



Python

编程入门与进阶

进 阶 篇

Python BIANCHENG RUMEN YU JINJIE

主编 韩迎春



北京邮电大学出版社
www.buptpress.com

Python 编程语言已有 30 余年的历史,目前广泛应用于各行业的软件开发领域。随着 Python 编程语言在大数据、云计算、人工智能、网络安全等领域广泛深入的应用,Python 编程成为各行业计算机从业者的必备技能,各高等院校、职业院校、培训机构大都把 Python 编程语言列为必修课程。

本书的作者团队由昆明冶金高等专科学校计算机信息学院的教师以及企业资深软件项目开发工程师组成。书中的知识结构依照 Python 编程语言的学习路径进行组织,并加入大量教学案例。

本书主要有以下几个特色。

(1)简化理论,强化编程。书中每个知识点的讲解都设置了代码示例,通过代码引导学生掌握编程思想,直观易懂。

(2)活页装订,板块丰富。教材采用活页式装订形式,便于读者根据自身情况抽取对应的章节进行学习,每章设置“思维导图”“知识清单”“本章总结”“术语”“程序阅读”“综合案例”“练习”“自我反思与评价”多个板块,使学习更具趣味性。

(3)注重实战,强化技能。每章中设置“程序阅读”和“综合案例”以强化技能训练,增强学生读懂代码和编写代码的能力。

(4)扫码观看,助力学习。本书针对重点和难点知识配有相应的微课视频,帮助学生解决疑难问题,扫描二维码即可获得微课以及程序代码资源,助力学生学习。

本书由韩迎春任主编,朵云峰任第一副主编,其余副主编有梁猛、袁建明、王瑞、王毅、何亚楠。

本书读者对象包括但不限于以下类型人员:Python 初学者、软件工程师、大学及中学教育工作者和学生、数据库工程师及管理员、技术或项目管理人员等。

由于作者水平有限,书中难免存在疏漏之处,敬请广大读者指正。

编 者

第 10 章 / Python 中的模块、包和库 1**10.1 模块 6**

10.1.1 编写一个模块 6

10.1.2 导入并使用模块 7

10.1.3 模块的导入顺序 9

10.1.4 重新导入模块 11

10.2 包 13

10.2.1 编写一个包 13

10.2.2 导入包中的模块或成员 14

10.3 库 16

10.3.1 标准库 17

10.3.2 使用 pip 管理第三方库 24

10.3.3 使用虚拟环境解决第三方库冲突 31

第 11 章 / Python 数据库存储 37**11.1 数据库概述 41****11.2 数据库存储操作 42**

11.2.1 PyMySQL 简介 42

11.2.2 PyMySQL 的安装 43

11.2.3 PyMySQL 连接 MySQL 数据库 44

11.2.4 PyMySQL 操作 MySQL 数据库 44

11.2.5 PyMySQL 封装 MySQL 数据库操作 50

第 12 章 / Python 网络编程 58**12.1 socket 64**

12.1.1 socket 中的常用数据类型 65

12.1.2	新建一个 socket 对象	66
12.1.3	绑定	68
12.1.4	监听	68
12.1.5	连接	68
12.1.6	接受连接申请	69
12.1.7	发送数据	70
12.1.8	接收数据	71
12.1.9	关闭连接	71
12.1.10	设置 socket 选项	72
12.1.11	设置 socket 超时	72
12.2	socket 案例	74
12.2.1	编写一个简单的 TCP 通信客户端和 服务端(标准流程)	74
12.2.2	编写一个 ping 客户端(RAW SOCKET)	75
12.3	scapy	78
12.3.1	scapy 的基本用法	78
12.3.2	构建数据包对象	82
12.3.3	发送数据	83
12.3.4	发送数据并接收响应	86
12.3.5	嗅探数据并做出响应	88
12.4	scapy 案例	90
12.4.1	交换机 MAC 泛洪(只发不收)	90
12.4.2	DNS 劫持(嗅探并响应)	90
12.4.3	TCP SYN 半连接端口扫描器(发送并 接收响应)	91

第 13 章 / Python 网络爬虫

95

13.1	Requests	99
13.1.1	Requests 简介与安装	99
13.1.2	使用 Requests 直接发送请求	99
13.1.3	使用 Requests 以会话方式发送请求	100

13.1.4	Response 对象	101
13.1.5	高级用法	103
13.2	BeautifulSoup	106
13.2.1	BeautifulSoup 简介与安装	106
13.2.2	BeautifulSoup 中的基本数据类型	107
13.2.3	BeautifulSoup 遍历文档树	110
13.2.4	BeautifulSoup 查找文档树	112
13.3	Selenium	114
13.3.1	Selenium 简介与安装	114
13.3.2	Selenium 的基本使用	116
13.3.3	Selenium 选择器	117
13.3.4	Selenium 等待元素被加载	118
13.3.5	Selenium 元素交互	118
13.3.6	Selenium 切换 frame 和窗口	120
13.3.7	Selenium 关闭浏览器	121
13.4	Scrapy	121
13.4.1	Scrapy 简介与安装	121
13.4.2	Scrapy 常用命令	122
13.4.3	爬虫项目结构及爬虫应用简介	123
13.4.4	Spiders	124
13.4.5	Selectors	126
13.4.6	Item 与 Pipeline	126
13.4.7	下载中间件	128
13.4.8	爬虫中间件	129

第 14 章 / Python GUI 编程 **134**

14.1	GUI 介绍	139
14.2	Python GUI 框架介绍	140
14.2.1	跨浏览器 GUI 框架	140
14.2.2	跨平台 GUI 框架	140
14.2.3	特定平台 GUI 框架	141

14.3	Python 内置 GUI 框架 Tkinter	141
14.3.1	Tkinter 架构	141
14.3.2	Tkinter 的基本概念	142
14.3.3	Tkinter 开发 GUI 程序的结构	142
14.3.4	Tkinter 的基本组件	144
14.3.5	Tkinter 中的事件处理	186
14.3.6	Tkinter 对话框组件	191

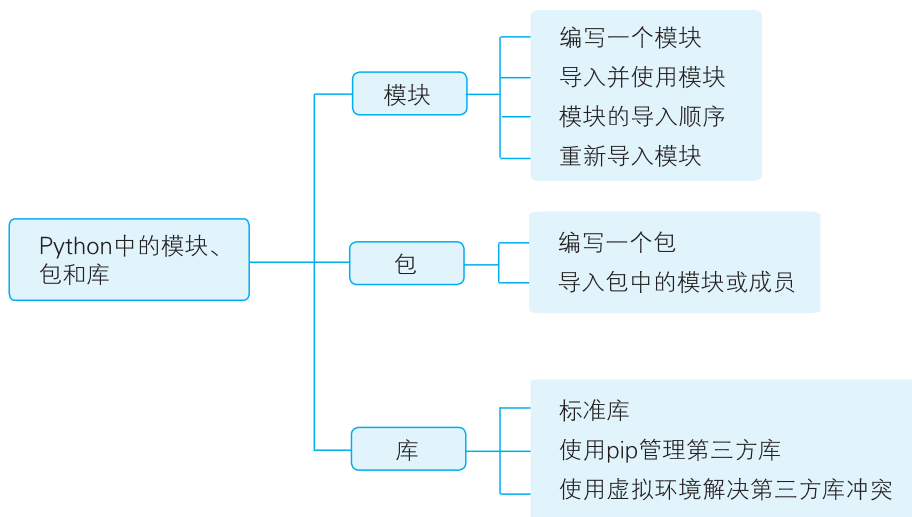
第 15 章 / **Python Web 开发** **196**

15.1	CGI 和 WSGI	201
15.1.1	CGI 介绍	201
15.1.2	WSGI 介绍	201
15.2	Python 三大主流 Web 框架	202
15.2.1	Django	202
15.2.2	Flask	202
15.2.3	Tornado	202
15.3	基于 Django 的 Web 开发	203
15.3.1	安装 Django	203
15.3.2	创建基于 Django 的 project	203
15.3.3	Django 模板	206
15.3.4	Django 模型	207
15.3.5	Django 视图	211
15.3.6	Django 路由	213
15.3.7	Django Cookie/Session 对象	214
15.3.8	Django Admin 管理工具	217



第 10 章

Python中的模块、 包和库

 思维导图



知识清单

课题	Python 中的模块、包和库				
班级		姓名		学号	
学习目标	1. 理解 Python 中的模块、包和库的含义,能够编写和使用包与模块; 2. 学习使用常用标准库,能够通过阅读官方文档了解标准库; 3. 使用 pip 管理第三方库; 4. 理解虚拟环境的原理,能够使用虚拟环境隔离第三方库				
重点/难点	重点:使用 import 导入库,使用 pip 管理第三方库,标准库的使用,虚拟环境的使用; 难点:环境变量在 Python 中的作用				
学习资源	https://www.zhihu.com/question/30082392 ; https://www.runoob.com/w3cnote/python-pip-install-usage.html ; https://www.cnblogs.com/f-ck-need-u/p/9955485.html ; https://blog.csdn.net/qq_39241986/article/details/106654549				
任务	任务 1:练习模块和包导入的几种方法。 学习记录:				
	任务 2:练习调整库搜索路径的几种方法。 学习记录:				
	任务 3:练习使用 pip。 学习记录:				
	任务 4:练习使用虚拟环境。 学习记录:				



案例导入

案例描述:

使用 Python 对访问学校网站首页的响应时间做一个简单的统计,重复执行 5 次并取平均值。

实现代码如下。

```
import requests,time
sumtime = 0
for i in range(1,6):
    starttime = time.time()
    res = requests.get(url = 'http://www.kmyz.edu.cn')
    endtime = time.time()
    spendtime = endtime - starttime
    print("第 %d 次: %.2fms" % (i,(spendtime * 1000)))
    sumtime = sumtime + spendtime
avgtime = sumtime / 5
print("平均值: {:.2f}ms".format(avgtime * 1000))
```

运行输出结果如下。

```
第 1 次:58.00ms
第 2 次:68.10ms
第 3 次:60.19ms
第 4 次:48.05ms
第 5 次:67.53ms
平均值:60.38ms
```

案例解读:

上述代码在全新安装的 Python 环境中是无法运行的,会提示缺少名为 requests 的模块(module)。

```
C:\Users\test\Desktop\test>python 1.py
Traceback (most recent call last):
  File "C:\Users\test\Desktop\test\1.py", line 1, in <module>
    import requests,time
ModuleNotFoundError: No module named 'requests'
```

为了让程序能够正常运行,需要在 Python 环境中使用 pip 引入名为 requests 的库(lib)。


```
C:\Users\test\Desktop\test>pip install requests
Defaulting to user installation because normal site-packages is not writeable
Collecting requests
  Using cached requests-2.27.1-py2.py3-none-any.whl (63 KB)
Requirement already satisfied: idna<4,>=2.5 in c:\users\test\AppData\Roaming\Python\Python310\site-packages (from requests) (3.3)
Requirement already satisfied: certifi>=2017.12.17 in c:\users\test\AppData\Roaming\Python\Python310\site-packages (from requests) (2021.10.8)
Requirement already satisfied: charset-normalizer~=2.0.0 in c:\users\test\AppData\Roaming\Python\Python310\site-packages (from requests) (2.0.12)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\test\AppData\Roaming\Python\Python310\site-packages (from requests) (1.26.8)
Installing collected packages: requests
Successfully installed requests-2.27.1
```

从上面的安装过程可以看出,在安装 requests 时,需要同时安装 idna、certifi、charset-normalizer、urllib3 四个库,并且对部分库的最高版本和最低版本有要求。例如,需要使用 pip 安装一个名为 demo 的第三方库,而 demo 要求 idna 版本大于 4,这时就会和 requests 对 idna 的版本要求产生冲突,当 requests 和 demo 不在同一个项目中使用,一般会使用虚拟环境来解决。

上面引入 time 模块时没有提示错误,这是因为 time 模块属于内置模块,内置模块是用 C 语言编写的,提供了对系统功能的访问。在安装 Python 时已经内置在 Python 解释器中,不需要额外引入。查看内置模块的方法如下。

```
>>> import sys
>>> print(sys.builtin_module_names)
('_abc', '_ast', '_bisect', '_blake2', '_codecs', '_codecs_cn', '_codecs_hk',
'_codecs_iso2022', '_codecs_jp', '_codecs_kr', '_codecs_tw', '_collections',
'_contextvars', '_csv', '_datetime', '_functools', '_heapq', '_imp', '_io', '_json',
'_locale', '_lsprof', '_md5', '_multibytecodec', '_opcode', '_operator', '_pickle',
'_random', '_sha1', '_sha256', '_sha3', '_sha512', '_signal', '_sre', '_stat',
'_statistics', '_string', '_struct', '_symtable', '_thread', '_tracemalloc', '_warnings',
'_weakref', '_winapi', '_xxsubinterpreters', 'array', 'atexit', 'audioop', 'binascii',
'builtins', 'cmath', 'errno', 'faulthandler', 'gc', 'itertools', 'marshal', 'math', 'mmap',
'msvcrt', 'nt', 'sys', 'time', 'winreg', 'xxsubtype', 'zlib')
```

内置模块不需要使用 pip 安装,标准库模块也不需要安装,如 datetime 模块,可以在 Python 安装文件夹下的 lib 目录中找到它,它是一个后缀名为 py 的文件或一个文件夹。通常情况下,也会将内置模块归为标准库的范围,后文不再对内置模块和标准库进行区分。



10.1 模块

Python 模块(module)是一个 Python 文件,以 .py 结尾,包含了 Python 对象定义和 Python 语句。任何 py 文件都可以作为一个模块。其他可作为 module 的文件类型还有 .so、.pyo、.pyc、.dll、.pyd,但 Python 初学者几乎用不到。通过使用模块,可以有效地避免命名空间的冲突,可以隐藏代码细节,使我们专注于高层的逻辑,还可以将一个较大的程序分为多个文件,提升代码的可维护性和可重用性。模块能定义函数、类、变量,也能包含可执行的代码。module 来源有以下三种。

- (1)Python 内置模块(标准库)。
- (2)第三方模块。
- (3)自定义模块。

10.1.1 编写一个模块

通过模块的定义得知模块是一个 Python 文件,以 .py 结尾,下面是个简单的自定义模块 demo,模块中定义了一个 hello 函数。

```
demo.py
def hello(arg):
    print("This is a demo module:" + str(arg))
    return
```

模块的说明文档放在 py 文件的开头,用成对的三个英文单引号引起来,还可以使用 `if __name__ == '__main__':` 对模块进行调试,如果该 py 文件是以模块的方式被调用的,那么其内置成员变量 `__name__` 的值为该模块的名字,如果直接运行该模块,那么其内置成员变量 `__name__` 的值为 `__main__`。

```
demo.py
'''This is demo info.'''
def hello(arg):
    print("This is a demo module:" + str(arg))
    return
if __name__ == '__main__':
    hello('demo debug')
```

```
D:\test>python demo.py
This is a demo module;demo debug
```

可以通过模块的内置成员变量 `__doc__` 输出模块的说明文档。

```
>>> import demo
>>> demo.__doc__
'This is demo info.'
```

10.1.2 导入并使用模块

先看导入模块的两种方式。

```
import [文件夹.]模块名 1 [as 别名 1], [文件夹.]模块名 2 [as 别名 2], ...
from [文件夹.]模块名 import 成员名 1 [as 别名 1], 成员名 2 [as 别名 2], ...
```

模块编写好后,可以使用 import 语句导入并使用模块。

```
>>> import demo
>>> demo.hello("demo")
This is a demo module;demo
```

也可以这样导入。

```
>>> from demo import hello
>>> hello("demo")
This is a demo module;demo
```

1. 使用 as 为模块起别名

如果自定义模块名字较长,如模块名为 ihavealongmodulename,那么再调用模块时如何缩短模块名以方便编写和阅读代码?

ihavealongmodulename 模块代码如下。

```
ihavealongmodulename.py
def test():
    print("my name is so long")
    return
```

大家可以尝试使用上面提到的方法使用 ihavealongmodulename 模块中的 test 方法,会发现代码冗长。这里为了使代码简化,可以使用 as 为模块起个别名。

```
>>> import ihavealongmodulename as ln
>>> ln.test()
my name is so long
```

2. 使用 as 为模块中的成员起别名

例如,自定义了 demo1 模块和 demo2 模块,两个模块中都有 hello 函数,这时如何在一个程序中同时使用 demo1 中的 hello 函数和 demo2 中的 hello 函数?

demo1 模块代码如下。

```
demo1.py
def hello(arg):
    print("hello1")
    return
```

demo2 模块代码如下。

```
demo2.py
def hello(arg):
    print("hello2")
    return
```

方法是在 import 相关模块时,使用 as 为函数起个别名。

```
>>> from demo1 import hello as hello1
>>> from demo2 import hello as hello2
>>> hello1()
hello1
>>> hello2()
hello2
This is a demo module;demo
```

3. 导入其他文件夹中的模块

如果要使用的模块在子文件夹中,可以使用“import 文件夹名. 模块名”或“from 文件夹名. 模块名 import 成员名”的方式导入。

假设工作目录为 d:\test,而模块的路径为 d:\test\demo\demo.py,可以使用 import demo. demo 来导入需要的模块。

4. 将模块中所有成员导入当前命名空间

如果 demo 模块中包含了很多成员,想一次性将这些成员全部导入当前的命名空间中,可以使用通配符 * 来解决。

```
demo.py
x = 1
y = 2
z = 3

>>> from demo import *
>>> x
1
>>> y
2
```

```
>>> z
3
```

使用通配符 * 时一定要注意成员名不能和现有命名空间中的成员名重名,否则会覆盖当前命名空间中的成员,因此一般不推荐使用这种方法。

10.1.3 模块的导入顺序

可以使用 import 导入并使用模块,前面演示的是导入自定义模块,要求自定义模块必须在当前运行目录下。例如,demo 模块的存储路径为 d:\test\demo.py,那么需要先在 cmd 中转到 d:\test 下,再运行 Python 才能执行 import demo,否则会报错,提示找不到 demo 模块。当然也可以用其他 Python 文件来导入 demo 模块,同样要求它们在同一个目录下。

但是使用标准库中的模块,如 datetime 时,无论当前运行目录是什么,也无论导入 datetime 模块的 Python 文件在哪里,都可以成功导入。

这是因为 Python 在导入模块时会按照一定的顺序在计算机磁盘上进行搜索,而这个搜索顺序就存放在 sys 模块的 path 字符串列表中,它根据环境变量 PYTHONPATH 进行初始化,再加上安装时的默认值。此列表的第一项 path[0]在程序启动时初始化,是包含调用 Python 解释器的脚本的目录。如果脚本目录不可用(如解释器以交互方式调用),path[0]是空字符串,表示 Python 首先搜索当前目录中的模块。从 path[1]开始是从环境变量 PYTHONPATH 中读取的,其后部分是安装时的默认值。

```
C:\Users\test>set PYTHONPATH
PYTHONPATH = d:\test

>>> import sys
>>> type(sys.path)
<class 'list'>
>>> sys.path
['', 'd:\test', 'C:\Program Files\Python310\python310.zip', 'C:\Program
Files\Python310\DLLs', 'C:\Program Files\Python310\lib', 'C:\Program Files\
Python310', 'C:\Users\test\AppData\Roaming\Python\Python310\site-packages',
'C:\Program Files\Python310\lib\site-packages']
```

可以通过调整 PYTHONPATH 环境变量永久改变搜索路径,也可以通过修改 sys.path 临时修改搜索路径来达到改变搜索路径的目的。

下面的例子演示了模块导入的先后顺序,在 d:\test\1 目录下和 d:\test\2 目录下同时存在 demo 模块,可以通过调整 sys.path 的方式调整导入的 demo 模块。

```
D:\test\1>type demo.py
def hello():
    print("test1")
```

```

    return

D:\test\2>type demo.py
def hello():
    print("test2")
    return

```

下面使用交互式方式在 d:\test 目录下打开 Python 并输出 sys.path。

```

D:\test>python
Python 3.10.2 (tags/v3.10.2:a58ebcc, Jan 17 2022, 14:12:15) [MSC v.1929 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', 'd:\test', 'C:\Program Files\Python310\python310.zip', 'C:\Program
Files\Python310\DLLs', 'C:\Program Files\Python310\lib', 'C:\Program Files\
Python310', 'C:\Users\test\AppData\Roaming\Python\Python310\site-packages',
'C:\Program Files\Python310\lib\site-packages']

```

可以发现 sys.path 中并没有 d:\test\1 目录和 d:\test\2 目录,并且当前目录和 sys.path 所包含的其他目录中也没有 demo.py 文件,根据上面提到的规则,导入 demo 模块时应当会报错。

```

>>> import demo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'demo'

```

现在修改 sys.path,使其能够导入 d:\test\1 目录下的 demo 模块。

```

>>> sys.path.insert(0,"d:\test\1")
>>> sys.path.insert(1,"d:\test\2")
>>> sys.path
['d:\test\1', 'd:\test\2', '', 'd:\test', 'C:\Program Files\Python310\
python310.zip', 'C:\Program Files\Python310\DLLs', 'C:\Program Files\Python310
\lib', 'C:\Program Files\Python310', 'C:\Users\test\AppData\Roaming\Python\
Python310\site-packages', 'C:\Program Files\Python310\lib\site-packages']
>>> import demo
>>> demo.hello()
test1

```

同理,如果 d:\test\2 在前面,会优先导入 d:\test\2 下面的 demo 模块。

除了通过修改 PYTHONPATH 环境变量和设置 sys.path 之外,还可以通过定义 .pth 文件将模块的搜索路径导入 sys.path 中。.pth 文件一行设置一个路径,.pth 的存放位置可以通过 site 模块的 getsitepackages 函数来获取,如果需要将 .pth 文件放在其他路径下,可通过 site.addsitedir('')增加查找路径。

```
D:\test>type a.pth
d:\test\1
d:\test\2

>>> import site
>>> site.getsitepackages()
['C:\Program Files\Python310', 'C:\Program Files\Python310\lib\site-packages']
>>> site.addsitedir('d:\test')
>>> sys.path
['', 'd:\test', 'C:\Program Files\Python310\python310.zip', 'C:\Program Files\Python310\DLLs', 'C:\Program Files\Python310\lib', 'C:\Program Files\Python310', 'C:\Users\test\AppData\Roaming\Python\Python310\site-packages', 'C:\Program Files\Python310\lib\site-packages', 'd:\test\1', 'd:\test\2']
```

通过上面的例子可以发现,通过 .pth 文件导入的路径会放在 sys.path 的最后。

10.1.4 重新导入模块

由于 Python 所特有的属性,同一个模块在重复导入时只会导入一次。假设 a.py 中导入了模块 b 和模块 sys,在 b.py 中也导入了模块 sys,但 Python 默认对每个模块只会导入一次,如果 a.py 中先导入 sys,再导入 b 时会发现 sys 已经导入了,不会再去导入 sys。

这是因为 Python 执行程序时,会将所有已经导入的模块放进 sys.modules 属性中,这是一个 dict,可以通过下面的方式来查看已导入的模块名。

```
>>> sys.modules
{'sys': <module 'sys' (built-in)>, 'builtins': <module 'builtins' (built-in)>, '_frozen_importlib': <module '_frozen_importlib' (frozen)>, '_imp': <module '_imp' (built-in)>, '_thread': <module '_thread' (built-in)>, '_warnings': <module '_warnings' (built-in)>, '_weakref': <module '_weakref' (built-in)>, '_io': <module '_io' (built-in)>, 'marshal': <module 'marshal' (built-in)>, 'nt': <module 'nt' (built-in)>, 'winreg': <module 'winreg' (built-in)>, '_frozen_importlib_external': <module '_frozen_importlib_external' (frozen)>, 'time': <module 'time' (built-in)>, 'zipimport': <module 'zipimport' (frozen)>, '_codecs': <module '_codecs' (built-in)>,
```

```
'codecs': <module 'codecs' from 'C:\Program Files\Python310\lib\codecs.py'>,
'encodings.aliases': <module 'encodings.aliases' from 'C:\Program Files\
Python310\lib\encodings\aliases.py'>, 'encodings': <module 'encodings' from
'C:\Program Files\Python310\lib\encodings\__init__.py'>, 'encodings.utf_8':
<module 'encodings.utf_8' from 'C:\Program Files\Python310\lib\encodings\utf_
8.py'>, '_codecs_cn': <module '_codecs_cn' (built-in)>, '_multibytecodec':
<module '_multibytecodec' (built-in)>, 'encodings.gbk': <module 'encodings.
gbk' from 'C:\Program Files\Python310\lib\encodings\gbk.py'>, '_signal':
<module '_signal' (built-in)>, '_abc': <module '_abc' (built-in)>, 'abc':
<module 'abc' from 'C:\Program Files\Python310\lib\abc.py'>, 'io': <module
'io' from 'C:\Program Files\Python310\lib\io.py'>, '__main__': <module '__
main__' (built-in)>, '_stat': <module '_stat' (built-in)>, 'stat': <module
'stat' from 'C:\Program Files\Python310\lib\stat.py'>, '_collections_abc':
<module '_collections_abc' from 'C:\Program Files\Python310\lib\_collections_
abc.py'>, 'genericpath': <module 'genericpath' from 'C:\Program Files\
Python310\lib\genericpath.py'>, 'ntpath': <module 'ntpath' from 'C:\Program
Files\Python310\lib\ntpath.py'>, 'os.path': <module 'ntpath' from 'C:\Program
Files\Python310\lib\ntpath.py'>, 'os': <module 'os' from 'C:\Program Files\
Python310\lib\os.py'>, '_sitebuiltins': <module '_sitebuiltins' from 'C:\
Program Files\Python310\lib\_sitebuiltins.py'>, 'site': <module 'site' from
'C:\Program Files\Python310\lib\site.py'>, 'atexit': <module 'atexit' (built-
in)>, 'demo': <module 'demo' from 'D:\test\demo.py'>}
```

sys.modules 中每一项的 key 为已导入的模块名, value 为模块对象的描述。

如果某个程序文件中多次使用 import(或 from) 导入同一个模块, 虽然不会报错, 但实际上还是直接使用内存中已装载好的模块对象。

例如, demo.py 中 x=3, 导入它之后修改该值, 再次导入, 发现 demo.x 并未发生改变, 如果 demo 模块已经导入, 修改 demo.py 的代码后再次导入也会发现修改未生效。

```
demo.py
def hello(arg):
    print("This is a demo module:" + str(arg))
    return
x = 3

>>> import demo
>>> demo.x
3
>>> demo.x = 33
```



```
>>> import demo
>>> demo.x
33
//修改 demo.py 文件中的 x = 44, 重新导入
>>> import demo
>>> demo.x
33
```

如果需要重新导入模块, 可以使用 `importlib` 模块中的 `reload` 函数。

```
>>> import demo, importlib
>>> demo.x
3
>>> demo.x = 33
>>> demo.x
33
>>> importlib.reload(demo)
<module 'demo' from 'D:\test\demo.py'>
>>> demo.x
3
//修改 demo.py 文件中 x = 44, 重新导入
>>> importlib.reload(demo)
<module 'demo' from 'D:\test\demo.py'>
>>> demo.x
44
```

10.2 包

在比较大型的项目中常常需要编写或用到大量的模块, 此时可以使用包(package)来管理这些模块。包是一个分层次的文件目录结构, 它定义了一个由模块及子包, 和子包下的子包等组成的 Python 应用环境。

简单来说, 包就是文件夹, 但该文件夹下必须存在 `__init__.py` 文件, 该文件的内容可以为空。`__init__.py` 用于标识当前文件夹的是一个包。

10.2.1 编写一个包

- (1) 创建一个文件夹 `packagedemo`。
- (2) 在文件夹中创建空的 `__init__.py` 文件。
- (3) 在 `packagedemo` 包中创建两个模块文件 `demo1.py` 和 `demo2.py`, `demo1.py` 定义了一个变量 `demo1=1`, `demo2.py` 定义了一个变量 `demo2=2`。

10.2.2 导入包中的模块或成员

包的导入方式与模块类似,有如下三种。

```
import 包名[. 模块名/子包名 [as 别名]]
from 包名 import 模块名/子包名/成员名 [as 别名]
from 包名. 模块名/子包名 import 成员名 [as 别名]
```

下面依次通过不同包结构的案例来分析这几种导入方式。

1. import 包名[. 模块名/子包名 [as 别名]]

分析第一种导入方式“import 包名”。这种方式的本质是导入包中的__init__模块,与模块导入后的效果相同,只不过模块名为文件夹的名字。下面导入 10.2.1 小节中的包。

```
>>> import packagedemo
```

由于__init__.py 文件为空,相当于导入了一个空模块,虽然导入成功了,但是不会有任何实际作用。一般情况下,应当在__init__.py 文件中对成员进行初始化。下面修改__init__.py 文件,以让用户能够导入 demo1 中的 value1 变量和 demo2 中的 value2 变量。

```
__init__.py
from .demo1 import value1
from .demo2 import value2

>>> import packagedemo
>>> packagedemo.value1
1
>>> packagedemo.value2
2
```

注意上面__init__.py文件中导入包中模块是在模块名前面加了“.”,表示从当前目录开始查找模块,如果不加“.”,就会根据 sys.path 中的路径进行查找,而 packagedemo 目录并不在模块搜索路径中,因此会导入失败。

上面__init__.py 文件是将 demo1 和 demo2 中的成员 value1 和 value2 导入了自己的命名空间中,当然也可以直接将 demo1 和 demo2 两个模块导入自己的命名空间中。

```
__init__.py
from . import demo1
from . import demo2

>>> import packagedemo
>>> packagedemo.demo1.value1
1
```

```
>>> packagedemo.demo2.value2
2
```

如果只想导入 packagedemo 中的 demo1 模块,可以进行如下操作。

```
>>> import packagedemo.demo1
>>> packagedemo.demo1.value1
1
```

2. from 包名 import 模块名/子包名/成员名 [as 别名]

前文导入 packagedemo 中的 demo1 模块后,大家发现调用时比较麻烦,需要写明包名和模块名,因此如果只想导入包中的某个模块,推荐使用下面的方法。

```
>>> from packagedemo import demo1
>>> demo1.value1
1
```

这样 demo1 就会进入当前主程序的命名空间中,可以直接被调用。

在前面的例子中,__init__.py 文件本身没有包含任何成员,如果要调用__init__.py 自身命名空间中的成员,那么可以使用下面的方法:先修改__init__.py 文件,将 demo1 中的 value1 和 demo1 中的 value2 导入__init__.py 文件的命名空间中,然后新建一个 value0 成员。

```
__init__.py
from .demo1 import value1
from .demo2 import value2
value0 = 0
```

然后使用下面的方法将__init__.py 文件中的三个成员导入主程序的命名空间中。

```
>>> from packagedemo import value1, value2, value0
>>> value1
1
>>> value2
2
>>> value0
0
```

如果需要导入多个模块或成员,这种方法需要将每个模块和成员的名字都写出来,太过烦琐,__init__.py 提供了一个变量__all__,可以批量导入需要的模块或成员。在__init__.py 文件中添加__all__。

```
__init__.py
__all__ = ['value1', 'value2']
from .demo1 import value1
from .demo2 import value2
value0 = 0
```

然后可以使用通配符 * 来指代__all__中的内容。

```
>>> from packagedemo import *
>>> value1
1
>>> value2
2
>>> value0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

因为__all__中没有 value0,所以 value0 没有被调入当前的命名空间中。

3. from 包名.模块名/子包名 import 成员名 [as 别名]

这种方法一般用在包中还有子包的情况,在 packagedemo 中再创建一个子包 subpack,子包包含两个模块 subdemo1.py 和 subdemo2.py,分别有两个成员 subvalue1 = 1 和 subvalue2 = 2,子包的__init__.py 文件将 subvalue1 和 subvalue2 导入了自己的命名空间中。现在需要将子包中的所有成员导入当前命名空间中,可以进行如下操作。

```
>>> from packagedemo.subpack import *
>>> subvalue1
1
>>> subvalue2
2
```

10.3 库

严格来说,Python 中是没有库(library)的,模块(module)和包(package)都是 Python 语法中的概念,而库只是一个通俗的说法。平时说的库既可以是一个模块也可以是一个包。证据就是可以打开 Python 的安装目录,里面有一个名称为 Lib 的文件夹,打开这个文件夹之后就会发现很多“标准库”,如人们常用的 os 库、random 库、re 库等,这些“库”都是一个个的 Python 文件(模块);而 json、collections、sqlite3 和 tkinter 这些“库”又是一个个的文件夹(里面都有__init__.py),也就是包,如图 10-1 所示。

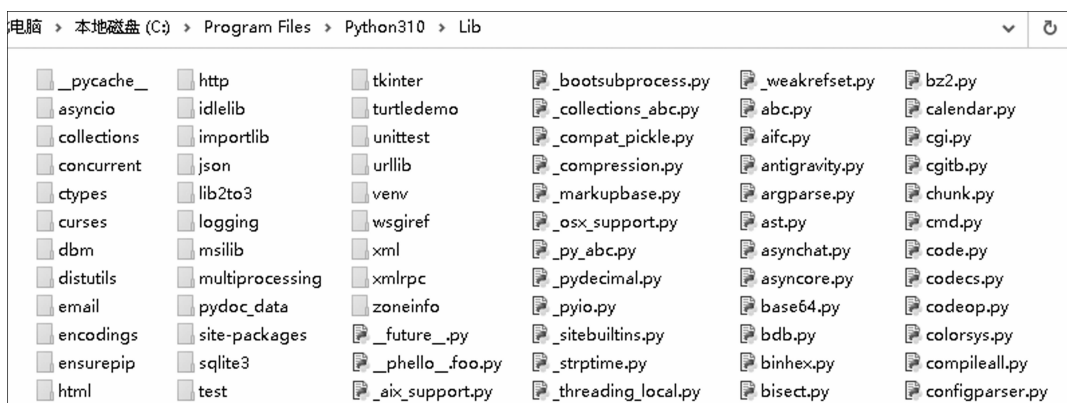


图 10-1

10.3.1 标准库

Python 标准库非常庞大,所提供的组件涉及范围十分广泛,正如以下内容目录所显示的。标准库包含了多个内置模块(用 C 语言编写),Python 程序员必须依靠它们来实现系统级功能,如文件 I/O,此外还有大量以 Python 编写的模块,提供了日常编程中许多问题的标准解决方案。其中有些模块经过专门设计,通过将特定平台功能抽象化为平台中立的 API 来鼓励和加强 Python 程序的可移植性。

Windows 版本的 Python 安装程序通常包含整个标准库,还包含许多额外组件。对于类 UNIX 操作系统,Python 通常会分成一系列软件包,因此可能需要使用操作系统所提供的包管理工具来获取部分或全部可选组件。标准库的默认位置在 Python 安装目录下的 Lib 文件夹中,如图 10-2 所示,但不包含 Lib 下的 site-packages 文件夹,site-packages 文件夹存储的是第三方库。

此电脑 > 本地磁盘 (C:) > Program Files > Python310 > Lib				
名称	修改日期	类型	大小	
crypt.py	2022/1/17 14:25	Python File	4 KB	
csv.py	2022/1/17 14:25	Python File	17 KB	
dataclasses.py	2022/1/17 14:25	Python File	57 KB	
datetime.py	2022/1/17 14:25	Python File	89 KB	
decimal.py	2022/1/17 14:25	Python File	1 KB	
difflib.py	2022/1/17 14:25	Python File	84 KB	
dis.py	2022/1/17 14:25	Python File	21 KB	
doctest.py	2022/1/17 14:25	Python File	106 KB	
enum.py	2022/1/17 14:25	Python File	40 KB	
filecmp.py	2022/1/17 14:25	Python File	11 KB	
fileinput.py	2022/1/17 14:25	Python File	17 KB	

图 10-2

在这个标准库以外还存在成千上万且不断增加的其他组件(从单独的程序、模块、软件包直到完整的应用开发框架),访问 <https://pypi.org/> 即可获取第三方库。

下面列举一些常用的标准库。

1. os

本模块提供了一种使用与操作系统相关的功能的便捷式途径。

(1) 获取当前路径。

```
>>> os.getcwd()
'D:\test'
```

(2) 返回指定目录下的所有文件名和目录名。

```
>>> os.listdir('d:\')
[' $ RECYCLE. BIN', 'BaiduNetdiskDownload', 'demo1.py', 'System Volume
Information', 'test', 'tools']
```

(3) 创建目录。

```
>>> os.makedirs('img')
os.makedirs('img\rrr\ttt')
```

(4) 删除目录。

```
os.rmdir('img') // 目录非空, 无法删除
```

(5) 删除文件。

```
os.remove('00_ok.txt')
```

(6) 文件重命名。

```
os.rename('data.txt', 'data1.txt')
```

(7) 判断文件或目录是否存在。

```
os.path.exists('ips.txtyyyy')
```

(8) 执行系统命令。

```
>>> os.system("whoami")
desktop-l5pkmal\test
```

2. time

time 模块中时间表现的格式主要有以下三种。

(1) timestamp 时间戳, 时间戳表示的是从 1970 年 1 月 1 日 00:00:00 开始按秒计算的偏移量。

- (2)struct_time 时间元组,共有九个元素组。
 - (3)format string 格式化时间,已格式化的结构使时间更具可读性,包括自定义格式和固定格式。
- 三种时间格式之间可以相互转换,转换关系如图 10-3 所示。

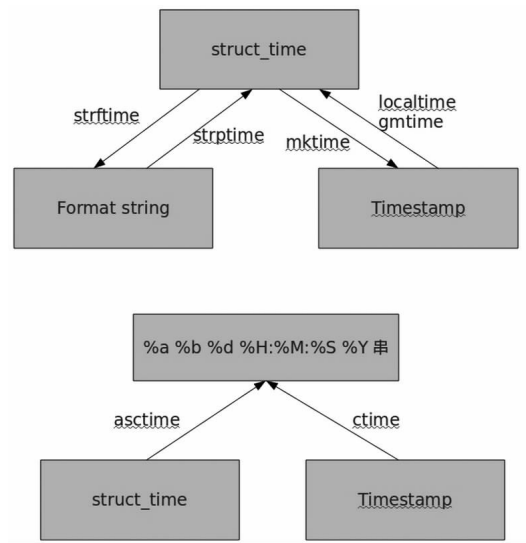


图 10-3

format time 结构化表示如表 10-1 所示。

表 10-1

格 式	含 义
%a	本地(locale)简化星期名称
%A	本地完整星期名称
%b	本地简化月份名称
%B	本地完整月份名称
%c	本地相应的日期和时间表示
%d	一个月中的第几天(01~31)
%H	一天中的第几个小时(24 小时制,00~23)
%I	一天中的第几个小时(12 小时制,01~12)
%j	一年中的第几天(001~366)
%m	月份(01~12)
%M	分钟数(00~59)
%p	本地 am 或 pm 的相应符号
%S	秒(01~60)
%U	一年中的星期数。(00~53,星期天是一个星期的开始)第一个星期天之前的所有天数都放在第 0 周

续表

格 式	含 义
%w	一个星期中的第几天(0~6,0 表示星期天)
%W	和%U 基本相同,不同的是,%W 以星期一为一个星期的开始
%x	本地相应日期
%X	本地相应时间
%y	去掉世纪的年份(00~99)
%Y	完整的年份
%Z	时区的名字(若不存在,则为空字符)
%%	'%' 字符

time.struct_time 包含的属性如表 10-2 所示。

表 10-2

下标/索引	属性名称	描 述
0	tm_year	年份,如 2017
1	tm_mon	月份,取值范围为[1, 12]
2	tm_mday	一个月中的第几天,取值范围为[1~31]
3	tm_hour	小时,取值范围为[0~23]
4	tm_min	分钟,取值范围为[0, 59]
5	tm_sec	秒,取值范围为[0, 59]
6	tm_wday	一个星期中的第几天,取值范围为[0~6],0 表示星期一
7	tm_yday	一年中的第几天,取值范围为[1, 366]
8	tm_isdst	是否为夏时制,可取值为:0, 1 或 -1

下面看几个例子。

(1)取当前时间的 timestamp。

```
>>> time.time()
1648658453.6449816
```

(2)取当前时间的 struct_time。

```
>>> time.localtime() //本地时间
time.struct_time(tm_year = 2022, tm_mon = 3, tm_mday = 31, tm_hour = 0, tm_min = 41, tm_sec = 45, tm_wday = 3, tm_yday = 90, tm_isdst = 0)
>>> time.gmtime() //格林尼治时间
time.struct_time(tm_year = 2022, tm_mon = 3, tm_mday = 30, tm_hour = 16, tm_min = 41, tm_sec = 51, tm_wday = 2, tm_yday = 89, tm_isdst = 0)
```


(3)取当前时间的 format time(年-月-日 时:分:秒格式)。

```
>>> time.strftime("%Y-%m-%d %X")
'2022-03-31 00:44:01'
```

(4)timestamp 转 struct_time。

```
>>> time.localtime(1648658453.6449816)
time.struct_time(tm_year=2022, tm_mon=3, tm_mday=31, tm_hour=0, tm_min=40, tm_sec=53, tm_wday=3, tm_yday=90, tm_isdst=0)
>>> time.gmtime(1648658453.6449816)
time.struct_time(tm_year=2022, tm_mon=3, tm_mday=30, tm_hour=16, tm_min=40, tm_sec=53, tm_wday=2, tm_yday=89, tm_isdst=0)
```

(5)struct_time 转 timestamp。

```
>>> t = time.localtime(1648658453.6449816)
>>> time.mktime(t)
1648658453.0
```

(6)format time 转 struct_time。

```
>>> time.strptime("2022-03-31 00:44:01", "%Y-%m-%d %X")
time.struct_time(tm_year=2022, tm_mon=3, tm_mday=31, tm_hour=0, tm_min=44, tm_sec=1, tm_wday=3, tm_yday=90, tm_isdst=-1)
```

(7)struct_time 转 format time。

```
>>> t = time.strptime("2022-03-31 00:44:01", "%Y-%m-%d %X")
>>> time.strftime("%Y-%m-%d %X", t)
'2022-03-31 00:44:01'
```

3. sys

sys 模块提供了一系列有关 Python 运行环境的变量和函数。

(1)sys.argv。可以用 sys.argv 获取当前正在执行的命令行参数的参数列表(list)。sys.argv[0]表示当前程序名,sys.argv[1]表示第一个参数,sys.argv[n]表示第 n 个参数。例如:

```
test.py 文件内容:
import sys
print(sys.argv)
```

执行 test.py

```
D:\test>python test.py arg1 arg2 arg3
['test.py', 'arg1', 'arg2', 'arg3']
```

(2) `sys.path`。搜索模块的路径,前文已做过详细介绍,这里不再赘述。

(3) `sys.platform`。获取当前执行环境的平台,如 `win32` 表示是 Windows 系统,`linux2` 表示是 Linux 平台。

```
>>> sys.platform
'win32'
```

(4) `sys.exit(n)`。调用 `sys.exit(n)` 可以中途退出程序,当参数非 0 时,会引发一个 `SystemExit` 异常,从而可以在主程序中捕获该异常。

`test.py` 文件内容:

```
import sys
try:
    sys.exit(1)
except SystemExit as e:
    print(e)

D:\test>python test.py
1
```

(5) `sys.version`。获取 Python 解释程序的版本信息。

```
>>> sys.version
'3.10.2 (tags/v3.10.2:a58ebcc, Jan 17 2022, 14:12:15) [MSC v.1929 64 bit (AMD64)]'
```

(6) `sys.modules`。记录已经导入当前程序的模块,前文已做过详细介绍,这里不再赘述。

4. random

Python 中的 `random` 模块用于生成随机数。

(1) `random.random()`。用于生成一个 0~1 的随机浮点数。

```
>>> random.random()
0.5186284965325015
```

(2) `random.uniform(a,b)`。用于生成一个指定范围内的随机浮点数,两个参数中一个是上限,另一个是下限。

```
>>> random.uniform(10,20)
17.320040238007707
>>> random.uniform(20,10)
15.33872274233473
```

(3) `random.randint(a,b)`。用于生成一个指定范围内的整数。其中,参数 `a` 是下限,参数 `b` 是上限,生成的随机数 `n`: $a \leq n \leq b$ 。

```
>>> random.randint(10,20)
17
```

(4) `random.randrange([start], stop[, step])`。从指定范围内按指定基数递增的集合中获取一个随机数。例如,`random.randrange(10,30,2)`,结果相当于从`[10,12,14,16,...,26,28]`序列中获取一个随机数。

```
>>> random.randrange(10,30,2)
22
```

(5) `random.choice(sequence)`。`random.choice` 从序列中获取一个随机元素。`sequence` 在 Python 中不是一种特定的类型,而是泛指一系列的类型,如 `list`、`tuple`、字符串都属于 `sequence`。

```
>>> random.choice([1,2,3,4,5])
2
```

(6) `random.shuffle(x[, random])`。用于将一个列表中的元素打乱,即将 `x` 列表内的元素随机排列。

```
>>> x = [1,2,3,4,5]
>>> random.shuffle(x)
>>> x
[2,4,5,3,1]
```

(7) `random.sample(sequence, k)`。从 `sequence` 中随机获取 `k` 个元素,作为一个片段返回。注意:`sample` 函数不会修改原有序列。

```
>>> random.sample([1,2,3,4,5],4)
[4,5,3,1]
```

这里只对上面常用的几个标准库做了简单的讲解,更详细的使用和其他标准库的使用大家可以参考官方文档:<https://docs.python.org/zh-cn/3/library/index.html>。

10.3.2 使用 pip 管理第三方库

Python 有内置的标准库,在安装完 Python 后,这些标准库都已默认安装好,但是还有很多有用的 Python 库不是内置安装的,人们称这些库为第三方库,并通过 pip 来安装和管理第三方库,以及管理库之间的依赖(dependency)关系。pip 是 Python 的第三方库管理器,可以很方便地安装、卸载与升级、查看 Python 第三方库。

不是所有的 Python 第三方库都能通过 pip 来安装,只有发布在 <https://pypi.org> 上面的才能通过 pip 安装。pypi 是一个仓库,上面存放了大量的 Python 第三方库,由 Python 官方社区维护。只要遵守 pypi 规则,所有开发者都可以把自己开发的 Python 库发布在 pypi 上,供其他人下载使用。

通过执行 pip help 可以查看 pip 的功能和参数。

```
C:\Users\test>pip help

Usage:
  pip <command> [options]

Commands:
  install           Install packages.
  download          Download packages.
  uninstall         Uninstall packages.
  freeze            Output installed packages in requirements format.
  list              List installed packages.
  show              Show information about installed packages.
  check             Verify installed packages have compatible dependencies.
  config            Manage local and global configuration.
  search            Search PyPI for packages.
  cache             Inspect and manage pip's wheel cache.
  index             Inspect information available from package indexes.
  wheel             Build wheels from your requirements.
  hash              Compute hashes of package archives.
  completion        A helper command used for command completion.
  debug             Show information useful for debugging.
  help              Show help for commands.

General Options:
  ...
```

下面进行 pip 的安装和配置并对常用的功能进行讲解。

1. pip 的安装与配置

Python 2.7.9+ 或 Python 3.4+ 以上版本都自带 pip 工具,不需要自己额外安装。用户可以通过以下命令来判断是否已安装。

```
C:\Users\test>pip --version
pip 21.2.4 from C:\Program Files\Python310\lib\site-packages\pip (python 3.10)
```

从上面的输出可以看出,pip 本身也是一个第三方库,为什么在 Windows 系统上能够不在前面加 python 而直接运行呢? 这是因为 pip 包被封装成了可执行程序,这个可执行程序存放在 Python 安装目录的 scripts 文件夹中,而这个 scripts 文件夹又在环境变量的 path 中,因此可以直接运行。注意,图 10-4 中的三个文件其实是完全相同的,pip3 和 pip3.10 是 pip 的副本,这样做的好处是方便在同一台计算机上安装多个 Python 版本,使用 pip 时容易区分。

C:\Program Files\Python310\Scripts				
名称	修改日期	类型	大小	
pip.exe	2022/3/9 17:16	应用程序	104 KB	
pip3.10.exe	2022/3/9 17:16	应用程序	104 KB	
pip3.exe	2022/3/9 17:16	应用程序	104 KB	

图 10-4

```
C:\Users\test>set path
Path = C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\Program Files\Python310\Scripts\;C:\Program Files\Python310\;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\System32\OpenSSH\;C:\Users\test\AppData\Local\Microsoft\WindowsApps;
```

因为 pip 是一个 Python 库,所以除了直接运行封装好的 pip 之外,还可以用下面的方法运行 pip。

```
C:\Users\test>python -m pip --version
pip 21.2.4 from C:\Program Files\Python310\lib\site-packages\pip (python 3.10)
```

这里 pip 其实执行的就是 python -m pip。

如果使用的 Python 版本没有自带 pip,可以通过下面的方式获取安装脚本进行安装。

- (1) 下载 <https://bootstrap.pypa.io/get-pip.py>。
- (2) 执行 python get-pip.py。

pip 官方升级比较频繁,当当前 pip 版本低于官方最新版本时,使用 pip 时会产生一个警告,提示官方发布了最新版,要求升级。

```
C:\Users\test>pip install  
WARNING: You are using pip version 21.2.4; however, version 22.0.4 is  
available.  
You should consider upgrading via the 'C:\Program Files\Python310\python.  
exe -m pip install --upgrade pip' command.
```

根据官方提示升级 pip。

```
C:\Users\test>pip install --upgrade pip  
Defaulting to user installation because normal site-packages is not writeable  
Requirement already satisfied: pip in c:\program files\python310\lib\site-  
packages (21.2.4)  
Collecting pip  
Using cached pip-22.0.4-py3-none-any.whl (2.1 MB)  
Installing collected packages: pip  
WARNING: The scripts pip.exe, pip3.10.exe and pip3.exe are installed in 'C:  
\Users\test\AppData\Roaming\Python\Python310\Scripts' which is not on PATH.  
Consider adding this directory to PATH or, if you prefer to suppress this  
warning, use --no-warn-script-location.  
Successfully installed pip-22.0.4  
WARNING: You are using pip version 21.2.4; however, version 22.0.4 is  
available.  
You should consider upgrading via the 'C:\Program Files\Python310\python.  
exe -m pip install --upgrade pip' command.
```

如果 Python 安装在 C:\Program Files\Python310\ 中,而运行 Python 的用户为 test,对 Python 安装目录没有写入权限,安装或升级第三方库时,无法写入系统的第三方库目录(Python 安装目录下的 Lib\site-packages 文件夹)中,因此,Python 提示升级或安装的第三方库将被安装到用户的第三方库目录(C:\Users\test\AppData\Roaming\Python\Python310\site-packages)中。

```
Defaulting to user installation because normal site-packages is not writeable
```

由于 pip 升级到了用户的第三方库目录中,那么对应的 pip.exe 等三个文件也将会进行升级,同样也会升级到用户的 scripts 文件夹(C:\Users\test\AppData\Roaming\Python\Python310\Scripts)中,由于用户的 scripts 文件夹未在环境变量的 path 中,系统给出如下提示信息。

```
WARNING: The scripts pip.exe, pip3.10.exe and pip3.exe are installed in 'C:\Users\test\AppData\Roaming\Python\Python310\Scripts' which is not on PATH.  
Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
```

这种方式升级之后的结果是 C:\Users\test\AppData\Roaming\Python\Python310\site-packages 和 C:\Program Files\Python310\Lib\site-packages 中存在着不同版本的 pip, 那么 Python 在运行 pip 时会提醒用户用的是哪里的 pip。

这是 pip 升级前, 运行 pip 时的提示。

```
Requirement already satisfied: pip in c:\program files\python310\lib\site-packages (21.2.4)
```

这是 pip 升级后, 运行 pip 时的提示。

```
Requirement already satisfied: pip in c:\users\test\appdata\roaming\python\python310\site-packages (22.0.4)
```

可以发现, 升级前使用的是系统第三方库中的 pip, 升级后使用的是用户的第三方库中的 pip, 通过前面 Python 模块导入顺序的学习, 我们知道库的搜索顺序是由 sys.path 决定的, 再来看 sys.path 中的内容。

```
>>> sys.path  
['', 'd:\test', 'C:\Program Files\Python310\python310.zip', 'C:\Program Files\Python310\DLLs', 'C:\Program Files\Python310\lib', 'C:\Program Files\Python310', 'C:\Users\test\AppData\Roaming\Python\Python310\site-packages', 'C:\Program Files\Python310\lib\site-packages']
```

很明显, 用户的第三方库目录是优先于系统的第三方库的, 因此, 当用户的第三方库中有 pip 库时, 将优先调用用户第三方库中的库。这里可以做个实验, 通过 PYTHONPATH 将系统的第三方库排在用户库的前面, 看一下实验效果, 这里就不做演示了。

前面提到了升级 pip 时, pip.exe 等三个文件也将升级, 但是升级到了用户的 scripts 目录中, 而用户的 scripts 目录又没在 path 环境变量中, 因此, 后面运行 pip 时其实还是在使用系统目录中的 pip.exe, 但是系统目录中的 pip.exe 又没有被升级, 那么为什么运行 pip.exe 还是显示升级之后的版本呢? 前面还提到了 pip.exe 其实是对 python -m pip 命令的一个封装, 这也从侧面印证了我们的猜想, pip 的本质就是执行 python -m pip, 由于 sys.path 的原因, 当然是优先调用升级后的 pip。

下面切换一个用户再来执行 pip。

```
C:\Windows\system32>runas /user:demo cmd  
输入 demo 的密码:  
试图将 cmd 作为用户 "DESKTOP-L5PKMAL\demo" 启动...
```

```
C:\Windows\system32>pip --version
pip 21.2.4 from C:\Program Files\Python310\lib\site-packages\pip (python 3.10)
```

可以发现, pip 又变成升级之前的版本了。

2. 搜索第三方库

通过查看 pip 帮助文档,发现可以使用 pip search 来搜索官方的软件仓库,但是由于 pip search 服务的请求量激增,而开发新的搜索方式需要投入大量的人力和物力,所以官方永久禁用 XMLRPC 搜索功能。

下面介绍另外两种搜索 pypi 第三方库应用市场的方法。

(1)通过官网搜索。访问 pypi 的官方网站:https://pypi.org/(见图 10-5),在搜索框中输入要搜索的第三方库信息,如 pip(见图 10-6)。

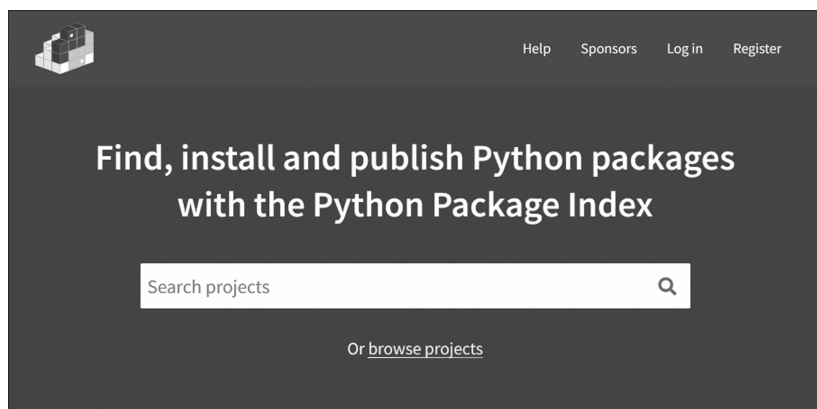


图 10-5

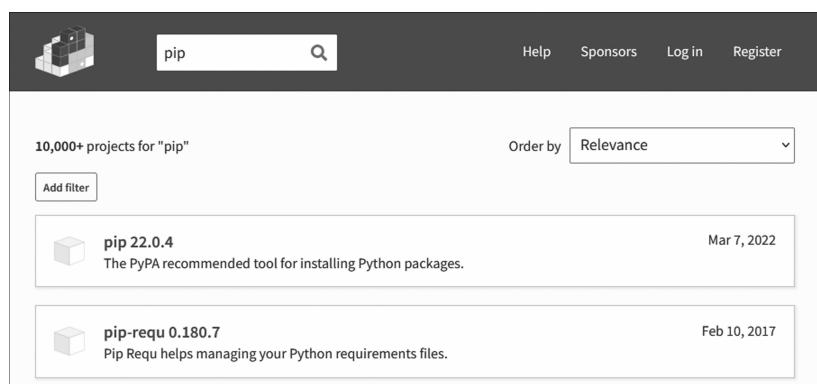


图 10-6

(2)通过第三方库 pip-search 来搜索。通过安装第三方搜索库(如 pip_search)来搜索 pypi,其本质就是一个网络爬虫,爬取 pypi 页面上的数据,安装和使用都非常简单。


```
C:\Users\test>pip install pip-search
C:\Users\test>C:\Users\test\AppData\Roaming\Python\Python310\Scripts\
pip_search.exe pip
```

从上面的用法可以看出, `pip_search` 作为第三方库, 当使用非管理员权限进行安装时, 将会被安装到用户的第三方库目录下, 而用户的 `scripts` 目录又没在环境变量 `path` 中, 因此, 需要补全 `pip_search.exe` 的路径才能使用。为了方便起见, 可以将 `C:\Users\test>C:\Users\test\AppData\Roaming\Python\Python310\Scripts\` 目录添加到用户环境变量的 `path` 中。

3. 列出第三方库

列出已安装的第三方库。

```
C:\Users\test>pip list
```

Package	Version
beautifulsoup4	12.10.0
bs4	0.0.1
certifi	2021.10.8
charset-normalizer	2.0.12
commonmark	0.9.1
idna	3.3
pip	22.0.4
pip-search	0.0.11
Pygments	2.11.2
requests	2.27.1
rich	12.0.1
scapy	2.4.5
setuptools	58.1.0
soupsieve	2.3.1
urllib3	1.26.8

列出可升级的第三方库。

```
C:\Users\test>pip list -o
```

Package	Version	Latest	Type
setuptools	58.1.0	61.2.0	wheel
urllib3	1.26.8	1.26.9	wheel

4. 安装与升级第三方库

在“2. 搜索第三方库”我们已经简单学习了使用 `pip install ×××` 命令安装第三方库,下面介绍几个在安装第三方库过程中常用的参数。

(1)使用 pypi 国内镜像站进行加速。例如,使用清华大学的 pypi 镜像网站安装 requests。

```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple requests
```

这里需要注意,如果镜像站为 http 的,需要增加 `--trusted-host` 参数,如使用豆瓣的镜像时:

```
pip install -i http://pypi.douban.com/simple --trusted-host pypi.douban.com xxx
```

如果想将 `https://pypi.tuna.tsinghua.edu.cn/simple` 设为默认的下载站点,可以使用以下方法。

```
D:\> pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
Writing to C:\Users\test\AppData\Roaming\pip\pip.ini
```

(2)通过使用 `==`、`>=`、`<=`、`>` 和 `<` 来指定一个第三方库的版本号。

```
pip install requests==2.13.0           //安装 2.13.0 版本的 requests
pip install requests>=2.13.0           //requests 的版本要大于等于 2.13.0
```

(3)批量安装第三方库,使用 `-r` 参数加软件列表文件。

例如,要批量安装 requests 2.18.0 和 numpy,需要先编写第三方库列表,一行一个。

```
D:\test>type a.txt
requests==2.18.0
numpy
D:\test>pip install -r a.txt
```

(4)升级第三方库,使用 `--upgrade` 参数对第三方库进行升级。

“1. pip 的安装与配置”已经演示了如何将 pip 升级到最新版,这里不再赘述。

5. 卸载第三方库

可以通过 `pip uninstall` 卸载一个第三方库,如要卸载 requests,命令如下。

```
D:\test>pip uninstall requests
```

10.3.3 使用虚拟环境解决第三方库冲突

Python 应用程序经常会使用一些不属于标准库的包和模块。应用程序有时需要某个特定版本的库,因为它需要一个特定的 bug 已得到修复的库或者它是使用了一个过时版本的库的接口编写的。这就意味着可能无法安装一个 Python 来满足每个应用程序的要求。如果应用程序 A 需要一个特定模块的 1.0 版本,但是应用程序 B 需要该模块的 2.0 版本,这两个应用程序的要求是冲突的,安装版本 1.0 或版本 2.0 将会导致其中一个应用程序不能运行。

这个问题的解决方案就是创建一个虚拟环境(通常称为 virtualenv),包含一个特定版本的 Python,以及一些附加的包的独立的目录树。

不同的应用程序可以使用不同的虚拟环境。为了解决前面例子中的冲突,应用程序 A 可以有自己的虚拟环境,其中安装了特定模块的 1.0 版本。而应用程序 B 拥有另外一个安装了特定模块 2.0 版本的虚拟环境。如果应用程序 B 需要一个库升级到 3.0,也不会影响到应用程序 A 的环境。

1. 虚拟环境的原理

虚拟环境的原理非常简单,其实就是应用了“10.1 模块”中“10.1.3 模块的导入顺序”中讲到的原理,通过设置 sys.path 调整第三方库的搜索路径来实现的。简单来说,就是每个虚拟环境都有自己的第三方库(site-packages)和第三方脚本(scripts)存储路径,并且可以选择是否继承 Python 系统级别的第三方库和第三方脚本。每个虚拟环境的第三方库和第三方脚本是相互隔离的。从这个意义上说,回想“10.3.2 使用 pip 管理第三方库”的“1. pip 的安装与配置”中讲到的用户级别的第三方库,思考是不是每个非管理员级别的系统用户也是一个虚拟环境。

2. 在命令行环境中使用虚拟环境

(1)创建虚拟环境。在 Python 3.3 之前,只能通过 virtualenv 创建虚拟环境,Python 3.3 之后,可以用模块 venv 代替 virtualenv 工具,这样做的好处是不用单独安装。Python 3.3 及之后的版本,已经内置了 venv 库,virtualenv 和 venv 用法相似,后面以 venv 为例创建虚拟环境。

下面创建一个名为 testvenv 的虚拟环境。

```
python -m venv testvenv
```

利用这种方法创建的虚拟环境不会继承系统级别的第三方库,如果想要继承系统级别的第三方库,需要在创建时增加--system-site-packages 参数。

```
python -m venv --system-site-packages testvenv-sys
```

如果不想使用独立的 pip,还可以增加--without-pip 参数。

```
python -m venv --without-pip testvenv-pip
```

这时要注意,由于没有将 pip 引入虚拟环境,因此在虚拟环境中使用 pip 时其实使用的是系统级别的 pip, pip 会将第三方库安装到系统的第三方库或用户的第三方库中,起不到隔离环境的作用,因此这个参数一般不会使用。

(2) 激活使用虚拟环境。虚拟环境创建好后,将在虚拟环境\scripts 目录中创建激活与退出脚本,在 Windows 系统中,可以运行 activate. bat 来激活虚拟环境。

```
D:\test>testvenv\Scripts\activate.bat
(testvenv) D:\test>
```

进入虚拟环境后,观察环境变量 path 和 Python 中的 sys. path。

```
(testvenv) D:\test>set path
Path=D:\test\testvenv\Scripts;C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\Program Files\Python310\Scripts\;C:\Program Files\Python310\;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\System32\OpenSSH\;C:\Users\test\AppData\Local\Microsoft\WindowsApps;

>>> sys.path
['', 'C:\Program Files\Python310\python310.zip', 'C:\Program Files\Python310\DLLs', 'C:\Program Files\Python310\lib', 'C:\Program Files\Python310', 'D:\test\testvenv', 'D:\test\testvenv\lib\site-packages']
```

(3) 退出虚拟环境。在虚拟环境\scripts 目录下有一个退出虚拟环境的脚本 deactivate. bat,运行该脚本后即可退出虚拟环境。

```
(testvenv) D:\test>testvenv\Scripts\deactivate.bat
```

3. 在 PyCharm 中使用虚拟环境

本章使用的是 PyCharm Community Edition 2021. 3. 3 版本,其他版本的 PyCharm 与本版设置虚拟环境的原理相同。打开 PyCharm 后单击 New Project 按钮,新建一个项目,打开新建项目界面。虚拟环境的建立过程如图 10-7~图 10-9 所示。

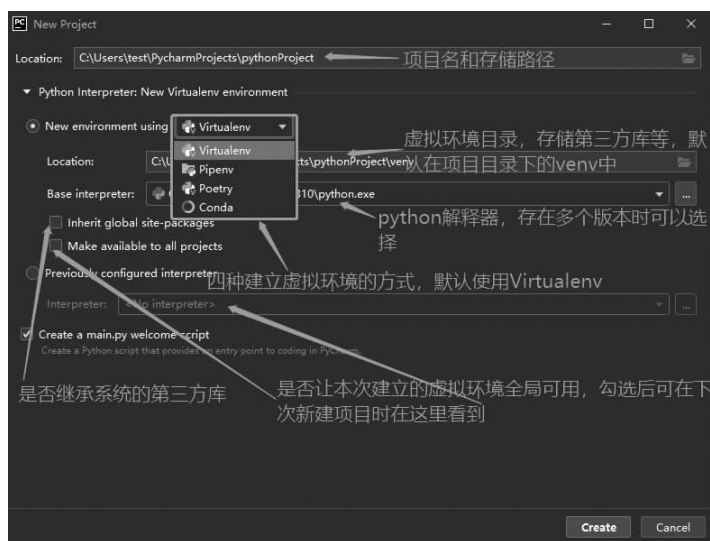


图 10-7

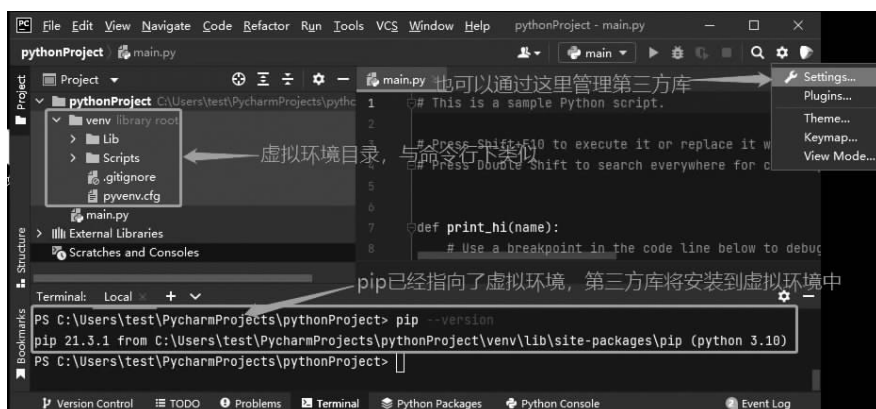


图 10-8

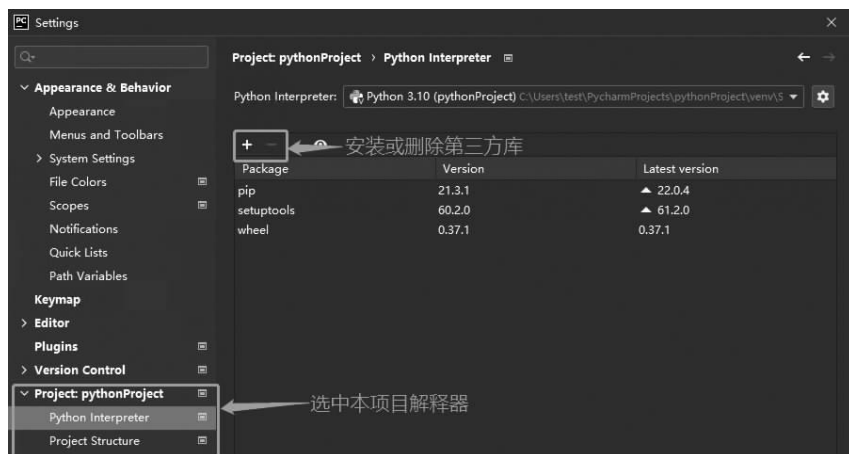


图 10-9



本章总结

• 库(lib)是模块(module)和包(package)的统称,包中包含模块,模块可以不在包中,包是一个包含了__init__.py 文件的文件夹,模块是一个.py 文件。通常在口语表述中不对三者进行区分。

• 所有库都需要使用 import 导入后才能使用。

• sys.path 决定了库的导入顺序,sys.path 由当前运行目录、PYTHONPATH、安装默认配置和.pth 文件决定,.pth 只能放在第三方库的目录或 Python 的安装目录中,也可以自定义路径。

• 由于系统权限问题,当系统的第三方库目录没有写入权限时,将会为当前用户创建自己的第三方库目录,只对当前用户有效。系统的第三方库目录位于 Python 安装目录\Lib\site-packages 下。

• 标准库位于 Python 安装目录\Lib 下,也有部分 C 语言开发的内置标准库内置在 Python 相关文件中。

• 可以使用 pip 进行第三方库的管理。

• 虚拟环境的本质是环境变量。

• 虚拟环境主要用来解决第三方库冲突的问题。



术语

术语名称	术语含义
模块(module)	此对象是 Python 代码的一种组织单位。各模块具有独立的命名空间,可包含任意 Python 对象。模块可通过 importing 操作被加载到 Python 中
包(package)	一种可包含子模块或递归地包含子包的 Python module。从技术上说,包是带有 __path__ 属性的 Python 模块
虚拟环境(virtual environment)	一种采用协作式隔离的运行环境,允许 Python 用户和应用程序在安装及升级 Python 分发包时不会干扰到同一系统上运行的其他 Python 应用程序的行为
pip	pip 是 package installer for Python 的缩写,即 Python 的包安装器,用户可以从 Python Package Index(https://pypi.org/)安装第三方库



程序阅读

扫描二维码阅读程序,写出每行程序执行的逻辑。





综合案例

将文件名改为“文件名-创建时间戳”格式。例如,文件的文件名为 test.jpg,创建时间为 2022 年 5 月 30 日 19:30:30,则将 test.jpg 重命名为 test-2022 年 5 月 30 日 19-30-30.jpg。



练习

1. 通过修改环境变量、Python 程序名和 pip 程序名实现 Python 3.10 和 Python 2.7 共存。
2. 尝试将 Python 安装到非系统盘下,观察使用 pip 进行第三方库安装时装在哪里。
3. 在命令行下创建一个虚拟环境。



自我反思与评价

学习到了什么?

学习中遇到了什么问题? 是如何解决的? (可以列出查阅资料的链接或书籍,解决问题的途径)