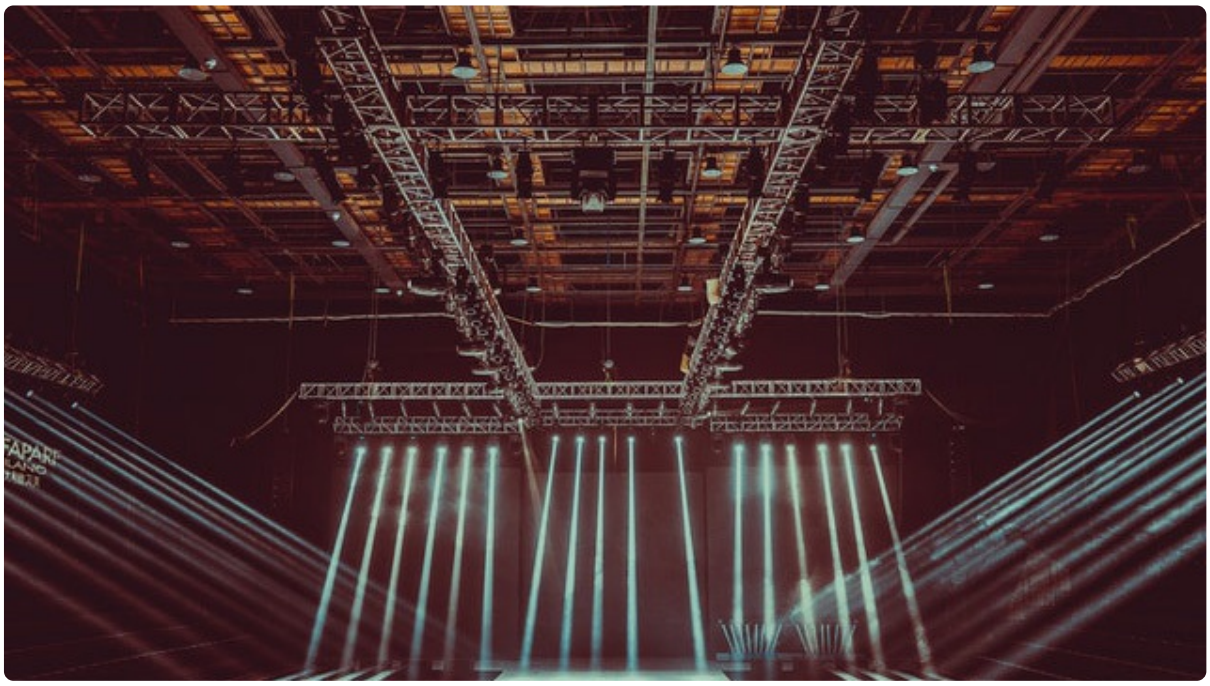
02 网络爬虫基础知识准备及基本开发环境介绍

更新时间：2019-07-03 13:57:46



> 如果说我比别人看得要远一点，那是因为我站在巨人的肩上。
>
> ——牛顿

本章将会介绍在开发网络爬虫项目时最常使用的 Python 技术内容，以及介绍 Python 的集成开发环境(IDE)。

- 网络爬虫中常用的 Python 基础知识
  - 包
  - 面向对象
  - 字典、列表与元组
  - 迭代器与`yield`
- 如何使用 Virtualenv
- Python 集成开发环境 PyCharm
  - 简介，下载
  - 如何配置 Virtualenv

爬虫中常用的 python 基础知识

Python中的包

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在 Python 中，一个.py文件就称之为一个模块（Module）。

使用模块有什么好处？

最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。点这里查看Python的所有内置函数。

你也许还想到，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python 又引入了按目录来组织模块的方法，称为包（Package）。

举个例子，一个abc.py的文件就是一个名字叫abc的模块，一个xyz.py的文件就是一个名字叫xyz的模块。

现在，假设我们的 abc 和 xyz 这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如 MyCompany，按照如下目录存放：

```
mycompany
├─ __init__.py
├─ abc.py
└─ xyz.py
```

引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，abc.py模块的名字就变成了MyCompany.abc，类似的，xyz.py的模块名变成了MyCompany.xyz。

请注意，每一个包目录下面都会有一个`__init__.py`的文件，这个文件是必须存在的，否则，Python 就把这个目录当成普通目录，而不是一个包。`__init__.py`可以是空文件，也可以有 Python代码，因为`__init__.py`本身就是一个模块，而它的模块名就是MyCompany。

类似的，可以有多级目录，组成多级层次的包结构。比如下面的目录结构：

```
mycompany
 ├─ web
 │  ├─ __init__.py
 │  ├─ utils.py
 │  └─ www.py
 ├─ __init__.py
 ├─ abc.py
 └─ xyz.py
```

文件www.py的模块名就是`mycompany.web.www`，abc.py的模块名则是`mycompany.abc`。

面向对象

面向对象编程——Object Oriented Programming，简称 OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

在 Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

那为什么学习爬虫技术要先了学习面向对象呢？从一个实际的例子入手就能更好地体验到面向对象所带来的好处了。

假设我们要爬取一个博客上面的文章，然后将这些文章的数据保存到本地，面向过程的程序可以用一个`dict`表示博客上的文章数据：

```
article1 = { 'title': '学习python的心得总结', 'pub_date': '2019/1/2', 'body': '正文内容省略' }
article2 = { 'title': '如何用python编写命令行工具', 'pub_date': '2019/1/20', 'body': '正文内容省略'  }
```

如果要打印文章的标题和发表时间，我们可以写一个函数进行处理：

```
def print_title(article):
    print('%s: %s' % (article['title'], article['pub_date']))
```

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是文章这种数据类型应该被视为一个对象，这个对象拥有`title`、`pub_date` 和 `body` 这三个属性（Property）。如果要打印一篇文章的标题与日期，首先必须创建出这篇文章对应的对象，然后，给对象发一个`print_title`消息，让对象自己把自己的数据打印出来，下面则是采用面向对象的方式来实现这个逻辑。

```
class Article(object):

    def __init__(self, title, pub_date, body):
        self.title = title
        self.pub_date = pub_date
        self.body = body

    def print_title(self):
        print('%s: %s' % (self.title, self.pub_date))
```

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

```
learn_python = Article('学习python的心得总结', '2019/1/2', '正文内容')
write_cmd = Article('如何用python编写命令行工具', '2019/1/20', '正文内容')
learn_python.print_title()
write_cmd.print_title()
```

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义的Class——Article，是指文章这个概念，而实例（Instance）则是一个个具体的`Article`，比如，`learn_python`和`write_cmd`是两个具体的文章。

所以，面向对象是一种代码复用的技术，同时也是一种对实现世界中某一类共通事物进行抽象的设计思想：抽象出Class，根据 Class 创建 Instance。

面向对象的抽象程度又比函数要高，因为一个类(`Class`)用属性存储数据，用方法(`Method`)表示行为。反过来说当我们要用计算表示对一类具有共同特性，共同行为的事物时就可以采用面向对象的方式定义一个类来表示。而当发现相似事物间的具有类似父子一般的传承关系时，则可以采用继承来实现。如我们要定义一个“新闻”类，它只是比“文章”类多出了一个“有效日期”的属性，那么我们就可以将“新闻”类视为从“文章”类中继承得到，在代码中则表示为：

```
class News(Article):

    def __init__(self,title, pub_date, body, expried):
        Article.__init__(self,title,pub_date,body) # 调用父类的__init__进行构造，称之 重载
        self.expried = expried
```

通过“继承”我们可以用最小量的代码构造出一个庞大的类族，不同的类具有各自的特性与行为，就像是一个小型的社会中不同的人具有不同的角色与职责一般，通过实例化去使用这些类就能构造出各色各样的系统。

“抽象”，“封装”，“继承” 就是构成了面向对象思想的三大基本要素，也是一种将现实世界翻译成逻辑世界的方法。

当你深入阅读本专栏的内容后你将会发现我们就是在不停地编写各种的蜘蛛类，数据类，中间件类等等。

Python 中的集合类型

从上文的例子中我们已经很清楚一个实例可以表示一条具体的数据，那么如果要表示多条相同类型的数据就要使用“集合”的方式。

Python 提供了多种类来处理不同的集合方式，分别有以下几种：

- `[]` - 数组 - 最常用的 Python 集合方式，也是衍生其它集合的基础
- `dict` - 字典 - 以"键-值"形式存对象
- `list` - 列表 - 数据列表，提供更多的方法操作列表的内容
- `set` - 集合 - 无序的不重复元素序列，可以用这种集合方式进行去重处理
- `tuple` - 元组 - 位置固定的有序元素组合

生成器、迭代器与`yield`

网络爬虫所需要处理的数据量通常都非常庞大，而对集合数据处理方式又大多采用循环(`for...in`)的方式，但如果直接使用循环一行一行地处理数据的话那性能将会是非常低下的，因此我们会采用更为高效的方式来处理循环，那就是生成器和迭代器

迭代是 Python 最强大的功能之一，是访问集合元素的一种方式。迭代器能记住遍历的位置的对象。从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：`iter()` 和 `next()`。字符串，列表或元组对象都可用于创建迭代器,举一个具体的例子：

```
>>>list=[1,2,3,4]
>>> it = iter(list)    # 创建迭代器对象
>>> print (next(it))   # 输出迭代器的下一个元素
1
>>> print (next(it))
2
>>>
```

> 注: `iter()` 和 `next()` 都是python的内置方法

生成器(`generator`)是非常强大的工具，在 Python 中，可以简单地把列表生成式改成`generator`，也可以通过函数实现复杂逻辑的`generator`。

要理解`generator`的工作原理，它是在`for`循环的过程中不断计算出下一个元素，并在适当的条件结束`for`循环。对于函数改成的`generator`来说，遇到`return`语句或者执行到函数体最后一行语句，就是结束`generator`的指令，`for`循环随之结束。

生成器(`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。

> 注：关于生成器与迭代器的内容展开非常详细，由于篇幅所限建议读者能仔细阅读在上文中的链接。因为在后面的内容中将会大量使用到生成器技术。

参考

- python3教程
- python3 cookbook 中文

如何使用 Virtualenv

Python 发行版本共存问题是一个为人诟病的东西，从Python2.x到现在的Python 3.x一直存在。Python2.x 一直非常流行，不少的Linux操作系统和MacOS都会预装一个Python2.x的运行环境，而且采用 Python2.x 编写的第三方工具包、框架的占比也极为庞大，但 Python 一直都希望整个开发社区都能与官方同步地迁移到最新的 Python 版本中来，虽然现在 Python3 也正在逐渐地改变这这一现况，但这个进度依然缓慢。

但作为开发者我们不得不面对着这样的一些问题：

1. 如何在同一台机器上运行不同Python开发的程序 ？
2. 如何让不同 Python 项目之间的项目依赖包不会发生版本的冲突 ？

还有就是在了解全局环境中通过`pip install`指令安装的依赖包存放在哪里，就很容易明白上文所提及的这两个问题的根源了。

首先，Python2.x 与 Python3.x 是可以同时安装在同一台机器上的，如果要进入各自的运行环境则是这样调用的：

```
$ python myscript.py  # 用python2.x执行代码
$ python3 myscript.py # 用python3.x执行代码
```

同一个大版本的Python是会互联覆盖的，也就是说如果先安装了Python 2.6 然后再安装Python2.7，那么在命令行运行`python`调出的就是 Python2.7的环境，同理Python3也是这样的。毕竟Python官方无奈地用了两个指令来直接解决大版本并存的问题。

每个 Python 环境的依赖包都会被默认安装到 Python目录下的`site-packages`目录中，所以Python2.x与Python3.x都有各自的`site-packages`目录。

用 `pip`指令就是将包安装到Python2.x的`site-packages`中，而`pip3`就会安装到Python3.x的`site-packages`中。到现在已经可以看出问题的一些端倪了吧，如果在同一机器上开发不同的项目，而不同的项目又会引用相同或者不同的依赖包时，即使是被安装到同一个Python大版本下也容易产生依赖冲突，或者说搞不清楚哪个项目使用了哪些依赖，因为它们都被安装到同一个全局目录下了。

正是为了解决这一连串的麻烦，每个应用就需要各自拥有一套“独立”的Python运行环境。Virtualenv就是为了能让独立的应用创建一套“隔离”的Python运行环境而设计的，而且他几乎是每个Python程序员必装的工具包。

首先，我们用pip安装Virtualenv：

```
$ pip install virtualenv  # 为python2.x安装 virtualenv
$ pip3 install virtualenv # 为python3.x安装 virutalenv
```

当我们要开发一个新的项目时，就需要一套独立的Python运行环境，可以这么做：

第一步，创建目录：

```
mac:~ ray$ mkdir myproject
mac:~ ray$ cd myproject/
mac:myproject ray$
```

第二步，创建一个独立的 Python 运行环境，命名为venv(这是一个约定俗成的命名，也可以用其他的名称，不过这么干的人不多)：

```
mac:myproject ray$ virtualenv --no-site-packages venv
Using base prefix '/usr/local/.../Python.framework/Versions/2.7'
New python executable in venv/bin/python2.7
Also creating executable in venv/bin/python
Installing setuptools, pip, wheel...done.
```

命令Virtualenv就可以创建一个独立的 Python 运行环境，参数`--no-site-packages`的作用是：已经安装到系统Python环境中的所有第三方包都不会复制过来。这样，我们就得到了一个不带任何第三方包的“干净”的 Python 运行环境。

如果要创建一个 Python3 的虚拟环境则可以这样做:

```
mac:myproject ray$ virtualenv --no-site-packages -p python3 venv
```

用 `-p python3`参数就可以生成一个基于Python3的运行环境当中。

新建的Python环境被放到当前目录下的venv目录。有了venv这个Python环境，可以用`source`指令激活该环境：

```
mac:myproject ray$ source venv/bin/activate
(venv)mac:myproject ray$
```

也可以写得更简单一点：

```
mac:myproject ray$ . venv/bin/activate
(venv)mac:myproject ray$
```

> 注意: 命令提示符变了，有个`(venv)` 前缀，表示当前环境是一个名为`venv` 的Python环境。

另外：

> **Windows 中激活Virtualenv**
>
> 在window环境中激活Virtualenv创建的虚拟环境不需要`source` 或者`.` ，因为`activate.exe` 是一个可执行文件，可以在终端中直接运行 `venv\Scripts\activate` 激活。

此时我们就可以不管它是 Python2.x 还是 Python3.x 的环境，只要在命令行中运行 `python`命令都会直接指向创建时指定的运行版本。

然后我们就可以正常安装各种第三方包，并运行pip命令：

```
(venv)mac:myproject ray$ pip install scrapy
DEPRECATION: Python 2.7 will reach the end of its life on January 1st, 2020. Please upgrade your Python as Python 2.7 won't
 be maintained after that date. A future version of pip will drop support for Python 2.7.
...
Successfully installed Automat-0.7.0 PyDispatcher-2.0.5 PyHamcrest-1.9.0 Twisted-19.2.0 asn1crypto-0.24.0 attrs-19.1.0 cffi
-1.12.2 constantly-15.1.0 cryptography-2.6.1 cssselect-1.0.3 enum34-1.1.6 functools32-3.2.3.post2 hyperlink-19.0.0 idna-2.8
 incremental-17.5.0 ipaddress-1.0.22 lxml-4.3.3 parsel-1.5.1 pyOpenSSL-19.0.0 pyasn1-0.4.5 pyasn1-modules-0.2.4 pycparser-2
.19 queuelib-1.5.0 scrapy-1.6.0 service-identity-18.1.0 six-1.12.0 w3lib-1.20.0 zope.interface-4.6.0
(venv)mac:myproject ray$ python myapp.py
...
```

> 从上面的安装输出结果我们都会得知这次python官方的升级决心，直接来狠的，python2.7版本到2020/1/1就不再维护了，也属于他们对仍然使用着python2.x的开发人员的最后通牒吧。

在venv环境下，用pip安装的包都被安装到venv这个环境下，系统Python环境不受任何影响。也就是说，venv环境是专门针对myproject这个应用创建的。

退出当前的venv环境，可以使用deactivate命令：

```
(venv)mac:myproject ray$ deactivate
mac:myproject ray$
```

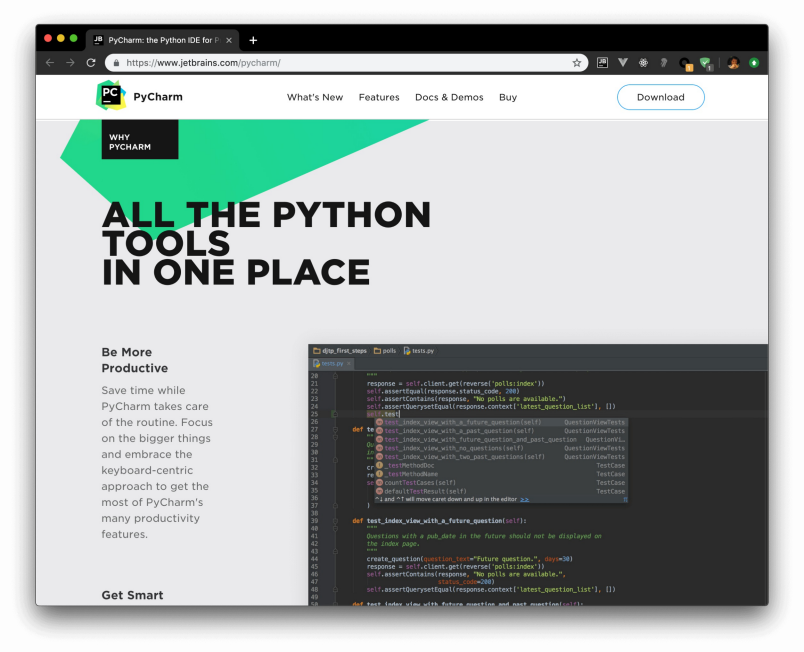
此时就回到了正常的环境，现在pip或python均是在系统Python环境下执行。

完全可以针对每个应用创建独立的Python运行环境，这样就可以对每个应用的Python环境进行隔离。

virtualenv是如何创建“独立”的Python运行环境的呢？原理很简单，就是把系统Python复制一份到virtualenv的环境，用命令`source venv/bin/activate`进入一个virtualenv环境时，virtualenv会修改相关环境变量，让命令python和pip均指向当前的virtualenv环境。

Python 开发利器 PyCharm

python　是一个开放的语言，也没有强制我们要采用什么方式来去编写python代码。有人喜欢用Vim，　有人喜欢用Sublime，有人喜欢用 VSCode这一类轻型的编辑器，但我更喜欢用 pyCharm 这款强大的IDE，我认为开发环境是越强大越好，相关工具配置只安装一次就能直接马上开干是最好的，而且pyCharm还为python配置了强大的调试器，这个大杀器对于代码的调试与测试起着至关重要的作用。

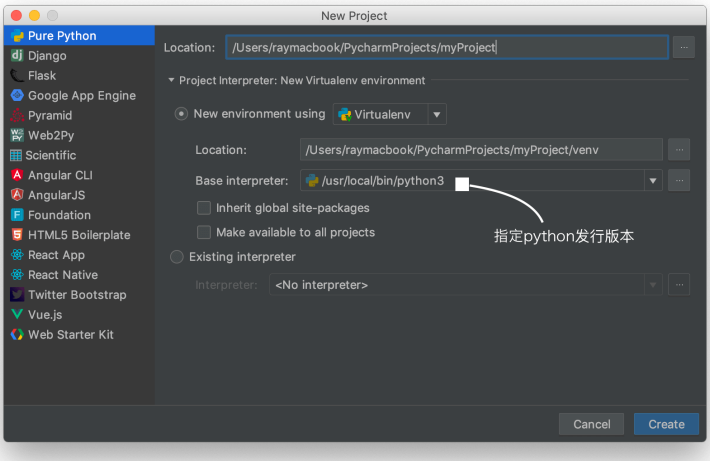


本专栏中所有的项目都会使用pyCharm进行开发与演示，其中也会介绍一些这个IDE强大的地方到底在哪里。

pyCharm的官网下载地址

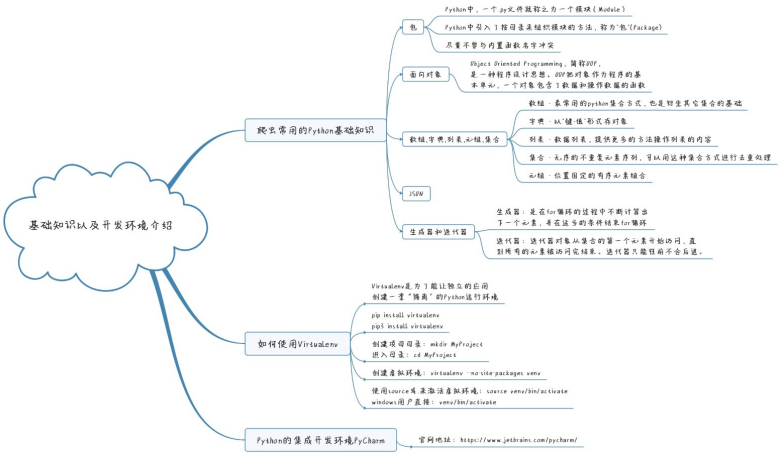
配置 virtualenv

用pyCharm就可以无需运行命令行来创建Virtualenv的虚环境了，一开始创建项目时它就已经为我们准备好了。



总结

Python 的博大精深并不能以一篇文章写尽，作为语言基础要系统化、全面地去学习才能真正灵活地运用。本节只是站在网络爬虫这一领域简略地指出哪些知识点是我们在后面的学习中会经常遇到的，你可以将本节所介绍的内容作为一种对 Python 持续学习的知识索引，将各个知识点持续地深化学习，在后面的例子中多动手，多理解，多思考，虽然是从0基础开始，但也能从实践中逐渐成长为个中能手。



01 导读：为什么要学爬虫？

03 HTTP 协议通信原理与 HTML 基础入门