5 ##### 输出 #####
6
7 ['', '/usr/lib/python36.zip', '/usr/lib/python3.6', '/usr/lib/python3.6/lib-dynload', '']
```


请注意，它的第一项为空。其实，Pycharm 做的一件事，就是将第一项设置为项目根目录的绝对地址。这样，每次你无论怎么运行 main.py，import 函数在执行的时候，都会去项目根目录中找相应的包。

你说，你想修改下，使得普通的 Python 运行环境也能做到？这里有两种方法可以做到：

 复制代码


```
1 import sys
2
3 sys.path[0] = '/home/ubuntu/workspace/your_projects'
```

第一种方法，“大力出奇迹”，我们可以强行修改这个位置，这样，你的 import 接下来肯定就畅通无阻了。但这显然不是最佳解决方案，把绝对路径写到代码里，是我非常不推荐的方式（你可以写到配置文件中，但找配置文件也需要路径寻找，于是就会进入无解的死循环）。

第二种方法，是修改 PYTHONHOME。这里我稍微提一下 Python 的 Virtual Environment（虚拟运行环境）。Python 可以通过 Virtualenv 工具，非常方便地创建一个全新的 Python 运行环境。

事实上，我们提倡，对于每一个项目来说，最好要有一个独立的运行环境来保持包和模块的纯净性。更深的内容超出了今天的范围，你可以自己查资料了解。

回到第二种修改方法上。在一个 Virtual Environment 里，你能找到一个文件叫 activate，在这个文件的末尾，填上下面的内容：

 复制代码

```
1 export PYTHONPATH="/home/ubuntu/workspace/your_projects"
```

这样，每次你通过 activate 激活这个运行时环境的时候，它就会自动将项目的根目录添加到搜索路径中去。

神奇的 `if __name__ == '__main__':`

最后一部分，我们再来讲讲 `if __name__ == '__main__':`，这个我们经常看到的写法。

Python 是脚本语言，和 C++、Java 最大的不同在于，不需要显式提供 `main()` 函数入口。如果你有 C++、Java 等语言经验，应该对 `main() {}` 这样的结构很熟悉吧？

不过，既然 Python 可以直接写代码，`if __name__ == '__main__':` 这样的写法，除了能让 Python 代码更好看（更像 C++）外，还有什么好处吗？

项目结构如下：

 复制代码

```
1 .
2 |— utils.py
3 |— utils_with_main.py
4 |— main.py
5 |— main_2.py
```

 复制代码

```
1 # utils.py
2
3 def get_sum(a, b):
4     return a + b
5
6 print('testing')
7 print('{} + {} = {}'.format(1, 2, get_sum(1, 2)))
```

 复制代码

```
1 # utils_with_main.py
2
3 def get_sum(a, b):
4     return a + b
5
6 if __name__ == '__main__':
7     print('testing')
8     print('{} + {} = {}'.format(1, 2, get_sum(1, 2)))
```

```

1 # main.py
2
3 from utils import get_sum
4
5 print('get_sum: ', get_sum(1, 2))
6
7 ##### 输出 #####
8
9 testing
10 1 + 2 = 3
11 get_sum: 3

```

```

1 # main_2.py
2
3 from utils_with_main import get_sum
4
5 print('get_sum: ', get_sum(1, 2))
6
7 ##### 输出 #####
8
9 get_sum_2: 3

```

看到这个项目结构，你就很清晰了吧。

`import` 在导入文件的时候，会自动把所有暴露在外面的代码全都执行一遍。因此，如果你要把一个东西封装成模块，又想让它可以执行的话，你必须将要执行的代码放在 `if __name__ == '__main__':` 下面。

为什么呢？其实，`__name__` 作为 Python 的魔术内置参数，本质上是模块对象的一个属性。我们使用 `import` 语句时，`__name__` 就会被赋值为该模块的名字，自然就不等于 `__main__` 了。更深的原理我就不做过介绍了，你只需要明白这个知识点即可。

总结

今天这节课，我为你讲述了如何使用 Python 来构建模块化和大型工程。这里需要强调几点：

1. 通过绝对路径和相对路径，我们可以 import 模块；
2. 在大型工程中模块化非常重要，模块的索引要通过绝对路径来做，而绝对路径从程序的根目录开始；
3. 记着巧用 `if __name__ == '__main__':` 来避开 import 时执行。

思考题

最后，我想为你留一道思考题。`from module_name import *`和`import module_name`有什么区别呢？欢迎留言和我分享，也欢迎你把这篇文章分享给你的同事、朋友。

 极客时间

Python 核心技术与实战

系统提升你的 Python 能力



景霄
Facebook 资深工程师

新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 面向对象（下）：如何实现一个搜索引擎？

精选留言 (11)

写留言



书了个一
2019-06-07

4

老师端午节快乐!

展开 ▾



jim

2019-06-07

👍 4

from module_name import * 是导入module_name 内的所有内容，可以直接调用内部方法；import module_name，则是导入module_name，在代码中必须写成module_name.function的形式。

展开 ▾



小智e

2019-06-07

👍

老师的基础模块，帮助自己扫清了一下知识盲区，期待老师接下来的进阶篇，up,up,up。老师端午节快乐。



lllong33

2019-06-07

👍

不推荐使用*，会导入过多不需要的内容。

from module_name import *，可以直接使用类、函数、变量等
import module_name，需module_name.func()



SCAR

2019-06-07

👍

思考题：

from module_name import * 是把module_name下的所有变量或对象映射到当前命名空间，访问调用的话直接写变量或者变量对象调用即可。这种调用方式要注意一点的是如果变量和调用者里变量对象同名或者调用者import其他模块而来的变量同名的话，该变量会覆盖其他同名变量或者被其他同名变量所覆盖。...

展开 ▾



John Si

2019-06-07

👍

老师端午节快乐!

from import module_name import * 和 import module_的主要分别是 from import module_name import * 可直接调用module内的函数或值，假如不是用这种方式import的话，则需透过module_name.xxxxx 方式去调用。...

展开 ▾



canownu

2019-06-07



大家端午安康

展开 ▾



GentleCP

2019-06-07



from module import *之后，module里的函数类都可以直接用该名字调用，而import module调用其类和函数需要在前面加module.函数名（类名）的方式，建议采用第二种，因为这样可以知道某个函数，类是从哪个模块来的，第一种方式万一你有两个模块都有一个同名函数，调用就会出问题



mickle

2019-06-07



from module_name import * 可能和作用域中现有变量同名从而产生覆盖，而import module_name 是用module_name为前缀，不会同名。



□

2019-06-07



from module_name import * 是倒入module_name文件中所有的函数、类

import module_name 是倒入文件名，底层的模块需要以 module_name. 的方式进行调用

展开 ▾



王征

2019-06-07



import module_name之后，引用的时候要使用 module_name.function（或 class）的方式，from module_name import * 可以直接调用模块中的方法或类，不需要带模块名。

展开 ∨