第17章. 软件分发

当你阅读到本章时，应该已经能够将自己的想法转化为相应的Python脚本了。然而自己编写软件自己使用，就好像玩一款单机游戏。而现实中的Python世界更接近于一款超大型网络游戏——有大量用户参与其中，可以使用别人编写的软件，也可以分享自己编写的软件。在工业界和学术界，软件的交换被称为“分发（distribution）”，而实现分发的基础是能够对软件进行“打包（packaging）”。Python官方手册对这些技术描述得较简略，但它们是Python用户必须掌握的，因此本书在最后用一整章来讨论这些技术。

17-1. 软件分发概述

“技术社区（technology communities）”是一个模糊的概念。最广义地讲，只要你使用了用某种技术开发出的产品，就已经与该技术产生了联系，因此也就被动地成为了相关技术社区的一份子。但一般而言，我们提到一个技术社区时，指的是使用该技术的开发者形成的群体。对技术社区最狭义的解释是它的在线平台（官方网站、论坛、博客等等）。

技术社区的核心目标是交流技术。对于Python社区来说，分发软件也自然成为了它的基本职能的一部分。需要强调的是，软件分发并不如人们直觉中那样简单。穆罕默德·哈希米（Mahmoud Hashemi）针对Python的软件分发写了一篇非常好的文章，将软件打包分为了多个层次，形成了一个金字塔[1]。图17-1中的金字塔对穆罕默德的金字塔进行了少许修改，将被本书作为Python软件分发的模型。下面按照从顶部到底部的顺序依次介绍这些软件打包技术。

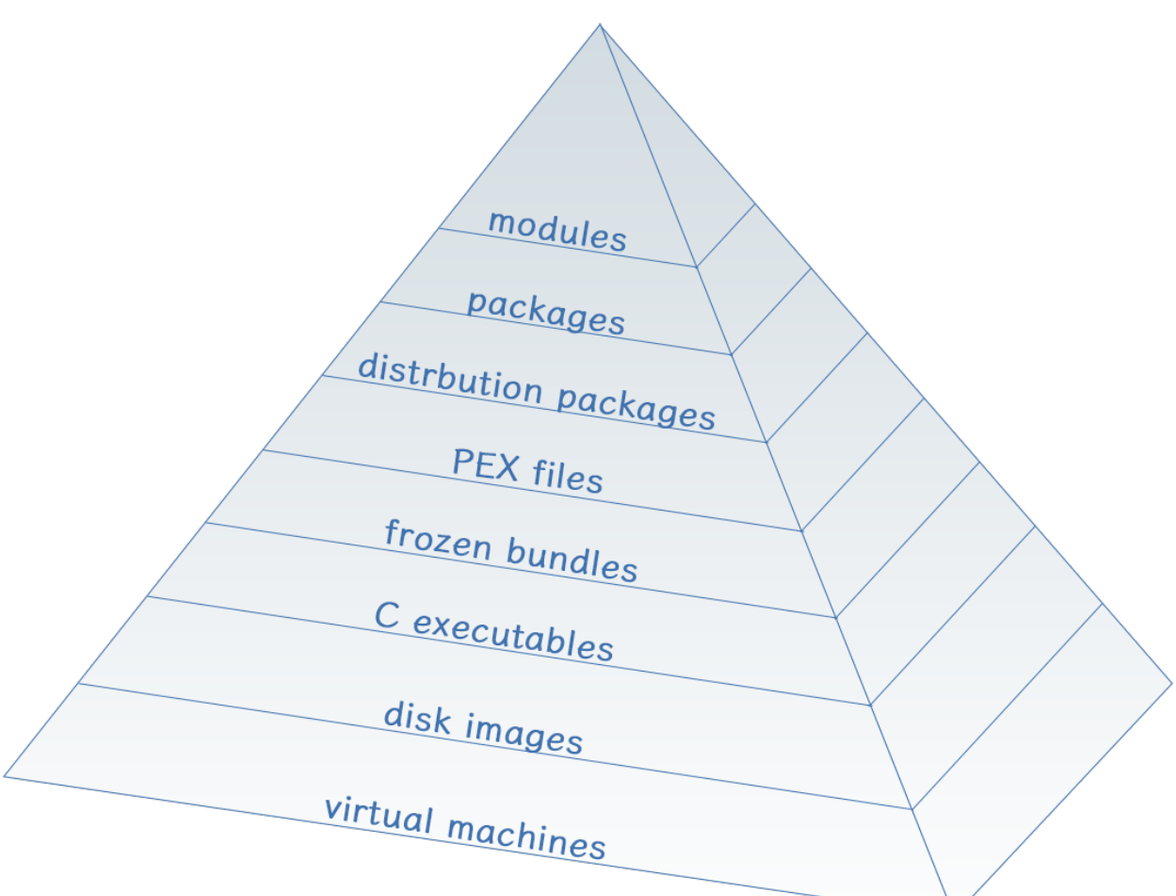

图17-1. 软件分发金字塔

➤ 当一个Python脚本仅导入了Python标准库提供的模块时，就能够以单独一个模块的形式被分发，也就是说直接将该Python脚本拷贝到另一台计算机上。为了保证该脚本能够被使用，该计算机必须安装有某种Python解释器，且版本必须满足该脚本的要求。此外，如果该脚本使用了标准库中与操作系统相关的模块（例如posix和winreg），那么还会对该计算机的操作系统类型有限制。注意该脚本既能够以.py文件的形式存在，也能够以.pyc文件的形式存在，后者被称为无源文件发行版，这在第6章已经提到了。

➤ 当多个Python脚本相互导入时，需要用第6章介绍的技巧将它们组织成一个包，然后以包为单位进行分发。这与以模块为单位进行分发没有本质区别，只不过从拷贝文件变成了拷贝目录。但除了.py文件和.pyc文件之外，包还可以包含.so文件，.dll文件和.pyd文件，以及仅包含.py文件和.pyc文件的ZIP归档文件。这意味着包还可以携带扩展模块。理论上扩展模块也可以直接分发，但通常会提供扩展模块的伴随模块，并以包的形式分发它们。

➤ 对于简单的Python程序（例如本书中的所有例子），以模块或包为单位进行分发已经足够了。然而随着Python程序复杂程度的增加，包的规模也越来越大。为了控制Python程序的规模，我们需要将复杂程序按照功能划分为很多部分，每个部分作为一个独立的单元被分发。这种单元被称为“分发包（distribution packages）”。

引入了分发包之后，一个完整的Python程序被视为由多个分发包构成，而具有相同功能的多个程序可以共享实现该功能的分发包，以有效减少这些程序在计算机上占用的总存储空间。然而这种节省是存在代价的：分发包之间存在依赖关系，而满足这种依赖关系是一件相当困难的事情，被称为“依赖地狱（dependency hell）”。基于分发包的发布系统必须提供一组专门的工具来解决依赖地狱问题，使得软件分发不再是简单的拷贝。

分发包与包的区别就在于，分发包除了包含实现模块或扩展模块的文件之外，还必须额外包含提供元数据的文件，这些元数据最重要的功能是说明该分发包与其他分发包之间的依赖关系。如果一个分发包只能包含.py文件和源代码文件（例如.c文件），则被称为“源代码分发包（source distribution packages）”；否则被称为“二进制分发包（binary distribution pacakges）”。源代码分发包和二进制分发包的元数据格式是不同的。

在基于分发包的发布系统中，一个源代码分发包和若干二进制分发包合在一起构成了一个逻辑上的分发包。当涉及扩展模块时，源代码分发包和二进制分发包的区别是明显的：前者包含实现扩展模块的源代码文件（例如.c文件），后者包含扩展模块本身（.so文件，.dll文件和.pyd文件）。在这种情况下二进制分发包是不能跨平台的。但如果不涉及扩展模块，则源代码分发包和二进制分发包的区别仅在于元数据的格式。此时的二进制分发包是跨平台的。源代码分发包总是跨平台的，但如果涉及扩展模块，在安装时就需要重新编译，速度较慢，且可能存在“安装时要求（install requires）”（例如C编译器）。而二进制分发包安装时只需要简单的拷贝，速度较快，且不存在安装时要求。由于两种分发包的优点和缺点正好互补，所以通常捆绑出现。

上述三种软件分发方式都是针对开发者的，不论模块、包还是分发包，都应被视为软件的组件，类似于发动机和轴承之于汽车，芯片和线路板之于计算机。由于开发者具有更多关于Python的知识（例如通过命令行启动Python解释器来执行一个模块），所以上述分发方式最终提供的是模块和/或扩展模块本身。从2003年开始，Python社区就启动了“Python包索引（Python Package Index，PyPI）”项目。该项目的核心是一个面向公众的在线分发包仓库，其Web接口是https://pypi.org，但使用专门工具（例如pip和twine）访问更加方便。从2011年起，PyPI由“Python打包权威机构（Python Packaging Authority，PyPA）”[2]负责管理和维护，并成为了Python社区的默认分发包仓库。

然而有许多Python程序的使用者对Python技术一点也不了解，只掌握了输入文件名或点击图标直接运行软件的技能。此类用户被软件业称为“最终用户（end users）”。向最终用户发布软件时使用的分发方式与向开发者发布软件时使用的分发方式是不同的。图17-1的金字塔中，从上往下的第4层到第6层是专门针对最终用户的分发方式。

➤ “Python可执行文件（Python EXecutables，PEX）”本质上是一个ZIP归档文件，其内除了实现模块和/或扩展模块的文件外，还包含提供元数据的文件，而这些元数据包括如何执行PEX文件的信息，使得PEX文件可以被直接执行。但PEX文件的执行依赖于预先安装好的Python解释器（它自动包含Python标准库），而目前只有Unix和类Unix操作系统预先安装了CPython（且大概率实现Python 2），Windows则没有预先安装任何Python解释器。当PEX文件不包含扩展模块时，它是跨平台的，仅依赖于Python解释器。当操作系统中没有Python解释器，或Python解释器的版本不符合要求时，执行PEX文件会报错。

➤ “冻结包袱（frozen bundles）”与PEX文件的区别在于将实现模块和/或扩展模块的文件、元数据文件和Python解释器自身打包在一起，做到了“自包含（self-contained）”。从操作系统和用户的视角来看，冻结包袱是真正的可执行文件。但由于Python解释器需要为不同平台提供不同的二进制代码，所以冻结包袱不论是否包含扩展模块都不是跨平台的。

➤ 在第1章已经提到过，用Nuitka可以将Python脚本转化为C源文件，然后再生成可执行文件。这本质上也是一种打包技术。显然，通过这种方式生成的可执行文件比冻结包袱小得多，且执行速度也要快4到5倍。

Python社区并没有提供一个权威的平台来发布PEX文件、冻结包袱和C可执行文件。它们像其他应用程序一样，通常是通过操作系统本身的应用商店来发布。然而这些应用商店有自己的格式要求，例如：

- Windows上的Microsoft Store：要求exe或msi格式。
- macOS和iOS上的APP Store：要求ipa、pxl或deb格式。
- RedHat系列Linux上的YUM系统：要求rpm格式。
- Debian系列Linux上的APT系统：要求deb格式。
- BSD系列Unix上的PKG系统：要求pkgng格式。

因此需要将PEX文件、冻结包袱或C可执行文件封装在这些格式的文件中。但通过Web发布Python应用时，可以直接让最终用户下载PEX文件、冻结包袱或C可执行文件。

其实图17-1中的金字塔应该到C可执行文件为止，该金字塔的最后两层严格说来并不属于软件分发方式，虽然它们确实可被用于分发软件。

- “磁盘映像（disk images）”是一种虚拟技术，用一个文件来虚拟一个外存储器分区，这样就能以分区为单位分发软件。一个磁盘映像不局限于包含一个软件，而可以包含大量软件共同形成的一个软件环境。将磁盘映像中的文件完整拷贝到某个本地目录，就相当于使该目录重现了磁盘映像封装的软件环境。磁盘映像的一个典型例子是代表光盘的ISO文件。

- “虚拟机（virtual machine）”也是一种虚拟技术，与磁盘映像的本质区别在于它不仅仅虚拟一个外存储器分区，而是虚拟整个计算机，包括硬件和操作系统。虚拟机又分为两类：第一类虚拟机直接运行在硬件之上，而第二类虚拟机需要运行在宿主操作系统之上。显然，运行虚拟机所需的资源要比打开磁盘映像所需的资源多得多。

磁盘映像和虚拟机都不是专门为分发软件而设计的，但都可以被应用于分发软件。通过磁盘映像来分发软件的最早例子是macOS中的DMG文件，即“Apple磁盘映像（Apple Disk Image）”。与ISO文件需要专门的软件来处理不同，DMG文件可以直接双击打开，成为一个只读的临时目录，使用标准的文件系统访问技巧就可以完成DMG文件内容的拷贝。

在Apple磁盘映像技术的启发下，RedHat系列Linux引入了Flatpak，Debian系列Linux引入了Snappy，它们都是基于磁盘映像的软件分发系统，但局限于特定的操作系统系列。而AppImage是一种类似的技术，但设计目标是支持所有Linux发行版。

目前最成功的跨平台软件分发技术是Docker，它能够真正做到完美跨平台，即一台计算机只要能安装Docker Engine就可以使用，在硬件和操作系统上没有任何限制。Docker结合了磁盘映像和虚拟机，其磁盘映像被称为“Docker容器（Docker containers）”，而Docker Engine则可被视为一种轻量级虚拟机。

至此我们已经介绍完软件分发金字塔中的每一层。在本节的最后，有必要介绍基于conda的软件生态系统。第1章已经提到了，Python的一个重要应用领域是人工智能，但虽然它是人工智能开发者的重要工具，却不是唯一的重要工具。举例来说，PyTorch和OpenCV虽然提供了Python接口，可以被当成扩展模块使用，但其本身是标准的C/C++项目，还提供了针对C和C++的API。而PyPI只负责维护标准的Python项目。另外，虽然numpy和基于它的其他模块都是标准的Python项目，但想让它们发挥最大功效，必须针硬件（例如GPU）和底层库（例如Cuda C库）进行编译，这意味着需要配置复杂的编译选项。而由于PyPI追求的是跨平台通用性，因此不具备对编译过程进行精确控制的能力。

为了解决这一问题，人工智能开发者们于2012年成立了Anaconda公司（事实上该公司最初的名称是“Continuum Analytics”，而“Anaconda”是其核心产品的名称）[3]。该公司的职能与PyPA类似，收集人工智能领域的软件并将它们整理成一个仓库。但Anaconda与PyPI存在如下不同：

➤ Anaconda仓库中的项目不局限于Python项目，可以是任何编程语言的项目，只要与人工智能相关。
➤ PyPI的包管理器pip只能解决Python的依赖地狱问题，而Anaconda的包管理器conda能够解决任何编程语言的依赖地狱问题。
➤ Anaconda提供的部分服务是收费的。

简而言之，Anaconda是一个专门服务人工智能开发者的软件生态系统，由于人工智能开发者们选择Python作为自己的编程工具，所以Anaconda也就与Python产生了密切联系。但Anaconda并非一个Python项目，而conda也并非专门针对Python的包管理器。

准确地说，Anaconda是一个包含了CPython、pip、conda和大量数据处理工具的软件集合。完整安装Anaconda需要超过5GB的磁盘空间。如果你不想一次性安装所有这些软件，则可以安装Miniconda[4]，它只包含CPython、pip、conda和它们所依赖的一些基本库（例如zlib），此后可以通过conda自行安装Anaconda仓库中的其他软件。

conda并不是与Anaconda仓库绑定的。如果conda是通过安装Anaconda或Miniconda获得的，那么Anaconda仓库会被设置为conda的默认频道，但我们还可以给conda设置其他频道。一个典型的应用场景：每个Anaconda帐户都附带一个云空间，一个人工智能开发团队在Anaconda仓库中找不到合适的工具时，可以将他们自己开发的工具上传到Anaconda云，然后设置conda的一个频道为该Anaconda云。

Anaconda云通常被作为私有conda仓库使用，而conda-forge是Anaconda之外最著名的公有conda仓库，它是由开源社区维护的非赢利性项目，可以作为Anaconda的补充，也可以作为Anaconda的纯开源替代品[5]。我们可以通过将conda的一个频道设置为conda-forge来使用它。如果你希望获得一个纯开源环境，则可以安装conda-forge项目自身提供的软件集合miniforge，它与miniconda的唯一区别是以conda-forge仓库作为默认频道。

本章剩下部分将这样安排：由于模块和包已经在第6章讨论过，磁盘映像和虚拟机并非专门用来分发Python程序的技术，所以接下来将依次讨论分发包、PEX文件、冻结包袱和C可执行文件，而在最后会用单独一节简单介绍conda的使用。

17-2. 安装和下载分发包

（标准库：ensurepip）

PyPA除了维护PyPI外，还负责提供关于如何使用PyPI的文档。这包括Python官方手册中的“安装Python模块”和“分发Python模块”，然而这两个文档过于简略。PyPA把重点放在维护“Python打包用户导引（Python Packaging User Guide）”上，它相当于PyPI的官方手册[6]。本节和下节介绍的内容都来自于该导引，当你需要了解本书未提及的内容时，请自行查阅这一资料。此外，PyPA还列出了它提供的用于使用PyPI的工具，以及推荐的用于使用PyPI的第三方工具[7]。

访问PyPI的前提是安装了Python解释器和pip。如果按照Python官方手册的“Python安装和使用”中描述的方法安装了实现Python 3的CPython，那同时也会安装pip。如果你通过某款Linux发行版自带的包管理器安装了实现Python3的CPython，则可能需要再单独安装pip[8]。请通过下面的命令行验证它们已经正确安装：

```
$ python3 --version
$ python3 -m pip --version
```

一切正常的话，这会依次输出Python版本和pip版本。如果你发现成功安装了Python但没有安装pip，则可以尝试使用标准库中的ensurepip模块来安装pip，即：

```
$ python3 -m ensurepip
```

当然，如果pip已经安装，则上述命令行将不做任何操作。

pip的语法可以概括为：

```
python3 -m pip command [options]
python3 -m pip {--version|-V}
python3 -m pip {--help|-h}
```

其中必须的command代表某个pip命令，而可选的options则代表任意多个pip选项。特别的，我们也可以直接将pip当成一个shell命令使用，即省略“python3 -m”（在Windows上不省略“python3 -m”会导致一些bug）。但要注意，如果你使用的是Unix或Linux，那么等价于“python3 -m pip”的将是“pip3”。

表17-1列出了所有的pip命令，本书只讨论其中最常用的几个。表17-2列出了所有通用的pip选项，但本书并不讨论它们。每个pip命令还有仅适用于自己的选项。如果你想了解本书未覆盖的内容，请查阅pip的官方手册[9]。

表17-1. pip命令

命令	说明
list	列出所有已安装分发包。
show	显示指定已安装分发包的相关信息。
check	检查指定已安装分发包的依赖性是否被满足。
install	安装指定分发包。
uninstall	卸载指定分发包。
freeze	将已经安装的所有分发包记录到一个需求文件。
download	下载指定分发包。
wheel	按指定的要求创建轮子。
hash	计算指定分发包的哈希值。
completion	命令补全。
help	显示帮助信息。
debug	显示对调试有用的信息。
inspect	检查Python环境。
config	管理pip的全局配置文件和用户配置文件。
cache	检查pip的轮子缓存。

表17-2. pip通用选项

选项	说明
--debug	使运行过程中抛出的异常被传递到main()之外，而非写入标准出错。
--isolated	以隔离模式运行，忽略环境变量和用户配置文件。
--require-virtualenv	强制要求在虚拟环境中运行。
-v, --verbose	增加输出信息。该选项可以叠加以增强效果。至多叠加三次。
-q, --quiet	减少输出信息。该选项可以叠加以增强效果。至多叠加三次。
--no-color	不以彩色显示输出信息。
--log *path*	将输出信息写入path参数指定的日志文件。
--no-input	忽略用户的输入。
--proxy *proxy*	指定代理。proxy参数需取一个URL。
--retries *retries*	指定尝试建立TCP连接的最大次数。默认为5次。
--timeout *sec*	指定建立TCP连接的最大等待时间。默认为15秒。
--cert *path*	指定服务器端的CA证书，用于建立SSL/TLS连接。
--client-cert *path*	指定客户端的CA证书，用于建立SSL/TLS连接。

选项	说明
--trusted-host *hostname*	强制信任指定的主机，即便它没有提供SSL/TLS连接。
--cache-dir *dir*	以dir参数指定的目录作为缓存。
--no-cache-dir	禁用缓存。
--exists-action *action*	指定当需要下载或生成的文件已经存在时的处理方式。
--disable-pip-version-check	启动时不检查pip自身的版本是否可更新。
--no-python-version-warning	不显示该版本pip关于即将不被新版本Python支持的警告。
--use-feature *feature*	使用feature参数指定的特性。
--use-deprecated *feature*	使用feature参数指定的已弃用特性。

pip的核心功能是从PyPI仓库下载分发包，并将其安装到站点设置相关目录下。第7章已经讨论了如何通过site模块完成站点设置，下面再次对其进行总结：

➤ 以sys.prefix和sys.exec_prefix指定的目录为基路径，后面添加sys.platlibdir指定的目录，得到的路径指向的是实现全局站点设置的目录，对所有用户都有效。
➤ 以site.USER_BASE指定的目录为基路径，后面添加sys.platlibdir指定的目录，得到的路径指向的是实现用户站点设置的目录，仅对登录用户有效。
➤ 上述目录的如下子目录被用于存放第三方分发包：python3.11/site-packages、python/site-packages和site-packages。

在我所使用的计算机上，存放全局第三方分发包的目录是：

```
    /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages
```

而存放用户第三方分发包的目录是：

```
    /Users/wwy/Library/Python/3.11/lib/python/site-packages
```

下面通过如下命令行验证这两个目录的内容：

```
    $ ls -F /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
site-packages/
    README.txt                          mypyc/
    __pycache__/                        pip/
    _distutils_hack/                    pip-22.2.2.dist-info/
    distutils-precedence.pth            pkg_resources/
    mypy/                               setuptools/
```

```
mypy-0.991.dist-info/                setuptools-65.3.0.dist-info/
mypy_extensions-0.4.3.dist-info/     typing_extensions-4.3.0.dist-info/
mypy_extensions.py                   typing_extensions.py

$ ls -F /Users/wwy/Library/Python/3.11/lib/python/site-packages
```

可以看出，目前我的系统中只安装了全局第三方分发包，包括pip、setuptools、mypy、mypy-extensions和typing_extensions，其中除了pip和setuptools是在安装Python时自动安装的之外，其余3个都是在第15章通过pip安装mypy时被安装的。mypy-extensions和typing_extensions都是mypy依赖的分发包。（pip和setuptools则在第15章通过pip被升级到了编写本书时的最新版。）

下面开始介绍pip命令的用法。list命令的功能是列出已经安装的分发包：

```
$ pip list
Package            Version
------------------ -------
mypy               0.991
mypy-extensions    0.4.3
pip                22.2.2
setuptools         65.3.0
typing_extensions 4.3.0
```

它以表格的形式列出了每个分发包和它的版本。

show命令显示某个已安装分发包的相关信息：

```
$ pip show mypy
Name: mypy
Version: 0.991
Summary: Optional static typing for Python
Home-page: http://www.mypy-lang.org/
Author: Jukka Lehtosalo
Author-email: jukka.lehtosalo@iki.fi
License: MIT License
Location: /Library/Frameworks/Python.framework/Versions/3.11/lib/
python3.11/site-packages
Requires: mypy-extensions, typing-extensions
Required-by:
```

从这些信息可以获知该分发包被安装到了哪个目录下，进而可以推断出它是全局第三方分发包还是用户第三方分发包。此外，还可以获得该分发包的一些元数据，包括与其他分发包的依赖关系。

check命令用于检查某个已安装分发包的依赖是否被满足：

```
$ pip check mypy
No broken requirements found.
```

---

install命令和uninstall命令用于分别安装和卸载指定的分发包。下面的例子用uninstall卸载了mypy-extensions分发包，使mypy分发包的依赖不再被满足：

```
$ pip uninstall mypy-extensions
Found existing installation: mypy-extensions 0.4.3
Uninstalling mypy-extensions-0.4.3:
  Would remove:
    /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/mypy_extensions-0.4.3.dist-info/*
    /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-
packages/mypy_extensions.py
Proceed (Y/n)? Y
  Successfully uninstalled mypy-extensions-0.4.3

$ pip check mypy
mypy 0.991 requires mypy-extensions, which is not installed.
```

只给出分发包名的话，如果该分发包已经被安装，则install什么也不做；否则install会自动安装该分发包的最新版。但很多时候我们需要指定安装分发包的哪个版本。此外，由于PyPI是一个社区维护的开源仓库，其内的分发包来自成千上万的独立开发者，必须允许这些开发者按照具体项目的需求使用最合适的版本编号方式，同时又要保证这些不同的版本编号方式具有形式上的统一性。PEP 440制定了Python项目的版本编号规则[10]，下面对该PEP文档的内容进行概括。

PyPI中的分发包的版本被称为“公有版本（public version）”，其标识符的格式为：

```
[N!]N(.N)*[{a|b|rc}N][.postN][.devN]
```

其中N代表一个自然数（包括0）。该格式可以被分为多个段。被方括号扩起来的段是可选的。下面解释各段的含义。

“N!”部分被称为“纪元段（epoch segment）”。纪元是对公有版本最粗粒度的划分，仅当该分发包发生翻天覆地的变化（例如Windows从DOS内核转换到NT内核）时，才有必要使用不同的纪元来区别。对于大部分项目来说，其公有版本标识符是没有必要用到纪元段的。PEP 440规定，当纪元段被省略时，相当于默认添加了“0!”。

公有版本标识符的核心是“N(.N)*”部分，即用“.”分隔的若干自然数，被称为“发布段（release segment）”。理论上，发布段可以包含任意多个自然数。但在现实中，通常采用如下格式之一：
“major.minor”格式：两个数字从左到右分别是主版本号和次版本号，例如0.1、0.2、1.0、1.1、……。

“major.minor.micro”格式：三个数字从左到右分别是主版本号、次版本号和微版本号，例如1.1.0、1.1.2、1.2.0、……。
主版本号变化通常意味着软件整体架构的改变，例如从Python 2到Python 3。次版本号变化通常意味着特性的增减，例如从Python 3.10到Python 3.11。微版本号变化通常意味着bug的修复，例如从Python 3.11.1到Python 3.11.2。如果一个公有版本标识符只具有发布段，或者只具有发布段和纪元段，则被视为“正式版（final release）”。

“{a|b|rc}N”部分被称为“预发布段（pre-release segment）”，也就是在正式版之前的版本。在正式版的基础上添加预发布段，则得到“预发布版（pre-release）”。软件业将预发布版分为“阿尔法测试版（alpha release）”、“贝塔测试版（beta release）”和“候选发布版（release candidate）”三个阶段，分别用“aN”、“bN”和“rcN”来表示，例如3.12.0a0、3.11.0rc1、……。

“.postN”部分被称为“后发布段（post-release segment）”，用于表示对正式版或预发布版的修订。通常是在正式发布后或开发过程中修复了某些错误，但发布段又不包括微版本号的情况下，才会用到后发布段，例如1.0.post1、1.1rc0.post0、……。只要添加了后发布段，就被称为“后发布版（post-release）”。

“.devN”部分被称为“开发版段（development release segment）”。它可以被添加在正式版、预发布版和后发布版后，表示上述版本开发过程中的某个里程碑，被统称为“开发版（development release）”，例如1.0.dev0、3.12.0a0.dev0、1.0.post1.dev1、1.1rc0.post0.dev0、……。

当我们从PyPI（或别的途径）获得了一个开源的分发包后，可以对它进行修改，而在这一开发过程中仍然需要通过某种方式区分修改后分发包的版本。但显然，我们不应该改变该分发包的公有版本标识符。PEP 440引入了“本地版本（local version）”来解决这一问题，并将版本标识符的格式扩展为：

```
public_version_identifier[+local_version_identifier]
```

也就是允许在公有版本标识符后跟一个加号，然后连接“本地版本标识符（local version identifers）”。PEP 440对本地版本标识符格式的限制较少，只包括：
1. 只能使用英文字母（区分大小写）、阿拉伯数字和“.”。
2. 不能以“.”开头和结尾。
当然，具有本地版本标识符的分发包只能在本地使用，不能上传到PyPI。

符合上述格式的版本标识符相互间总是能比较大小，比较的规则为：

- 先比较公有版本标识符，仅当两者相等时才比较本地版本标识符。
- 对于公有版本标识符，按照如下子规则比较：

a. 对于正式版，先比较纪元段的数字（省略的纪元段按照“0!”处理）。如果纪元相同再比较发布段，此时基于点号将它们分段，然后按照从左到右的顺序依次比较对应片段的数字，对于不存在的片段则自动补0。

b. 对于预发布版，首先按照a < b < rc的规则比较字母部分，当字母部分相同的才比较数字部分。此外，预发布版小于去掉预发布段后得到的正式版，但大于该正式版之前的任何正式版。

c. 对于后发布版，忽略“post”部分，直接比较数字部分。此外，后发布版大于去掉后发布版字段后得到的正式版/预发布版，但小于该正式版/预发布版之后的任何正式版/预发布版。

d. 对于开发版，忽略“dev”部分，直接比较数字部分。此外，开发版小于去掉开发版字段后得到的正式版/预发布版/后发布版，但大于该正式版/预发布版/后发布版之前的任何正式版/预发布版/后发布版。

➤ 对于本地版本标识符，基于点号将它们分段，然后按照从左到右的顺序依次按照如下子规则比较对应片段：

a. 如果两个对应片段都只包含数字，则进行数字比较。

b. 只要有一个片段包含英文字母，就进行字典顺序比较（忽略大小写），而阿拉伯数字大于任何英文字母。

事实上，一个分发包的每个版本都至少对应一个源代码分发包，还可能对应若干个二进制分发包。换句话说，我们平时谈论的“分发包”其实是个抽象概念，可能对应非常多的分发包。

下面让我们回到指定安装分发包的版本的问题。这是通过在分发包名之后增加一个用逗号分隔的“版本说明符（version specifiers）”序列实现的。版本说明符包括：

```
==version     #限制等于某个版本。
!=version     #限制不等于某个版本。
>version     #限制大于某个版本。
>=version     #限制大于等于某个版本。
<version     #限制小于某个版本。
<=version     #限制小于等于某个版本。
~=version     #限制与某个版本兼容。
===version     #限制等于某个版本，且对版本号只进行字符串比较。
```

其中“===”与“==”的不同之处在于不会为版本标识符中被省略的部分自动添加默认值（“0!”和0）。在版本说明符序列中，逗号相当于逻辑与。

版本说明符在PyPI这套软件生态系统中被广泛使用，很多情况下允许在运算符和逗号两侧添加任意多个空格。但由于在命令行中以空格作为分隔符，所以使用pip命令时，版本说明符应直接跟随分发包名，逗号两侧也不添加空格，例如：

```
mypy>=0.8,<0.9,!=0.8.2     #限制版本为0.8.x，且不能为0.8.2。
```

此外需要强调，如果命令行中的版本说明符包含“<”或“>”，则必须将整个分发包名和版本说明符序列用引号括起来，以避免“<”或“>”被识别为重定向符号。

当用版本说明符序列指定了一个版本范围时，该范围很可能包含全部上述4种版本类型，那么install会如何从PyPI选择版本呢？规则是这样的：

> ➤ 忽略所有开发版、预发布版和基于预发布版的后发布版，除非它们是该范围内的唯一选择。在后一种情况下，会给出警告，并让用户确认。
> ➤ 在剩下的正式版和基于正式版的后发布版中，选择版本最大者。

这意味着如果想安装开发版、预发布版或基于预发布版的后发布版，则最好使用==或===版本说明符。

下面的例子通过install安装了mypy-extensions 0.4.2：

```
      $ pip install mypy-extensions==0.4.2
      Collecting mypy-extensions==0.4.2
        Downloading mypy_extensions-0.4.2-py2.py3-none-any.whl (4.5 kB)
      Installing collected packages: mypy-extensions
      ERROR: pip's dependency resolver does not currently take into account all
the packages that are installed. This behaviour is the source of the following
dependency conflicts.
      mypy 0.991 requires mypy-extensions>=0.4.3, but you have mypy-extensions
0.4.2 which is incompatible.
      Successfully installed mypy-extensions-0.4.2
```

输出的提示信息说明该版本的mypy-extensions与mypy 0.991不兼容，下面验证：

```
      $ pip check mypy
      mypy 0.991 has requirement mypy-extensions>=0.4.3, but you have mypy-
extensions 0.4.2.
```

下面的例子则安装了mypy-extensions 0.4.1：

```
      $ pip install 'mypy-extensions<0.4.2'
      Collecting mypy-extensions<0.4.2
        Using cached mypy_extensions-0.4.1-py2.py3-none-any.whl (3.6 kB)
      Installing collected packages: mypy-extensions
        Attempting uninstall: mypy-extensions
          Found existing installation: mypy-extensions 0.4.2
          Uninstalling mypy-extensions-0.4.2:
            Successfully uninstalled mypy-extensions-0.4.2
      ERROR: pip's dependency resolver does not currently take into account all
the packages that are installed. This behaviour is the source of the following
dependency conflicts.
      mypy 0.991 requires mypy-extensions>=0.4.3, but you have mypy-extensions
0.4.1 which is incompatible.
      Successfully installed mypy-extensions-0.4.1
```

输出的信息说明为了安装mypy-extensions 0.4.1，先自动卸载了mypy-extensions 0.4.2。

---

list具有--outdate选项，可以列出所有能被升级的分发包：

```
$ pip list --outdate
Package         Version Latest Type
--------------- ------- ------ -----
mypy-extensions 0.4.1   0.4.3  wheel
```

注意此时输出的表格有4列，其中第三列为可以升级到的版本，而第四列则是可升级到版本对应的分发包的类型：“dist”代表源代码分发包，“wheel”代表二进制分发包。

我们可以根据list --outdate的输出结果依次手动升级分发包到最新版本，但这样做有可能会破坏分发包之间的依赖关系。一个更好的办法是在通过install命令安装一个分发包时添加--upgrade选项，这会从整体角度考虑所有已安装的分发包，将指定的分发包升级到不破坏依赖关系前提下的最新版本，亦即“最合适”的版本。此外这也会自动升级指定的分发包所依赖的其他分发包到最合适的版本，例如：

```
$ pip install --upgrade mypy
Requirement already satisfied: mypy in /Library/Frameworks/
Python.framework/Versions/3.11/lib/python3.11/site-packages (0.991)
Requirement already satisfied: typing-extensions>=3.10 in /Library/
Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages (from
mypy) (4.3.0)
Collecting mypy-extensions>=0.4.3
  Using cached mypy_extensions-0.4.3-py2.py3-none-any.whl (4.5 kB)
Installing collected packages: mypy-extensions
  Attempting uninstall: mypy-extensions
    Found existing installation: mypy-extensions 0.4.1
    Uninstalling mypy-extensions-0.4.1:
      Successfully uninstalled mypy-extensions-0.4.1
Successfully installed mypy-extensions-0.4.3
```

上面所有使用install的例子都会将分发包安装到全局站点配置相关目录下。如果想将分发包安装到用户站点配置相关目录下，则需要在使用install时添加--user选项。下面的例子为当前登录用户安装boltons分发包，并自动将它升级到最合适的版本：

```
$ pip install --user --upgrade boltons
Collecting boltons
  Downloading boltons-21.0.0-py2.py3-none-any.whl (193 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 193.7/193.7 kB 336.8 kB/s eta 0:00:00
Installing collected packages: boltons
Successfully installed boltons-21.0.0
```

可以通过如下命令行验证该分发包被安装到了用户站点配置相关目录下：

```
$ pip show boltons
Name: boltons
Version: 21.0.0
```

```
Summary: When they're not builtins, they're boltons.
Home-page: https://github.com/mahmoud/boltons
Author: Mahmoud Hashemi
Author-email: mahmoud@hatnote.com
License: BSD
Location: /Users/wwy/Library/Python/3.11/lib/python/site-packages
Requires:
Required-by:
```

需要强调的是，如果你使用的是Unix或Linux，那么当以没有root权限的用户身份登录并执行“pip install”时，会自动添加--user选项，以将分发包安装到用户站点配置相关目录下。

很多时候，我们需要同时维护多台计算机，并期望它们提供的Python环境相互兼容。“需求文件（requirements files）”就是为满足这种需要而被提出的，它们可以对Python环境进行精确地描述，且可以作为install和uninstall的-r选项的选项参数使用，以使得某台计算机上的Python环境满足/不满足该需求文件的要求。需求文件有自己的语法，具体请查阅pip官方手册，本书不详细讨论。我们只需要知道可以使用freeze命令自动生成描述当前Python环境的需求文件：

```
$ pip freeze > my_requirements.txt
$ cat my_requirements.txt
boltons==21.0.0
mypy==0.991
mypy-extensions==0.4.3
typing_extensions==4.3.0
```

然后可通过uninstall删除该环境：

```
$ pip uninstall -r my_requirements.txt
```

以及通过install复现该环境：

```
$ pip install -r my_requirements.txt
```

---

download命令的功能是下载分发包到指定目录，但并不安装。这为我们学习制作分发包提供了方便，因为PyPI中的每个分发包都可以被视作范例。表17-3列出了download命令较常用的专属选项。

表17-3. download命令专属选项

选项	说明
-d *dir*	将分发包下载到dir参数指定的目录。
--no-deps	不自动下载指定分发包所依赖的其他分发包。
--no-binary *format_control*	下载通过format_control参数列出的分发包时，强制下载源代码分发包。
--only-binary *format_control*	下载通过format_control参数列出的分发包时，强制下载二进制分发包。
--implementation *implement*	下载二进制分发包时，针对implement参数指定的Python解释器类型。
--python-version *version*	下载二进制分发包时，针对version参数指定的Python版本。
--abi *abi*	下载二进制分发包时，针对abi参数指定的ABI。
--platform *platform*	下载二进制分发包时，针对platform参数指定的平台。

➤ -d选项用于指定将分发包下载到那个目录，如果被省略则默认取当前的工作目录。

➤ 在默认情况下，下载指定的分发包时还会自动下载它们所依赖的所有其他分发包。通过提供--no-deps选项可以使得只下载指定的分发包。

➤ 在默认情况下，会优先下载指定版本范围内版本最大的分发包，而当版本相同时则优先下载二进制分发包。选项--no-binary和--only-binary共同控制着这一行为，它们都可以出现多次，且可以相互穿插，后者会叠加在前者上，不断精确对每个分发包的要求。它们的选项参数format_control可以取用逗号分隔的分发包名列表，还可以取特殊值“:all:”表示所有被直接指定的分发包和它们所依赖的其他分发包。

接下来讨论在下载二进制分发包时如何指定其“兼容性（compatibility）”。PEP 425定义了二进制分发包的“兼容性标签（compatibility tags）”[11]。简单来说，二进制分发包（也被称为“轮子”）的文件名格式为：

***distribution-version*[-*buildtag*]-*pythontag-abitag-platformtag*.whl**

其中除了可选的编译标签（buildtag）之外，Python标签（pythontag）、ABI标签（abitag）和平台标签（platformtag）都属于兼容性标签，它们共同确定了该二进制分发包的兼容性。

Python标签开头的英文字母指定了Python解释器类型，后面的数字则指定了Python解释器的版本。PEP 425定义了如下5个英文缩写：

- py：表示对Python解释器的类型没有限制。
- cp：CPython。
- jp：Jython。
- ip：IronPython。
- pp：PyPy。

其他类型的Python解释器应取sys.implementation.name引用的名称。而后面的数字是某正式版的版本标识符去掉“.”形成的，可以只包含主版本号和次版本号，例如“cp311”就代表CPython 3. 11；也可以只包含主版本号，例如“py2”表示任何Python 2解释器，“py3”表示任何Python 3解释器。特别的，Python标签还可以是通过“.”形成标签集，例如“cp39.pp71”表示兼容CPython 3.9和PyPy 7.1，而“py2.py3”则等价于兼容任何Python解释器。

如果二进制分发包不包含任何扩展模块，则其ABI标签应取值“none”。当ABI标签取其他值（例如“cp32dmu”）时，表明该二进制分发包包含扩展模块，该扩展模块依赖于该ABI标签指定的CPython ABI。

平台标签用于指定计算机的CPU的指令集架构，以及操作系统，例如“linux_x86_64”和“win32”。这同样仅当该二进制分发包包含扩展模块时才有意义，否则应取值“any”。

下面是一个二进制分发包的文件名的例子：

```
boltons-20.0.0-py2.py3-none-any.whl
```

这表明该二进制分发包兼容任何Python解释器，不依赖CPython ABI，且对计算机的CPU和操作系统无任何限制。

在通过download下载二进制分发包时，可以通过下列选项指定其兼容性：

- --implementation：其参数为Python标签的英文字母部分。
- --python-version：其参数为Python解释器的某正式版的版本标识符（包括“.”），对应Python标签的数字部分。
- --abi：其参数为ABI标签。
- --platform：其参数为平台标签。

当省略了--implementation、--python-version和--abi选项时，会根据执行pip的Python解释器自动判断；当省略了--platform选项时，会根据当前计算机自动判断。

下面的例子下载mypy的二进制分发包到package_mypy目录下，但不下载它所依赖的分发包：

```
$ pip download -d package_mypy --no-deps --only-binary mypy mypy==0.991
Collecting mypy
  Using cached mypy-0.991-py3-none-any.whl (2.6 MB)
Saved ./package_mypy/mypy-0.991-py3-none-any.whl
Successfully downloaded mypy
```

下面的例子则下载mypy的源代码分发包到package_mypy目录下，同时下载它所依赖的mypy_extensions和typing_extensions的二进制分发包到相同目录：

```
$ pip download -d package_mypy  --only-binary :all: --no-binary mypy
mypy==0.991
Collecting mypy
  Using cached mypy-0.991.tar.gz (2.8 MB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Collecting typing-extensions>=3.10
  Using cached typing_extensions-4.3.0-py3-none-any.whl (25 kB)
Collecting mypy-extensions>=0.4.3
  Using cached mypy_extensions-0.4.3-py2.py3-none-any.whl (4.5 kB)
Saved ./package_mypy/mypy-0.991.tar.gz
Saved ./package_mypy/mypy_extensions-0.4.3-py2.py3-none-any.whl
Saved ./package_mypy/typing_extensions-4.3.0-py3-none-any.whl
Successfully downloaded mypy mypy-extensions typing-extensions
```

注意该例子先通过--only-binary选项指定所有分发包都强制下载二进制版本，然后又通过--no-binary选项指定对mypy强制下载源代码版本。

下面的例子则下载mypy_extensions和typing_extensions的源代码分发包到package_mypy目录下：

```
$ pip download -d package_mypy --no-binary mypy_extensions,
typing_extensions mypy_extensions==0.4.3 typing_extensions==4.3.0
Collecting mypy_extensions
  Using cached mypy_extensions-0.4.3.tar.gz (4.3 kB)
  Preparing metadata (setup.py) ... done
Collecting typing_extensions
  Using cached typing_extensions-4.3.0.tar.gz (47 kB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Saved ./package_mypy/mypy_extensions-0.4.3.tar.gz
Saved ./package_mypy/typing_extensions-4.3.0.tar.gz
Successfully downloaded mypy_extensions typing_extensions
```

当通过download下载了某个分发包后，可以通过install进行本地安装，此时不能给出分发包名，而需要给出指向已下载分发包的路径。下面是一个例子：

```
$ pip download -d package_boltons boltons==20.0.0

$ pip download --no-binary :all: -d package_boltons boltons==20.0.0

$ pip install --upgrade package_boltons/boltons-20.0.0-py2.py3-none-any.whl
Processing ./package_boltons/boltons-20.0.0-py2.py3-none-any.whl
Installing collected packages: boltons
  Attempting uninstall: boltons
    Found existing installation: boltons 21.0.0
    Uninstalling boltons-21.0.0:
      Successfully uninstalled boltons-21.0.0
Successfully installed boltons-20.0.0
```

可以通过如下命令行验证该boltons分发包被安装到全局站点设置相关目录下：

```
$ pip show boltons
Name: boltons
Version: 20.0.0
Summary: When they're not builtins, they're boltons.
Home-page: https://github.com/mahmoud/boltons
Author: Mahmoud Hashemi
Author-email: mahmoud@hatnote.com
License: BSD
Location: /Library/Frameworks/Python.framework/Versions/3.11/lib/
python3.11/site-packages
Requires:
Required-by:
```

而下面的命令行则证明安装在用户站点设置相关目录下的boltons并没有被删除：

```
$ ls /Users/wwy/Library/Python/3.11/lib/python/site-packages/
boltons
```

---

install还有一些较高级的功能，例如从PyPI之外的公共仓库、版本控制系统和本地归档文件安装分发包，本书不详细讨论。感兴趣的读者请查阅pip官方手册。

17-3. 虚拟环境

（教程：12）
（标准库：venv）

“虚拟环境（virtual environments）”是安装第三方模块时经常遇到的一个重要概念。第7章讨论site模块时给出的例子可以证明，在sys.path的取值中用户站点设置目录下的site-packages位于全局站点设置目录下的site-packages之前。这意味着用户第三方分发包的优先级高于全局第三方分发包。基于这一机制，用户已经可以实现个性化。那么为什么还需要虚拟环境呢？

Python并不像其他编程语言那样，新版本为了兼容旧版本，会勉强保留不合适的特性，甚至为此扭曲新特性。

Python为了最优化自身的语法，当决定放弃一个特性后，就会在下个次版本直接不支持该特性，并通过PendingDeprecationWarning警告通知开发者。这会导致一个第三方模块能够被当前版本的Python解释器正常使用，却无法被新版本的Python解释器正常使用，迫使该模块升级。然而分发包相互间是存在依赖关系的，不可能严格地在某一时刻同步升级，所以不得不保留多个版本，其中一些版本使用了被淘汰的特性，而一些版本使用了新特性，它们会要求使用不同版本的Python解释器。 虚拟环境就是为了解决该问题而被引入的。

任何Python解释器都支持在一台计算机上同时安装多个版本，精确到次版本号，即可以同时安装Python 3.10和Python 3.11，但不能同时安装Python 3.11.1和Python 3.11.2。在第2章已经说明，在Unix和类Unix操作系统中，python命令指向的是Python 2，而python3命令指向的是Python 3。事实上我们也可以通过在关键字python后添加主版本号和次版本号来精确指定启动哪个Python解释器，例如：

```
$ python3.11
>>> ^D
$ python2.7
>>> ^D
```

在我的计算机中其实python命令等价于python2.7，python3命令等价于python3.11。如果同时安装了Python 3.10和Python3.11，则可以通过设置符号链接的方式控制python3指向哪个Python解释器。

创建虚拟环境需要用到标准库中的venv模块。该模块的前身是第三方分发包virtualenv，该分发包甚至早于pip出现。venv对virtualenv做了简化。使用venv创建虚拟环境时，用指定的Python解释器通过-m选项直接执行venv模块，并提供一个指向目录的路径作为命令行参数。venv会在该目录下创建虚拟环境，如果该目录不存在则自动创建。

请通过下面的命令行在工作目录的example_env子目录下创建一个虚拟环境：

```
$ python3.11 -m venv example_env
```

该虚拟环境的目录结构为：

```
└─ example_env/
    ├─ pyvenv.cfg
    ├─ bin/
    │   ├─ Activate.ps1
    │   ├─ activate
    │   ├─ activate.csh
    │   ├─ activate.fish
    │   ├─ pip
    │   ├─ pip3
    │   ├─ pip3.11
    │   ├─ python
    │   ├─ python3
    │   └─ python3.11
    ├─ include/
    │   └─ python3.11/
    └─ lib/
        └─ python3.11/
            └─ site-packages/
                └─ ...
```

其中pyvenv.cfg文件记录了创建该虚拟环境的命令行，以及使用的Python解释器，这就使该Python解释器与该虚拟环境绑定。bin目录下的python、python3和python3.11都是指向绑定Python解释器的符号链接，pip、pip3和pip3.11则是匹配该Python解释器的pip，而剩下的四个文件则是用来激活该虚拟环境的脚本。include和lib目录下都包含python3.11子目录，前者用于安装C/C++库，后者用于安装分发包。

使用虚拟环境时必须先将其激活。在Unix和类Unix操作系统上，如果你使用的是bash或兼容bash的shell（这是最典型的情况），则应执行activate脚本；如果使用csh，则应执行activate.csh脚本；如果使用fish shell，则应执行activate.fish脚本。而在Windows上，这三个脚本都被activate.bat替代。Activate.ps1是用来支持在Windows上用PowerShell远程登录Unix或类Unix操作系统的场景。而不论通过哪种方式激活了虚拟环境，都是通过执行deactivate命令来退出虚拟环境。

当一个虚拟环境被激活后，“python3”和“pip”就将对应该虚拟环境的bin目录下的相应文件，而操作系统中安装的其他版本的Python解释器和pip就被屏蔽了。此外，此时启动Python解释器会自动修改sys.prefix和sys.exec_prefix到该虚拟环境所在目录，并禁用用户站点设置。请通过如下命令行和语句验证：

```
$ source example_env/bin/activate

(example_env)$ python3 -m site
sys.path = [
    '/Users/wwy',
    '/Library/Frameworks/Python.framework/Versions/3.11/lib/python311.zip',
    '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11',
    '/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/lib-
dynload',
    '/Users/wwy/example_env/lib/python3.11/site-packages',
]
```

```
      USER_BASE: '/Users/wwy/Library/Python/3.11' (exists)
      USER_SITE: '/Users/wwy/Library/Python/3.11/lib/python/site-
packages' (exists)
      ENABLE_USER_SITE: False

      (example_env)$ python3

      >>> import sys
      >>> sys.prefix
      '/Users/wwy/example_env'
      >>> sys.exec_prefix
      '/Users/wwy/example_env'
      >>> ^D

      (example_env)$ deactivate
```

在激活了虚拟环境后，通过pip进行的一切操作都会作用于该虚拟环境的lib目录下储存第三方分发包的目录，在我的系统中是python3.11/site-packages。下面的例子查看该虚拟环境默认安装了哪些分发包：

```
      $ source /Users/wwy/example_env/bin/activate

      (example_env)$ pip list
      Package    Version
      ---------- -------
      pip        22.0.4
      setuptools 58.1.0
      WARNING: You are using pip version 22.0.4; however, version 22.2.2 is
available.
      You should consider upgrading via the '/Users/wwy/example_env/bin/python3
-m pip install --upgrade pip' command.
```

该结果说明创建虚拟环境后，会自动安装pip和setuptools，但它们都需要升级。此外，在全局站点设置相关目录下安装的mypy和在用户站点设置相关目录下安装的boltons在该虚拟环境下都不可见。

下面对虚拟环境中的pip和setuptools进行升级：

```
      (example_env)$ pip install --upgrade pip setuptools
      Requirement already satisfied: pip in ./example_env/lib/python3.11/site-
packages (22.0.4)
      Collecting pip
        Using cached pip-22.2.2-py3-none-any.whl (2.0 MB)
      Requirement already satisfied: setuptools in ./example_env/lib/python3.11/
site-packages (58.1.0)
      Collecting setuptools
        Using cached setuptools-65.3.0-py3-none-any.whl (1.2 MB)
      Installing collected packages: setuptools, pip
        Attempting uninstall: setuptools
          Found existing installation: setuptools 58.1.0
          Uninstalling setuptools-58.1.0:
            Successfully uninstalled setuptools-58.1.0
        Attempting uninstall: pip
          Found existing installation: pip 22.0.4
          Uninstalling pip-22.0.4:
            Successfully uninstalled pip-22.0.4
```

```
      Successfully installed pip-22.2.2 setuptools-65.3.0
```

下面的例子给该虚拟环境安装了sampleproject，并指定了“额外特性（extras）”：

```
      (example_env)$ pip install --upgrade sampleproject[test]
      Collecting sampleproject[test]
        Using cached sampleproject-2.0.0-py3-none-any.whl (4.2 kB)
      Collecting peppercorn
        Using cached peppercorn-0.6-py3-none-any.whl (4.8 kB)
      Collecting coverage
        Using cached coverage-6.4.4-cp311-cp311-macosx_10_9_x86_64.whl (184 kB)
Installing collected packages: peppercorn, sampleproject, coverage
Successfully installed coverage-6.4.4 peppercorn-0.6 sampleproject-2.0.0

      (example_env)$ deactivate
```

可以验证在该虚拟环境储存第三方分发包的目录下多出了sampleproject、peppercorn和coverage。安装一个分发包时指定额外特性的方式是在该分发包的包名后面跟一个方括号，其内是用逗号分隔的额外特性列表（不能有空格）。上面的例子指定了额外特性test，这会导致额外安装coverage分发包。

显然，在一台计算机中可以安装任意多个虚拟环境，分别绑定不同的Python解释器，并安装不同的分发包。这些虚拟环境相互间不会影响，当不需要某个虚拟环境时，在该虚拟环境未激活的情况下删除该目录即可。当然，使用虚拟环境也是有代价的，即很可能需要安装同一分发包的多个副本，因而占用更多磁盘空间。但相较于虚拟环境给解决依赖地狱问题带来的便利，占用更多磁盘空间的缺点可以被接受。一般而言，每启动一个新的Python项目，就应创建一个新的虚拟环境，以使这些项目互不干扰。

PyPA还提供了综合利用pip和虚拟环境的高级工具，例如pipenv和pipx。对这些工具的讨论超出了本书的范围，感兴趣的读者请参阅这些工具的官方手册[12][13]。

17-4. setuptools和setup.py

现在你已经知道了如何使用别人上传到PyPI的分发包。相信你也很想知道如何将自己的Python项目制作成分发包，并上传到PyPI以分享到整个Python社区。然而在尝试打包自己的Python项目之前，你应该首先学会看懂PyPI中已有的分发包是如何打包的。本节和下一节将讨论Python源代码包中元数据的格式，以及处理它们的工具。

与安装分发包一直依赖于pip工具不同，制作分发包的工具经历了复杂的演化过程。本书将该演化过程分为4个阶段。

➤ 第一个阶段为基于distutils进行打包和分发。

PyPI是2003年出现的，然而从2000年开始，Python开发者已经在使用标准库中的distutils模块来制作分发包了。截止Python 3.11，distutils模块依然保留在Python标准库中，但官方手册中的相关文档提示该模块已经被弃用，且将在Python 3.12被移除。

distutils只能制作源代码分发包，且它们只针对Unix和类Unix操作系统。这些分发包被称为“sdist包”，是归档文件（后缀名.zip、.tar、.tgz等等），通常通过U盘、CD或电子邮件传播，就好像Linux发行版在当时的传播方式一样。sdist包内的存储元数据的核心文件是setup.py，它同时也是一个可被执行的Python脚本，且支持“install”、“uninstall”等命令行参数，因此还覆盖了pip的功能。

➤ 第二个阶段是用setuptools代替distutils。

发布于2004年的setuptools是distutils的增强版，突破了distutils的局限性，支持Windows和二进制分发包。用setuptools制作的源代码分发包依然是sdist包，而用setuptools制作二进制分发包则被称为“egg包”。

egg包本质上是一个ZIP归档文件，但后缀名是.egg。setuptools沿用了setup.py，且同样支持直接执行它来安装sdist包，但egg包则需要通过easy_install.py来安装。然而PyPI提供了twine来实现分发包的上传，pip来实现分发包的下载和安装，所以一般而言setup.py脚本只被用来制作sdist包和egg包。

➤ 第三个阶段是用“轮子（wheels）”替代了egg包。

egg包的格式设计存在一些问题，例如它会直接包含.pyc文件，这使得系统中的Python解释器必须与制作egg包时使用的Python解释器完全一致。egg包的命令规则也缺乏足够的兼容性信息。为了解决这些问题，轮子格式于2013年被推出，它通过PEP 345[14]、PEP 376[15]、PEP 425和PEP 427[16]共同定义。pip和setuptools随后被扩展为支持轮子。

➤ 第四个阶段是将Python的“编译系统（build systems）”分解为前端和后端两部分，淡化setuptools的作用。

2016～2017年间通过的PEP 517[17]和PEP 518[18]明确了编译系统的概念，并定义了pyproject.toml文件以向编译系统前端（例如build、hatch、poetry等）提供信息，使其知道该使用哪种编译系统后端（例如setuptools、flit、hatching、pdm等）。这使得setup.py逐渐演变为一个纯元数据文件，不再推荐将其当成Python脚本来使用[19]。随后，setuptools的编写者推出了setup.cfg来代替setup.py，以避免用一个可执行的脚本来提供元数据。PEP 621则提出了直接在pyproject.toml文件中储存元数据的方法[20]。

本书不想过多讨论已经过时的打包技术。然而截止Python 3.11，在安装CPython的同时除了会自动安装pip外，还会自动安装setuptools，因为setuptools依然是使用最广泛的

Python编译系统后端。虽然setup.py现在已经不是必须的了，但由于它在历史上造成了巨大影响，从setup.py开始讨论Python的打包技术是最自然的，否则你将无法理解PyPI中已有的大量分发包的构成。本书不可能覆盖setuptools的所有用法，关于setuptools的权威信息请参考其官方手册[21]。

为了便于讨论，先要获得一些分发包的例子。在上一节我们已经下载了mypy 0.991和boltons 20.0.0的sdist包和轮子，下面通过如下命令行下载sampleproject 2.0.0的sdist包和轮子：

```
$ pip download -d package_sampleproject sampleproject==2.0.0

$ pip download --no-binary :all: -d package_sampleproject
sampleproject==2.0.0
```

注意sampleproject是PyPA提供的一个专门为学习Python打包技术的范例，下面的讨论会主要针对该分发包。

---

setup.py可以被概括为从setuptools模块导入setup()函数，然后对其进行一次调用，即可以总结为如下模板：

```
from setuptools import setup

setup(
    ... keywords assignments ...
)
```

不论创建分发布还是安装分发布，setup.py都会被执行，进而调用setup()。setup.py储存的元数据事实上表现为对setup()函数的仅关键字形式参数传入的实际参数。附录 VI列出了setup()的所有（未被淘汰的）仅关键字形式参数。为了简便起见，下面将这些参数称为setup()的“关键字”。

下列关键字用于提供元数据，且不影响分发包的选择：name、version、description、long_description、long_description_content_type、author、author_email、maintainer、maintainer_email、license、license_files、url、download_url和project_urls。它们包含的信息会被pip命令show显示，如果该分发包被上传到PyPI则还会在相关Web页面上被显示。这些关键字绝大部分需被传入一个字符串，而该字符串本身就是相应元数据。下面仅对较特殊的几个进一步解释。

name被传入的分发包名必须符合PEP 503的要求[22]，即仅包含小写英文字母、阿拉伯数字和连字符，且只能以英文字母开头。

long_description_content_type虽然也需要被传入一个字符串，但该字符串必须是一个格式为“major/minor”的MIME类型[23]，以说明long_description被传入的字符串该如何解

读。事实上，与description是分发包的一句话总结不同，long_description是对分发包的详细描述。

PyPI要求我们将对该分发包的详细描述储存在文本文件README.md或README.rst中，这些信息会通过PyPI的Web接口被显示。如果该文件的内容不是太多，则可以在setup.py中通过I/O操作从该文件读取所有文本传入long_description。例如在sampleproject的setup.py中是这样处理的：

```
from os import path

#获取setup.py所在目录的绝对路径。
here = path.abspath(path.dirname(__file__))

#在setup.py所在目录下寻找README.md文件，并读取其全部内容。
with open(path.join(here, 'README.md'), encoding='utf-8') as f:
    long_description = f.read()

setup(
    ...
    long_description=long_description,
    long_description_content_type='text/markdown',
    ...
)
```

注意README.md必须与setup.py放在同一个目录下，因此这段代码没有问题。

然而很多分发包的README.md或README.rst非常长，不适合传入long_description来使用。此时可以直接给一个变量赋值一个字符串，然后再将该变量传入long_description。mypy的setup.py就是这样处理的：

```
long_description = '''
Mypy -- Optional Static Typing for Python
=========================================

Add type annotations to your Python programs, and use mypy to type
check them.  Mypy is essentially a Python linter on steroids, and it
can catch many programming errors by analyzing your program, without
actually having to run it.  Mypy has a powerful type system with
features such as type inference, gradual typing, generics and union
types.
'''.lstrip()

setup(
    ...
    long_description=long_description,
    ...
)
```

而boltons的setup.py则采用了更有趣的处理方式，给long_description传入了其自身的文档字符串：

```
    setup(
        ...
        long_description=__doc__,
        ...
    )
```

在上述两种情况下都没有必要设置long_description_content_type，因为它的默认值就是“text/plain”，代表一段纯文本。

分发包的许可证需要license和license_files共同描述。license给出的是许可证的类型，例如“MIT License”（mypy中的setup.py）和“BSD”（boltons中的setup.py），如果被省略则需要通过classifiers来指定（sampleproject中的setup.py）。而许可证的内容则总是储存在一个文本文件中的，但文件名可以有很多种。license_files用于给出搜索文件名的范围，默认值是['LICEN[CS]E*', 'COPYING*', 'NOTICE*', 'AUTHORS*']，因此文件名以“License”、“Licence”、“Copying”、“Notice”和“Authors”开头（不区分大小写）的文件将被识别为许可证。一般而言，我们不需要改变license_files的默认值，只需要让我们提供的许可证的文件名符合上述规范。sampleproject的许可证是LICENSE.txt，mypy和boltons的许可证都是LICENSE。

当该分发包所属项目有自己的网站时，可以通过url给出该网站。如果该分发包除了PyPI之外还有额外的下载地址，则可以通过download_url给出。project_urls仅当还有其他关于该分发包的在线资源时才需要使用，例如在mypy的setup.py中有：

```
    setup(
        ...
        project_urls={
            'News': 'http://mypy-lang.org/news.html',
            'Documentation': 'https://mypy.readthedocs.io/en/stable/
index.html',
            'Repository': 'https://github.com/python/mypy',
        },
        ...
    )
```

接下来讨论关键字classifiers、keywords、platforms、provides和obsoletes，它们也用于提供元数据，但在安装或下载分发包时会影响分发包的选择。

classifiers是最重要的选择依据，它的取值是一个字符串列表，而其中的每个字符串都通过“::”被分隔为多段。下面是sampleproject的setup.py中的classifiers设置：

```
    setup(
        ...
        classifiers=[
            'Development Status :: 3 - Alpha',
            'Intended Audience :: Developers',
            'Topic :: Software Development :: Build Tools',
            'License :: OSI Approved :: MIT License',
            'Programming Language :: Python :: 3',
            'Programming Language :: Python :: 3.5',
```

```
            'Programming Language :: Python :: 3.6',
            'Programming Language :: Python :: 3.7',
            'Programming Language :: Python :: 3.8',
            'Programming Language :: Python :: 3 :: Only',
        ],
        ...
    )
```

这种数据结构是通过PEP 301定义的，本质上是一棵树，对于每个“::”来说，其左边段是父节点，右边段是子节点[24]。而该树的根节点的子节点可能是：

- “Development Status”：该分发包的完成度。
- “Environment”：该分发包的运行环境。
- “Framework”：该分发包使用的框架。
- “Intended Audience”：该分发包针对的用户群体。
- “License”：该分发包的许可证类型。
- “Natural Language”：该分发包使用的自然语言。
- “Operating System”：该分发包支持的操作系统。
- “Programming Language”：该分发包支持的编程语言。
- “Topic”：该分发包针对的技术话题。
- “Typing”：该分发包使用类型注解的方式。

该树的每个叶子都对应一个分类器。PyPI提供了一个动态维护的分类器列表[25]，只需要从该列表中选取符合对分发包的描述的分类器即可。

值得特别说明的是Programming Language项：对于Python之外的编程语言，都只精确到语言本身，而对于Python却精确到次版本。这意味着理论上Python每发布一个新版本就需要检查该分布包是否支持该版本，并更新setup.py。然而这样做非常麻烦，如果某个分发包的维护频率没有那么高，就无法对新版本做出及时响应。该问题的解决办法是首先在如下三者里选一：

- 如果该分发包同时支持Python 2和Python 3，则包含“Programming Language :: Python”，或者同时包含“Programming Language :: Python :: 3”和“Programming Language :: Python :: 2”。
- 如果该分发包只支持Python3，则同时包含“Programming Language :: Python :: 3”和“Programming Language :: Python :: 3 :: Only”。
- 如果该分发包只支持Python2，则同时包含“Programming Language :: Python :: 2”和“Programming Language :: Python :: 2 :: Only”。

如果该分发包对Python解释器类型有要求，则包含对应支持的Python解释器版本的分类器，精确到次版本号。被明确给出的次版本号经过了实测，肯定支持。而未明确给出的次版本，如果小于所有明确给出的次版本号则肯定不支持，否则不确定。

keywords被传入的字符串列表或逗号分隔的字符串可以包含任何关键字，它们主要被用于支持PyPI的站内搜索引擎。在sampleproject的setup.py中有：

```
setup(
    ...
    keywords='sample setuptools development',
    ...
)
```

platforms被传入的字符串列表或逗号分隔的字符串应包含PEP 425规定的平台标签，例如['win32', 'linux_x86_64']或'win32, linux_x86_64'。platforms的默认值时'any'，表示该分发包适用于任何平台。本书作为例子的三个分发包都是适用于任何平台的，但只有boltons显式包含了platforms：

```
setup(
    ...
    platforms='any',
    ...
)
```

我们还可以通过provides关键字说明该分发包内提供了哪些包，以及通过obsoletes关键字说明该分发包可以替代哪些其他分发包。然而pip会忽略这两个关键字。

接下来讨论的关键字会影响分发包的安装过程，相当于一些配置选项。下列关键字描述了分发包的结构：package_dir、packages、py_modules、ext_modules、ext_package、package_data、include_package_data和exclude_package_data。

分发包有两种布局方式。在“平坦布局（flat-layout）”中，该分发包所包含的包和脚本会被直接放在setup.py所在目录下。而在“src布局（scr-layout）”中，会在setup.py所在目录下创建一个专门的子目录（目录名通常是“src”），然后将该分发包所包含的包和脚本都放在该子目录下。setuptools默认会认为分发包采用平坦布局。为了告诉setuptools该分发包采用了src布局，必须让传入package_dir的字典包含特殊键空串，并让该键的值为存放包和脚本的子目录的目录名。mypy和boltons都采用平坦布局，故省略了package_dir关键字。sampleproject则采用src布局，因此有：

```
setup(
    ...
    package_dir={'': 'src'},
    ...
)
```

除了表明采用了src布局之外，传入package_dir的字典的其他键值对用于定义一个路径映射，其格式都是'pkg': 'path'。其中pkg为分发包所包含的某个包或模块的目录名/文件名，而path则是一个以该分发包被安装到的目录（.../site-packages）为基路径的相对路径。这样就强制指定将该包或模块安装到指定位置。但建议尽量避免使用这一技巧，而应使包和模

块在分发包中位置（相对于setup.py所在目录或src子目录）与它们被安装后的位置（相对于site-packages）相同。

packages用于显式指定该分发包所包含的包，需要被传入一个列表，列表中的每个元素代表一个包名。boltons的setup.py包含下面的设置：

```
setup(
    ...
    packages=['boltons'],
    ...
)
```

这说明分发包boltons内只包含一个包，其名称也是“boltons”。当分发包中包含多个包时，则可以从setuptools导入find_packages()函数，该函数能自动在指定的目录下搜索包，并返回包含所有包名的列表。sampleproject的setup.py包含下面的设置：

```
from setuptools import setup, find_packages

setup(
    ...
    packages=find_packages(where='src'),
    ...
)
```

这相当于指定在setup.py所在目录的子目录src内搜索包。而mypy的setup.py则包含下面的设置：

```
from setuptools import setup, find_packages

setup(
    ...
    packages=find_packages(),
    ...
)
```

这相当于指定在setup.py所在目录内搜索包。

py_modules则用于显式指定分发包所包含的脚本，或者说以独立的脚本形式存在的模块，同样需要被传入一个列表，列表中的每个元素代表一个模块名。举例来说，如果一个分发包包含了脚本foo.py和bar.py，则需要包含如下设置：

```
setup(
    ...
    py_modules=['foo', 'bar'],
    ...
)
```

注意find_packages()函数无法用于发现独立的脚本。

如果同时省略了packages和py_modules，则setuptools会尝试自动搜索分发包中的模块和脚本。该功能被称为“自动发现（automatic discovery）”，但截至setuptools 65.3.0都还处于beta测试版阶段。

当分发包中包含扩展模块时，就需要通过ext_modules来指定它们了。由于扩展模块是通过C（或兼容C的编程语言）编写的，需要使用相应的编译器来编译成.so，.dll或.pyd文件，然后放在它们所属包的目录下（如果放在setup.py所在目录下，则等同于独立脚本）。为了创建扩展模块，setuptools定义了Extension类，其实例化语法为：

```
class setuptools.Extension(name, sources, include_dirs,
define_macros, undef_macros, library_dirs, libraries,
runtime_library_dirs, extra_objects, extra_compile_args,
extra_link_args, export_symbols, depends, language, optional)
```

表17-4列出了各参数的含义。Extension类的每次实例化都对应一次编译过程，编译结果是名为name的扩展模块（这同时也指定了该扩展模块属于哪个包）。

表17-4. Extension类的参数

参数	类型	说明
name	字符串	扩展模块的完整名字，包括它所属的包。
sources	字符串列表	所有源代码文件的路径。
include_dirs	字符串列表	C/C++头文件所在目录的路径。
define_macros	二元组列表	每个(name, value)形式的二元组代表一个#define指令。
undef_macros	字符串列表	每个字符串代表一个#undefine指令。
library_dirs	字符串列表	用于搜索静态链接库的目录的路径。
libraries	字符串列表	涉及的所有库的名称。
runtime_library_dirs	字符串列表	用于搜索动态链接库的目录的路径。
extra_objects	字符串列表	所有资源文件的路径。
extra_compile_args	字符串列表	额外的编译参数。
extra_link_args	字符串列表	额外的链接参数。
export_symbols	字符串列表	该扩展模块导出的符号。
depends	字符串列表	该扩展模块依赖的文件。
language	字符串	源代码使用的编程语言，例如“c”、“c++”和“objc”。
optional	布尔值	该扩展模块是否是可选的。默认取False，如果传入True，则该扩展模块编译失败不会报错，而会被直接跳过。

下面的例子创建了一个名为foo的扩展模块，它由ext/foo目录下的file1.c和file2.c编译得到，属于pkg包：

```
from setuptools import Extension

ext1 = Extension('pkg.foo', ['ext/foo/file1.c', 'ext/foo/file2.c'])
```

而下面的例子则创建了一个名为bar的扩展模块，它由ext/bar目录下的file.cpp编译得到，不属于任何包：

```
from setuptools import Extension

ext2 = Extension('bar', ['ext/bar/file.cpp'])
```

ext_modules需被传入一个Extension对象形成的列表。sampleproject和boltons没有包含扩展模块。mypy包含了扩展模块，因此其setup.py中有：

```
if USE_MYPYC:
    ...
    from mypyc.build import mypycify
    ext_modules = mypycify(
        mypyc_targets + ['--config-file=mypy_bootstrap.ini'],
        opt_level=opt_level,
        debug_level=debug_level,
        # Use multi-file compilation mode on windows because without it
        # our Appveyor builds run out of memory sometimes.
        multi_file=sys.platform == 'win32' or force_multifile,
    )
    ...
else:
    ext_modules = []

setup(
    ...
    ext_modules = ext_modules
    ...
)
```

这意味着ext_modules关键字被传入的是调用mypycify()得到的返回值，而该函数是在mypyc.build模块中定义的，打开mypyc/build.py可以找到相关函数定义语句：

```
def mypycify(
    paths: List[str],
    *,
    only_compile_paths: Optional[Iterable[str]] = None,
    verbose: bool = False,
    opt_level: str = "3",
    debug_level: str = "1",
    strip_asserts: bool = False,
```

```
        multi_file: bool = False,
        separate: Union[bool, List[Tuple[List[str], Optional[str]]]] = False,
        skip_cgen_input: Optional[Any] = None,
        target_dir: Optional[str] = None,
        include_runtime_files: Optional[bool] = None
    ) -> List['Extension']:
    ...
```

这说明ext_modules关键字被传入了一个Extension对象列表。

如果一个分发包中的所有扩展模块都属于一个特定的包，则不需要在创建Extension对象时通过扩展模块名指定该包，而是通过ext_pacakge给出该包。例如下面的设置使得扩展模块foo和bar都属于包pkg：

```
setup(
    ...
    ext_package = 'pkg',
    ext_modules = [Extension('foo', ['foo.c']),
                   Extension('bar', ['bar.c'])],
    ...
)
```

很多时候，一个包除了实现模块或扩展模块的文件外，还包含一些在执行这些模块或扩展模块中的代码时被访问的文件，后者被通称为“数据文件（data files）”。指定数据文件有两种方式，下面依次讨论。

通过package_data指定数据文件是被推荐的方式，因为它更准确直观。package_data需被传入一个字典，该字典的键是包名，而值则是一个可以含有通配符的模式列表，其含义是对于指定包，其内文件名匹配某个指定模式的文件被视为数据文件。sampleproject的setup.py包含如下设置：

```
setup(
    ...
    package_data={
        'sample': ['package_data.dat'],
    },
    ...
)
```

这意味着在其包含的sample包下的package_data.dat文件被视为数据文件。然而该例子没有用到通配符。下面再给一个例子，假设一个分发包包含了foo和bar两个包，这两个包下都有若干.txt文件，而只有bar下有若干.rst文件，则可以通过如下设置使得这些文件都被视为数据文件：

```
setup(
    ...
    package_data={
        'foo': ['*.txt'],
        'bar': ['*.txt', '*.rst'],
```

```
        },
        ...
    )
```

而此时还有一种等价写法：

```
    setup(
        ...
        package_data={
            '': ['*.txt'],
            'bar': ['*.rst'],
        },
        ...
    )
```

注意特殊键空串代表任何包，所以其内的模式将适用于分发包包含的所有包。

include_package_data则依赖于MANIFEST.in文件来指定包中的数据文件。它需被传入一个布尔值，默认值为False。需要强调的是，MANIFEST.in文件并不是专门用来指定数据文件的，它的基本职能是在创建分发包时，指定分发包应包含项目目录下的哪些文件。当给include_package_data传入TRUE时，相当于将分发包的每个包内除了用于实现模块和扩展模块的文件之外的所有文件都视为数据文件。boltons的setup.py包含如下设置：

```
    setup(
        ...
        include_package_data=True,
        ...
    )
```

而boltons的MANIFEST.in文件的内容是：

```
    include CHANGELOG.md
    include LICENSE
    include README.md
    include TODO.rst
    include pytest.ini
    include tox.ini
    recursive-include docs
    recursive-include tests
```

这会导致boltons分发包包含的任何包下的docs目录和tests目录中的文件都被视为数据文件。关于MANIFEST.in的语法会在下一节详细讨论。

package_data和include_package_data并不矛盾。在mypy的setup.py中，同时使用了两者：

```
    package_data = ['py.typed']
    package_data += find_package_data(os.path.join('mypy', 'typeshed'),
['*.py', '*.pyi'])
    package_data += [os.path.join('mypy', 'typeshed', 'stdlib', 'VERSIONS')]
    package_data += find_package_data(os.path.join('mypy', 'xml'), ['*.xsd',
'*.xslt', '*.css'])

    setup(
        ...
        package_data={'mypy': package_data},
        ...
        include_package_data=True,
        ...
    )
```

当给include_package_data传入True后，package_data通常被用来指定在安装分发包的过程中动态生成的文件。

exclude_package_data是对package_data和include_package_data的补充，后两者相当于做加法，而前者相当于做减法。举例来说，如下设置使包foo和bar下的temp.txt文件，以及bar下的0.rst、1.rst、……、9.rst文件不被视为数据文件：

```
    setup(
        ...
        package_data={
            '': ['*.txt'],
            'bar': ['*.rst'],
        },
        exclude_pacakge_data={
            '': ['temp.txt'],
            'bar': ['[0-9].rst'],
        }
        ...
    )
```

python_requires、install_requires、extras_require和setup_requires说明了分发包对Python解释器的要求，以及和其他分发包之间的依赖关系。它们会被编译系统前端严格遵循，因而与classifiers有本质区别。

python_requires需被传入一个版本说明符序列，例如在sampleproject的setup.py中有如下设置：

```
    setup(
        ...
        python_requires='>=3.5, <4',
        ...
    )
```

这意味着计算机中必须安装有Python 3.5以上版本才能使用sampleproject，但如果安装的是Python 4及更高版本（未来的Python）也不能使用sampleproject。当该条件不被满足时，pip会拒绝安装sampleproject并报错。

install_requires需被传入一个“依赖规格（dependency specification）”形成的列表，以表明该分发包依赖于列出的其他分发包。最简单的依赖规格仅包含一个分发包名，例如在sampleproject的setup.py中有如下设置：

```
setup(
    ...
    install_requires=['peppercorn'],
    ...
)
```

这意味着sampleproject本身仅依赖于peppercorn。上一节已经说明了pip对这种依赖关系的处理方式：如果可能，自动下载安装被依赖的分发包，否则报错。

依赖规格的语法通过PEP 508定义[26]。简单来说，一个依赖规格包含分发包名、版本说明符序列和“环境标记（environment markers）”表达式。环境标记表达式和版本说明符序列之间用分号分隔。mypy的setup.py中则包含如下设置：

```
setup(
    ...
    install_requires=["typed_ast >= 1.4.0, < 2; python_version<'3.8'",
                      'typing_extensions>=3.10',
                      'mypy_extensions >= 0.4.3',
                      "tomli>=1.1.0; python_version<'3.11'",
                      ],
    ...
)
```

这意味着mypy总是依赖于mypy_extensions和type_extensions，而当系统中的Python的版本小于3.11时还依赖于tomli，小于3.8时还依赖于typed_ast。例子中的python_version < 'xxx'就是一个由单个环境标记形成的环境标记表达式。

表17-5列出了所有的环境标记。但要注意，环境标记的名称和值之间需要用PEP 440引入的用于版本说明符的运算符（即==、!=、>、>=、<、<=、~=和===）连接。环境标记表达式则是将若干环境标记用逻辑运算符and和or，以及圆括号连接起来形成，例如：

```
(python_version<'3.8' and python_version>'2.7') or os_name=='nt'
```

表17-5. 环境标记

名称	等价Python对象	典型值
os_name	os.name	'posix'
sys_platform	sys.platform	'darwin'
platform_machine	platform.machine()	'x86_64'
platform_python_implementation	platform.python_implementation()	'CPython'
platform_release	platform.release()	'20.6.0'
platform_system	platform.system()	'Darwin'
platform_version	platform.version	'Darwin Kernel Version 20.6.0: Tue Jun 21 20:50:28 PDT 2022; root:xnu-7195.141.32~1/RELEASE_X86_64'
python_full_version	platform.python_version()	'3.11.0b3'
implementation_name	sys.implementation.name	'cpython'
implementation_version*	sys.implementation.version	'3.11.0b3'
extra**	-	'test'

(* 准确的说是派生自**sys.implementation.version**。)

(** 由PEP 508的实现自行定义。)

extras_require用于说明分发包的每个额外特性所依赖的其他分发包。当安装该分发包时指定了哪个额外特性，就需要安装那些额外的分发包。sampleproject的setup.py中包括如下设置：

```
    setup(
        ...
        extras_require={
            'dev': ['check-manifest'],
            'test': ['coverage'],
        },
        ...
    )
```

这就是为什么sampleporject本身只依赖于peppercorn，但指定了额外特性test后还会依赖于coverage。extras_require被传入字典的每个键所对应的值都是依赖规格。

setup_requires也会被传入一个依赖规格列表，其与install_requires的区别在于，它指定的依赖关系仅在安装分发包时需要被满足，而在使用该分发包时却不需要考虑。需要注意的是，setup_requires和install_requires可以重复指定一个分发包，这意味着该分发包既在安装时被依赖，也在使用时被依赖。pip在安装时会检查setup_requires指定的依赖环境是否被满足，并会尝试下载所需的分发包，而在完成安装后会自动删除不再需要的分发包。然而目前setup_requires关键字已经不推荐使用了，因为pyproject.toml替代了它的功能。

一个分发包被安装后，最基本的使用方法是import其内的包或模块，然后使用它们定义的类和函数。当然我们也可以通过-m选项直接执行一个包或模块，而为了使这样的行为有意义，被执行的包或模块需要定义main()。这些已经在第6章和第7章讨论过了。下面讨论的“入口点（entry points）”则使得安装了一个分发包后，相当于给计算机系统安装了一些命令，就好像pip可以被直接当成shell命令来使用一样。

入口点是通过entry_points实现的，它被传入的字典可以有如下两个键：

- “console_scripts”：用于定义“控制台脚本（console scripts）”的入口点。
- “gui_scripts”：用于定义“GUI脚本（GUI scripts）”的入口点。

两种脚本的区别是，控制台脚本相当于一个shell命令，会在CLI界面下执行；而GUI脚本相当于一个GUI程序，需要启动一个窗口来执行。这两个键的值都是一个如下格式的字符串形成的列表：

```
'name = module:object[.attr]*'
```

其中name代表入口点的名字，module代表包或模块的名字，object代表该包或模块中定义的某个对象，attr则代表该对象的属性。注意attr部分是可选的，可以重复任意多次。必须满足的要求是：object自身是可调用对象（此时不需要attr），或者object.attr.....是可调用对象。每个这样格式的字符串都定义了一个入口点，其含义是执行命令name等价于调用module.object或module.abject.attr.....指向的可调用对象。

在mypy的setup.py中包含如下设置：

```
setup(
    ...
    entry_points={'console_scripts':
        ['mypy=mypy.__main__:console_entry',
         'stubgen=mypy.stubgen:main',
         'stubtest=mypy.stubtest:main',
         'dmypy=mypy.dmypy.client:console_entry',
         'mypyc=mypyc.__main__:main',
         ]},
    ...
)
```

这意味着安装了mypy分发包后，系统中相当于多了5个shell命令：mypy、stubgen、stubtest、dmypy和mypyc。在第15章已经多次使用了mypy。那么它是如何实现的呢？

假如你使用的是Unix或类Unix操作系统，请先通过如下命令行查看环境变量PATH的值：

```
$ echo $PATH
/Library/Frameworks/Python.framework/Versions/3.11/bin:/opt/local/bin:/opt/
local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/Library/
Apple/usr/bin:/usr/local/homebrew/bin
```

该结果说明在安装了Python 3.11后，在执行一个命令行时会先在/Library/Frameworks/Python.framework/Versions/3.11/bin目录下搜索相应可执行文件。然后请通过如下命令行查看该目录的内容：

```
$ ls /Library/Frameworks/Python.framework/Versions/3.11/bin
2to3               pip               python3-intel64
2to3-3.11          pip3              python3.11
dmypy              pip3.11           python3.11-config
idle3              pydoc3            python3.11-intel64
idle3.11           pydoc3.11         stubgen
mypy               python3           stubtest
mypyc              python3-config
```

可见该目录下有名为mypy、stubgen、stubtest、dmypy和mypyc的文件。最后通过如下命令行查看mypy的内容：

```
$ cat /Library/Frameworks/Python.framework/Versions/3.11/bin/mypy
#!/Library/Frameworks/Python.framework/Versions/3.11/bin/python3
# -*- coding: utf-8 -*-
import re
import sys
from mypy.__main__ import console_entry
if __name__ == '__main__':
    sys.argv[0] = re.sub(r'(-script\.pyw|\.exe)?$', '', sys.argv[0])
    sys.exit(console_entry())
```

可见这是一个可直接执行的Python脚本，执行它会调用mypy模块的console_entry()，与setup.py中的mypy入口点的定义一样。事实上，入口点的作用就是使得编译系统后端安装分发包时自动创建类似的可直接执行的Python脚本。

在sampleproject的setup.py中包含如下设置：

```
setup(
    ...
    entry_points={
        'console_scripts': [
            'sample=sample:main',
        ],
    },
    ...
)
```

然而在上一节我们将sampleproject安装在虚拟环境example_env下。这样的入口点仅在激活了相应虚拟环境的情况下才能使用，例如：

```
$ source example_env/bin/activate

(example_env)$ sample
Call your main application code here
```

注意在/Library/Frameworks/Python.framework/Versions/3.11/bin目录下并没有名为“sample”的可执行文件。请通过如下命令行检查PATH的值的变化：

```
(example_env)$ echo $PATH
/Users/wwy/example_env/bin:/Library/Frameworks/Python.framework/Versions/3.11/bin:/opt/local/bin:/opt/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/Library/Apple/usr/bin:/usr/local/homebrew/bin
```

这说明虚拟环境被激活后，相应目录下的bin子目录会被添加到PATH环境变量中，并具有最高优先级。接下来检查/Users/wwy/example_env/bin的内容：

```
$ ls /Users/wwy/example_env/bin/
Activate.ps1      activate.fish     coverage3   pip3.11     python3.11
activate          coverage          pip         python      sample
activate.csh      coverage-3.11     pip3        python3

(example_env)$ cat /Users/wwy/example_env/bin/sample
#!/Users/wwy/example_env/bin/python3.11
# -*- coding: utf-8 -*-
import re
import sys
from sample import main
if __name__ == '__main__':
    sys.argv[0] = re.sub(r'(-script\.pyw|\.exe)?$', '', sys.argv[0])
    sys.exit(main())
```

可以看出可直接执行的sample脚本是被创建于虚拟环境对应目录的bin子目录下的，而执行它将会导致sample模块的main()被调用，如setup.py中所设置。

由于编写启动窗口的Python脚本超出了本书的范围，所以本书不给出关于GUI脚本的入口点的例子。此外，入口点还可以用于为分发包指定插件，但本书也不详细讨论。感兴趣的读者请自行查阅setuptools的官方手册。

不论是控制台脚本还是GUI脚本，其内作为入口点的函数都不能有任何形式参数。在这些函数的函数体内应通过sys.argv取得命令行参数。

值得一提的是，distutils遗留下来的关键字scripts也可被用于指定入口点，使用方法是传入一个路径列表，指定的脚本都被视为入口点，例如：

```
setup(
    ...
    scripts=['scripts/foo', 'scripts/bar'],
    ...
)
```

这意味着在setup.py所在目录下应有一个子目录scripts，其内的文件foo和bar都是可直接执行的Python脚本，而这些脚本在安装时会被直接拷贝到相应的bin目录下。由于需要自己编写作为入口点的Python脚本，所以这种指定入口点的方式已经不被推荐。

zip_safe和eager_resources控制着该分发包是否能被安装为一个ZIP归档文件。如果zip_safe被传入True，则意味着允许将该分发布安装为ZIP归档。而eager_resources则列出一些文件，当该分发包被安装为ZIP归档时，必须将这些文件从中抽取出来单独安装。

至此我们已经介绍完了setup()所有重要的关键字。剩余的script_name、script_args、options、cmdclass和distclass用于控制setup.py的执行方式。如果你感兴趣请自行查阅setuptools的官方手册。

17-5. 现代打包技术

现在你已经知道了该如何编写setup.py。换句话说，你已经掌握了setuptools最古老的用法。然而这种用法存在很多问题，因此目前已经逐渐在被现代打包技术所替代。本节将讨论替代setup.py的打包技术。

在软件业的发展史中曾经出现过很多种打包技术。最初不同的编程语言倾向于提供自己的打包技术，使用格式独特的元数据文件，setup.py就是一个典型例子。然而很多大型项目需要多种编程语言相互配合，这些互不兼容的打包技术就带来了很多麻烦。考虑到打包的本质是为源代码或可直接使用的二进制文件补充元数据文件，从某种角度讲这些元数据文件也可被视为特殊的“配置文件（configuration files）”，而配置文件的格式已经逐渐趋同，所以现代打包技术都要求元数据文件采用某种配置文件格式。

常用的配置文件格式包括XML、JSON、YAML和TOML。目前所有的主流编程语言都选择了TOML作为打包时元数据文件的格式，这是因为XML既不够简洁又缺乏可读性，JSON简洁但可读性不足，YAML可读性高但不够简洁，而TOML同时具备简洁性和可读性。在讨论Python的现代打包技术之前，有必要先介绍TOML。

TOML是“Tom's Obvious Minimal Language”的递归缩写[27]，由汤姆·普雷斯顿·沃纳（Tom Preston-Werner）创造，而他也是Github的创始人。TOML是专门为编写配置文件而设计的一门编程语言，然而其语法极其简单。

概括地说，一个TOML文件中只可能有四种非空白行：

- 以“#”开头的注释。
- “key = value”形式的键值对。
- “[name]”形式的表头。
- “[[name]]”形式的表数组头。

TOML文件的MIME类型是“application/toml”，一般情况下会采用.toml作为后缀名。

TOML中的注释与Python中的注释格式完全相同。

TOML中的键值对形式上与Python中的赋值表达式相同，但等号左边的是键而非标识符。TOML中的键本质上都是字符串，所以可以用引号将它们括起来。如果键没有用引号括起来，则只能包含英文字母（区分大小写）、阿拉伯数字、下划线和连字符。此外，空串也是一个合法的键，虽然不建议使用它。键值对的等号右边自然是值，而TOML中的值可以是如下类型：字符串、数字、布尔值、时间、数组和内联表。下面依次讨论它们。

TOML中的字符串与Python中的字符串相似，但存在如下区别：

- TOML中的字符串只能采用UTF-8编码方式。
- TOML中的字符串也分为单行和多行，但在用"括起来的单行字符串和用"""括起来的多行字符串中可以用“\”标记转义序列；而在用'括起来的单行字符串和用'''括起来的多行字符串中“\”没有特殊含义。
- TOML只支持表17-6列出的转义序列（请与表2-1进行比较）。此外，在用"""括起来的多行字符串中，行末尾的“\”会使得该行与下一行被合并为一个逻辑行。

表17-6. TOML转义序列

转义序列	说明
\b	退格，即\010。
\t	水平制表，即\011。
\n	换行，即\012。
\f	换页，即\014。
\r	回车，即\015。
\"	双引号，即\042。
****	反斜杠，即\134。
\uhhhh	任意Unicode BMP字符，其中h代表一个十六进制数字，为字符的Unicode编码。
\Uhhhhhhhh	任意Unicode字符，其中h代表一个十六进制数字，为字符的Unicode编码。

下面是一些字符串值的例子：

```
name = 'Lily'

sex = "\u2640"

occupation = '''
Developer,
Architecture Designer and
Team leader.'''

introduction = """
Hello all. I live in Long Island.
I spend my leisure time diving, riding and hiking.
I like \u3059\u3057 very much."""
```

TOML中的数字与Python中的数字也很相似，但存在如下区别：

➤ TOML中的数字只支持整数和浮点数，不支持复数。
➤ TOML中的浮点数中的小数点两侧都必须有数字。
➤ TOML中的浮点数包含特殊值nan和inf，分别对应IEEE 754标准中的NaN和Inf，而它们还支持正负号。

下面是一些数字值的例子：

```
maximum = 1_000

height = 1.0e-3

"Positive Infinity" = +Inf
```

TOML中的布尔值是true和false，注意首字母不大写。下面是一些布尔值的例子：

```
Auto_Truncate = false
```

TOML中的时间符合RFC 3339定义的Internet时间戳格式[28]，简单来说完整的时间通过如下格式表示：

YYYY-MM-DDThh:mm:ss[.xxxxxx]{+|-}HH:μμ

其中YYYY代表年，MM代表月份，DD代表天，hh代表小时，mm代表分钟，ss代表秒，xxxxxxx代表微秒（可省略），HH代表时区中的小时数，μμ代表时区中的分钟数，而正号和负号分别表示早于/晚于UTC时间。如果用字母Z代替了整个时区部分，则表示UTC时间。如果省略了时区部分，则表示本地时间。如果省略了字母T及它后面的部分，则表示本地日

期。如果同时省略了时区部分和字母T及它前面的部分，则表示一天中的某个时刻（与具体日期和时区无关）。下面是一些时间值的例子：

```
accurate-time = 1993-09-26T17:04:59-03:00

utc-time = 1993-09-26T14:04:59Z

local-time = 1993-09-26T17:04:59

local-date = 1993-09-26

time-every-day = 17:04:59
```

TOML中的“数组（arrays）”与Python中的列表格式相同。下面是一些例子：

```
trivial = [1, 2, 3]

"PRIMARY COLORS" = [
    "red",
    "green",
    "blue",
]

mixed = [0, false, 1.0, "true"]

nested = [[0, 1], ['a', 'b']]
```

TOML中的“内联表（inline-tables）”类似于Python中的字典，但不使用冒号表示键值对，且最后一个键值对不能跟逗号。此外，内联表必须写成一行。下面是一个例子：

```
full_name = {first_name = "Alan", middle_name = "Mathison", last_name = "Turing"}
```

内联表必须写成一行，所以虽然理论上内联表可以嵌套，但我们很少这样做。

下面给出两个内联表和数组相互嵌套的例子。第一个例子将数组嵌入内联表：

```
points = {p1 = [0, 1], p2 = [-1, 0], p3 = [2, 7]}
```

第二个例子将内联表嵌入数组：

```
points = [{x = 0, y = 1},
          {x = -1, y = 0},
          {x = 2, y = 7}]
```

需要强调，内联表是“表（tables）”的一种简略形式，而“表数组（table arrays）”则是对上面将内联表嵌入数组的例子的扩展。TOML文件本质上就是一个大表，其内包含了若干相互嵌套的表和表数组。

要想表示一个表，需要先给出一个表头，然后跟若干键值对，而这些键值对会被视为属于该表。例如上面关于内联表的例子展开为表的形式应写为：

```
[full_name]
first_name = "Alan"
middle_name = "Mathison"
last_name = "Turing"
```

而上面将数组嵌入内联表的例子展开为表的形式则应写为：

```
[points]
p1 = [0, 1]
p2 = [-1, 0]
p3 = [2, 7]
```

表头后面也可以不跟任何键值对，这用于定义一个空表。

就像内联表一样，表也可以相互嵌套，而这种嵌套的实现方式比内联表简介：可以在表头中以“.”将多个键连接起来，以使得该表属于另一个表。请看下面的例子：

```
[board]

[board.person1]
nationality = "UK"
rank = 1

[board.person1.full_name]
first_name = "Alan"
middle_name = "Mathison"
last_name = "Turing"

[board.person2]
nationality = "USA"
rank = 1

[board.person2.full_name]
first_name = "John"
middle_name = "von"
last_name = "Neumann"
```

这会定义多个相互嵌套的表，其结构用Python字典可以表示为：

```
board = {
    "person1": {
```

```
            "nationality": "UK",
            "rank": 1,
            "full_name": {
                "first_name": "Alan",
                "middle_name": "Mathison",
                "last_name": "Turing"
            }
        },
        "person2": {
            "nationality": "US",
            "rank": 1,
            "full_name": {
                "first_name": "John",
                "middle_name": "von",
                "last_name": "Neumann"
            }
        }
    }
```

注意在处理上述TOML代码时，会先创建一个空的[board]表，然后再逐渐向其填入键值对。事实上，如果键值对的键包含“.”，也会隐式创建一些空表，例如直接执行如下键值对：

```
    sloth.lightning.carspeed = 60.0
```

创建的表的嵌套结构可以用Python字典表示为：

```
    sloth = {
        "lightning": {
            "carspeed": 60.0
        }
    }
```

要想表示一个表数组，则需要先给出一个表数组头，然后跟若干键值对；再给出一个相同的表数组头，同样跟若干键值对；……这样每个表数组头和后面的键值对都形成一个表，而所有使用相同表数组头的表按顺序组成了一个数组。例如上面将内联表嵌入数组的例子展开为表数组应写为：

```
    [[points]]
    x = 0
    y = 1

    [[points]]
    x = -1
    y = 0

    [[points]]
    x = 2
    y = 7
```

表数组头后面也可以不跟任何键值对，这用于让该表数组包含一个空表。

所有这些表、表数组以及独立的键值对（与任何表头或表数组头之间都存在空行）被视为属于该TOML文件对应的大表（该表是抽象的）。

至此我们已经介绍完了TOML的语法，接下来让我们讨论TOML如何被应用于Python的现代打包技术。

首先需要解释setup.py带来了哪些不便。setup.py是一个Python脚本，必须被执行才能获得其内的元数据。在直接使用distutils打包的时候，这不会造成任何问题，因为不论是制作还是安装Python的分发包，前提都是已经安装了某种Python解释器，而作为标准库的一部分，只要安装了Python解释器，distutils就是可获得的。然而现在的setup.py需要导入setuptools模块，而setuptools并不是标准库的一部分，这给开发通用的Python打包工具带来困难：它必然依赖于setuptools，相当于重复安装了打包工具。作为一种临时解决方案，目前安装CPython时都会自动安装setuptools，然而这毕竟只是权宜之计。

另外，我们已经知道setup()的setup_requires关键字被用来指定安装时依赖。然而不执行setup()，就无法解读setup_requires指定了哪些依赖；而不满足这些依赖，setup()又难以执行完成。这极大提高了setuptools的设计难度，直到今天都存在一些无法应对的场景。

为了解决上述问题，PEP 517和PEP 518提出了将编译系统分为前端和后端的理念。按照这个模型，setuptools原本既是前端也是后端，以后将只作为后端使用（不建议再直接执行setup.py），而以build（将在下面讨论）作为前端。其他打包工具也必须分为前端和后端两部分。而这些工具的前端和后端并不是绑定的，一个工具的前端可以兼容其他工具的后端，而像build这样的通用前端可以支持所有后端。另外，pip本质上也是一个通用前端。

遵循PEP 517的分发包需要提供pyproject.toml来说明自己使用哪个编译系统后端，且在安装时依赖于哪些其他的分发包。从后缀名就可以看出，pyproject.toml是一个TOML文件，其核心是build-system表，其内可以有如下键的键值对：

➤ requires：值为一个数组，数组元素都是依赖规格，用于代替setup()的setup_requires关键字。
➤ build-backend：值为一个字符串，格式与代表入口点的字符串相同，用于指定使用的编译系统后端。也可以取特殊值“local_backend”，以表明编译系统后端由分发包自身携带，而由非某个专门的打包工具提供。
➤ backend-path：值为一个数组，数组元素都是指向目录的路径，这些路径在安装该分发包时会被添加到sys.path中，以使得在这些目录中搜索编译系统后端。仅当build-backend的值是“local_backend”时才有意义。

sampleproject中的pyproject.toml内容如下：

```
[build-system]
# These are the assumed default build requirements from pip:
# https://pip.pypa.io/en/stable/reference/pip/#pep-517-and-518-support
requires = ["setuptools>=40.8.0", "wheel"]
build-backend = "setuptools.build_meta"
```

这表明安装sampleproject时依赖于模块setuptools（版本不低于40.8.0）和wheel，使用的编译系统后端是setuptools模块的build_meta子模块。mypy提供的pyproject.toml具有类似内容。boltons没有提供pyproject.toml。如果分发包中没有pyproject.toml，或者提供了该文件但其内没有build-system表，则编译系统前端会以setuptool.build_meta为默认编译系统后端。

下面给出一些指定其他编译系统后端的pyproject.toml的例子。如下build-system表指定安装时依赖于flit_core模块（版本不低于3.2），以该模块的flit_core.buildapi子模块为编译系统后端：

```
[build-system]
requires = ["flit_core>=3.2"]
build-backend = "flit_core.buildapi"
```

而如下build-system表指定安装时依赖于hatchling模块，以该模块的build子模块为编译系统后端：

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

事实上flit_core是由flit分发包提供的，而flit模块是与之对应的编译系统前端；hatchling是由hatch分发包提供的，而hatch模块是与之对应的编译系统前端。flit和hatch都是由PyPA开发和维护的项目。

在有了pyproject.toml之后，我们就可以按自己的喜好选择打包工具，不再必须依赖于setuptools了。理想的情况是，PyPI中的所有分发包都包含pyproject.toml，通过它指定编译系统后端和依赖，而其余元数据则通过其他文件提供。然而除了setuptools之外，其他打包工具都不会使用一个可直接执行的Python脚本来提供元数据。为了保持与其他打包工具的风格一致，setuptools从版本30.3.0开始支持setup.cfg。

setup.cfg并不是一个TOML文件，但语法与TOML很像。具体来说，setup.cfg的语法与TOML存在如下不同：

➤ 键值对中的键必须是一个字符串。如果键被省略，则被解读为以空字符串为键。
➤ 键值对中的值的类型可以是：
str：字符串。不使用引号，只能写成一行。
bool：布尔值。true还可以写为True、yes和1，false还可以被写为False、no和0。
list-comma：用逗号分隔的一行值，或者省略逗号在键下面写成多行（被称为“悬挂式”）。
list-semi：用分号分隔的一行值，或者省略分号在键下面写成多行（被称为“悬挂式”）。
dict：用逗号分隔的一行键值对，或者省略逗号在键下面写成多行（被称为“悬挂式”）。
TOML中的数字和时间值都应视为str；数组可被视为list-comma；内联表则被视为dict。
➤ 没有表的概念，对应概念是section。setuptools自身支持两个section：
[metadata]：其内的名值对与附录 VI中表VI-1列出的关键字相对应。
[options]：其内的名值对与附录 VI中表VI-2列出的关键字相对应，但不包括ext_modules和ext_pacakge。
➤ 提供了特殊指令attr:来指定一个Python对象，file:来指定若干文件，find:来指定find_packages()函数。

编写setup.cfg的过程其实就是将setup.py中的参数赋值转换为键值对的过程。但由于参数赋值可以是任意Python对象，而setup.cfg只支持很少几种类型，所以转换时需要做一些处理。附录 VII总结了setup.cfg中对应setup.py的关键字的键所支持的类型。注意在附录 VII的表中，如果某行的值类型为“section”，则表示该键值对需用一个单独的section来描述；如果某行的值类型为“list-semi”，且包含特殊指令，则表示该键值对必须写成悬挂式。

下面是将sampleproject的setup.py改写为setup.cfg的结果：

```
[metadata]
name = sampleproject
version = 2.0.0
description = A sample Python project
long_description = file: README.md
long_description_content_type = text/markdown
author = A. Random Developer
author_email = author@example.com
url = https://github.com/pypa/sampleproject
project_urls =
    Bug Reports = https://github.com/pypa/sampleproject/issues
    Funding = https://donate.pypi.org
    Say Thanks! = http://saythanks.io/to/example
    Source = https://github.com/pypa/sampleproject/
classifiers =
    Development Status :: 3 - Alpha
    Intended Audience :: Developers
    Topic :: Software Development :: Build Tools
    License :: OSI Approved :: MIT License
    Programming Language :: Python :: 3
    Programming Language :: Python :: 3.5
    Programming Language :: Python :: 3.6
    Programming Language :: Python :: 3.7
    Programming Language :: Python :: 3.8
```

```
        Programming Language :: Python :: 3 :: Only
    keywords =
        sample setuptools development


    [options]
    package_dir =
        = src
    packages = find:
    python_requires = >=3.5, <4
    install_requires =
        peppercorn


    [options.packages.find]
    where = src


    [options.extras_require]
    dev =
        check-manifest
    test =
        coverage


    [options.package_data]
    sample =
        package_data.dat


    [options.entry_points]
    console_scripts =
        sample=sample:main
```

注意在这个例子中所有类型为list-comma、list-semi或dict的值都采用了悬挂式写法，而python_requires的值类型只能是src，所以其内的逗号会被视为字符串的一部分。file:指令的后面应跟一个list-comma类型的值，但这会导致被列出的文件的内容被依次拼接起来，因此最后等同于一个src类型的值。当通过find:调用了find_packages()，或者通过attr:调用了分发包自身包含的某个可调用对象时，都可以通过一个专门的section为其传入参数。

事实上，只要分发包中不包含扩展模块，那么setup.py就可以完全用setup.cfg替代。但为了兼容不支持PEP 517的编译系统前端，此时最好依然保留setup.py，但其内容只是：

```
from setuptools import setup

setup()
```

除了提供setup.py中包含的元数据外，setup.cfg还可以包含提供给其他工具使用的元数据。例如在sampleproject、mypy和boltons的setup.cfg中都包含如下section：

```
[egg_info]
tag_build =
tag_date = 0
```

这其实是表示在制作egg包时给tag_date参数传入0。此外，mypy和boltons还包含一些除[metadata]、[options]和[egg_info]的section。当一个工具在处理setup.cfg的过程中遇到了不识别section或键值对，可以直接跳过，不需要报错。

然而需要强调的是，setup.cfg以及与之类似的提供元数据的文件都只能被某种特定编译系统后端处理。然而在有很多情况下我们期望收集一个仓库中的所有分发包的元数据，并将它们以统一的方式展示和分析，而这些分发包使用不同格式的文件来提供元数据会为实现这一目标带来极大阻碍。为解决这一问题，PEP 621于2020年被通过，引入了“核心元数据（core metadata）”的概念，并定义了将它们储存在pyproject.toml中的方法。现在我们在创建分发包时理应遵守PEP 621的规范。

PEP 621定义了表17-7中列出的键值对来提供核心元数据，这些键值对必须位于[project]表内。关于这些键值对的详细说明除了PEP 621外还可以查阅Python打包用户导引中的相关文档[29]。不在表17-7范围内的元数据都不属于核心元数据，仅对特定打包工具有意义。对于setuptools来说，下列关键字对应非核心元数据：

- 帮助搜索分发包：platforms、provides和obsoletes。
- 描述分发包的内容：package_dir、packages、py_modules、ext_modules、ext_pacakge、package_data、include_pacakge_data和exclude_package_data。
- 直接安装为ZIP归档文件：zip_safe、eager_resources。

当我们将核心元数据存放在pyproject.toml中时，不应在setup.cfg中重复提供它们，以避免发生冲突。然而由于在pyproject.toml中包含核心元数据不是强制性的，所以我们也可以将部分或全部核心元数据放到setup.cfg中，在这种情况下应在[project]表中通过键dynamic列出所有被省略的关键字，该键的值是一个字符串数组。

表17-7. 核心元数据

键	说明	值的类型	对应关键字
name	符合PEP 503的分发包名称。	字符串	name
version	符合PEP 440的分发包版本。	字符串	version
description	分发包的总结性描述，只有一行。	字符串	description
readme	分发包的详细描述。如果是字符串，则代表文件路径，根据后缀名判断MIME类型： .md：text/markdown .rst：text/x-rst 如果是表，则包含如下两个键之一： file：文件路径。 text：完整内容。 此外还包含如下键： content-type：MIME类型。	字符串或表	long_description、long_description_content_type

键	说明	值的类型	对应关键字
authors	每个表都包含如下两个键之一或全部： name：作者的名字。 email：作者的电子邮箱。	表数组	author、author_email
maintainers	每个表都包含如下两个键之一或全部： name：维护者的名字。 email：维护者的电子邮箱。	表数组	maintainer、maintainer_email
license	该表包括如下键之一： file：文件路径。 text：完整内容。	表	license、license_files
urls	该表包含的键值对等同于project_urls中的键值对。	表	project_urls、url、download_url
classifiers	每个字符串都是符合PEP 301的分类器。	字符串数组	classifiers
keywords	每个字符串都是关键字。	字符串数组	keywords
requires_python	该字符串是一个版本说明符序列。	字符串	python_requires
dependencies	每个字符串都是符合PEP 508的依赖规格。	字符串数组	install_requires
optional-dependencies	表的每个键都是代表额外特性的字符串，值则是一个字符串数组，其中每个字符串都是一个符合PEP 508的依赖规格。	表	extras_require
scripts	表的每个键都是入口点名，值都是基于控制台脚本的入口点。	表	entry_points、scripts
gui-scripts	表的每个键都是入口点名，值都是基于GUI脚本的入口点。	表	
entry-points	每个表都对应某个类型的入口点，例如可以用plugins表来表示实现插件的入口点。表结构与scripts或gui-scripts相同。	表数组	

下面是将sampleproject的setup.py中的核心元数据储存到pyproject.toml中后得到的结果：

```
[build-system]
requires = ["setuptools>=40.8.0", "wheel"]
build-backend = "setuptools.build_meta"


[project]
name = "sampleproject"
version = "2.0.0"
description = "A sample Python project"
readme = "README.md"
license.file = "LICENSE.txt"
classifiers = [
    "Development Status :: 3 - Alpha",
    "Intended Audience :: Developers",
    "Topic :: Software Development :: Build Tools",
    "License :: OSI Approved :: MIT License",
    "Programming Language :: Python :: 3",
    "Programming Language :: Python :: 3.5",
    "Programming Language :: Python :: 3.6",
    "Programming Language :: Python :: 3.7",
    "Programming Language :: Python :: 3.8",
```

```
        "Programming Language :: Python :: 3 :: Only",
    ]
    keywords = ["sample setuptools development"]
    requires_python = ">=3.5, <4"
    dependencies = ["peppercorn"]


    [[project.authors]]
    name = "A. Random Developer"
    email = "author@example.com"


    [project.urls]
    Home = "https://github.com/pypa/sampleproject"
    "Bug Reports" = "https://github.com/pypa/sampleproject/issues"
    Funding = "https://donate.pypi.org"
    "Say Thanks!" = "http://saythanks.io/to/example"
    Source = "https://github.com/pypa/sampleproject/"


    [project.optional-dependencies]
    dev = ["check-manifest"]
    test = ["coverage"]


    [project.scripts]
    sample = "sample:main"
```

注意license.file键是针对取默认值的license_files关键字而添加的。在去掉核心元数据后，setup.cfg的内容将只剩下：

```
    [options]
    package_dir =
        = src
    packages = find:


    [options.packages.find]
    where = src


    [options.package_data]
    sample =
        package_data.dat
```

pyproject.toml也可以储存只对特定打包工具有意义的元数据。例如在mypy所依赖的typing_extension中，pyproject.toml还包含如下表：

```
    [tool.flit.sdist]
    include = ["CHANGELOG.md", "README.md", "*/test*.py"]
    exclude = []
```

该表仅对flit有意义，用于指定该分发包的结构。事实上typing_extension使用的打包工具是flit，但没有专门为flit提供元数据的文件，所有元数据都储存在pyproject.toml中。

17-6. 制作和发布分发包

至此，我们已经能够看懂任何从PyPI下载的源代码包中的元数据。是时候尝试打包我们自己的Python项目了。

首先创建一个虚拟环境test_env：

```
$ python3.11 -m venv test_env
```

然后将第7章中创建的包demo拷贝到该虚拟环境下，再给该包增加脚本self_explain.py，其内容为：

```
from os import path

#该函数读取intro.txt的内容并显示。
def self_explain():
    """Explain what this pacakge is."""

    here = path.abspath(path.dirname(__file__))

    with open(path.join(here, "resource/intro.txt")) as fd:
        print(fd.read())
```

最后给demo包增加子目录resource，在resource目录下增加文本文件intro.txt，其内容为：

```
Demo是一个用于展示规范格式的包。

它提供了函数factorial_sequence()来生成阶乘序列，该函数可被其他模块导入。

通过命令“demo”可以直接执行该包。

而通过命令“demo-intro”可以显示本介绍文档。
```

这样就凑齐了将被制作的分发包中的脚本和数据文件。接下来添加提供元数据的文件。

在这个例子中我们选择setuptools作为编译系统后端，因此需要在test_env目录下添加pyproject.toml、setup.cfg和setup.py。先编写pyproject.toml来提供核心元数据，其内容为：

```
#编译系统
[build-system]
requires = ["setuptools>=58.1.0"]
build-backend = "setuptools.build_meta"
```

```
#核心元数据
[project]
name = "wwy_demo"
version = "1.0.0"
description = "Demonstration of how to make a distribution package."
readme = "README.md"
license.text = "MIT LICENSE"
classifiers = [
    "Development Status :: 3 - Alpha",
    "Environment :: Console",
    "Intended Audience :: Developers",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
    "Programming Language :: Python :: 3",
    "Programming Language :: Python :: 3 :: Only",
    "Programming Language :: Python :: 3.11",
    "Topic :: Software Development",
]
keywords = ["packaging", "factorial"]
requires-python = ">=3.11, <4"


[[project.authors]]
name = "wwy"
email = "james_wwy@yahoo.com"


#入口点
[project.scripts]
demo = "demo.main:main"
demo-intro = "demo.self_explain:self_explain"
```

然后编写setup.cfg：

```
#包
[options]
packages = demo


#数据文件
[options.package_data]
demo =
    *.txt
```

最后提供一个空的setup.py：

```
from setuptools import setup

setup()
```

下面编写在pyproject.toml中指定的README.md，它也要放在test_env目录下。注意PyPI中的分发包的详细描述遵循GitHub的格式，而README.md和README.rst都有自己的语法，本书不详细讨论，请读者自行查阅GitHub中的相关文档[30][31]。在这个例子中README.md的内容为：

```
      # Demonstration of Packaging

      ## Overview

      The `demo` module show how to package your Python project. Its contents are
all simple.


      ## Included items

      function `facrotial_sequence()`

      commands `demo` and `demo-intro`


      ## Usage

      To use factorial_sequence(), add `from demo import *` into your scripts.

      To run demo, type `demo` in the command-line interface.

      To see introduction, type `demo-info` in the command-line interface.
```

注意在上述pyproject.toml中，通过license.text给出了“MIT LICENSE”作为许可证，而没有通过license.file指定以LICENSE文件作为许可证，这是因为我实测发现存在如下bug：当以setuptools为后端时，给pyproject.toml中的license提供了大量文本，则在执行“pip show”命令时，会将整个文本都显示出来。换句话说，setuptools会认为pyproject.toml中的license等价于license关键字，而非license-files关键字。然而我们依然有必要通过某个能被license-files关键字的默认值识别的文件来提供完整的许可证。在这个例子中我们选择使用MIT许可证，并通过test_env目录下的LICENSE文件来存放其完整内容：

```
      MIT License

      Copyright (c) [year] [fullname]

      Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"), to
deal in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is furnished
to do so, subject to the following conditions:

      The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

      THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN
AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

当不确定选择哪个许可证时，这个网站可以帮助你：https://choosealicense.com/。

至此，该分发包已经基本完整了。但请注意，该分发包是放在test_env虚拟环境下的，这是有意模拟开发环境。test_env目录下有虚拟环境自带的目录和文件，例如pyvenv.cfg。当我们对demo中的Python脚本进行了调试和分析后，该目录下会自动生成__pycache__子目录，用于储存相应的.pyc文件。上述文件是没有必要加入分发包的，但它们已经与构成分发包的文件混在了一起。如果使用的编译系统后端是setuptools，就可以用MANIFEST.in文件来明确该分发包的构成，它同样被放在test_env目录下。

MANIFEST.in也有自己的语法[32]。概括地说，MANIFEST.in是由表17-8列出的命令构成的序列。此外，MANIFEST.in中以“#”开头的行也被视为注释。MANIFEST.in的处理过程相当于往一个清单中增减文件，在未执行任何命令前默认就会将下列文件放入清单：

- MANIFEST.in自身。
- README、README.txt、README.rst和README.md（从setuptools 36.4.0开始）。
- setup.py、setup.cfg和pyproject.toml（从setup tools 43.0.0开始）。
- pyproject.toml所在目录的test子目录下文件名匹配test*.py的文件。
- 通过packages关键字指定的包中的所有.py文件，以及通过py_modules关键字指定的所有.py文件。
- 为创建通过ext_modules关键字指定的扩展模块所涉及的源代码文件（例如.c文件）。
- 通过package_data关键字和exclude_package_data关键字共同指定的数据文件。
- 通过license_files关键字指定的许可证文件。

此后每执行一条命令都可能增加或减少该清单中的文件。在得到最终的清单后，如果一个文件被添加到分发包，则它所在的目录、父目录、……直到pyproject.toml所在目录（即项目根目录）形成的子目录树都会自动被添加到分发包。

表17-8. MANIFEST.in的命令

命令	说明
include *pat* ...	如果一个文件相对于项目根目录的路径匹配指定的模式，则将该文件添加到清单中。
exclude *pat* ...	如果一个文件相对于项目根目录的路径匹配指定的模式，则从清单中删除该文件。
recursive-include *dir_pat pat* ...	如果一个文件相对于目录名匹配dir_pat的目录的路径匹配指定的模式，则将该文件添加到清单中。
recursive-exclude *dir_pat pat* ...	如果一个文件相对于目录名匹配dir_pat的目录的路径匹配指定的模式，则从清单中删除该文件。
global-include *pat* ...	如果一个文件相对于分发包目录树的任何一个目录的路径匹配指定的模式，则将该文件添加到清单中。
global-exclude *pat* ...	如果一个文件相对于分发包目录树的任何一个目录的路径匹配指定的模式，则从清单中删除该文件。

命令	说明
graft *dir_pat*	将目录名匹配dir_pat的目录下的所有文件添加到清单中。
prune *dir_pat*	将目录名匹配dir_pat的目录下的所有文件从清单中删除。

在这个例子中，MANIFEST.in的内容为：

```
# Include pyproject.toml
include pyproject.toml

# Include README
include *.md

# Include license
include LICENSE

# Include data files
recursive-include demo/resource *.txt
```

当然这些文件默认也会被添加的清单中，这里只是为了说明怎么使用表17-8中的命令。

当使用的编译系统后端不是setuptools时，MANIFEST.in将不起作用。这种情况下打包工具通常根据核心元数据和专属于编译系统后端的元数据文件判断分发包由哪些文件构成。

至此我们已经做好了打包的准备，可以选择某款“包生成器（package builders）”进行打包了。注意编译系统前端不一定是包生成器（例如pip），但包生成器一定是编译系统前端。下面用PyPA提供的通用包生成器build完成打包。

首先安装build：

```
$ pip install --upgrade build
```

然后将工作目录设置到test_env，通过如下命令行同时创建源代码分发包和二进制分发包：

```
$ cd test_env/

$ python3 -m build
```

这样在test_env目录下会多出两个目录：wwy_demo.egg-info和dist，前者存放的是egg包的元数据（但并不是完整的egg包），后者则存放有dist包wwy_demo-1.0.0.tar.gz和轮子wwy_demo-1.0.0-py3-none-any.whl。build在创建轮子时需要先创建egg包，然后再将其转化为轮子，所以会出现wwy_demo.egg-info目录和其内的文件，但在创建好轮子后，该目录就已经没有用了。

build不能以shell命令的方式使用，只能通过-m选项被执行，其语法为：

```
python -m build [options] [srcdir]
python -m build {--help|-h}
python -m build {--version|-V}
```

其中srcdir指向项目根目录（pyproject.toml所在目录），如果被省略则取当前工作目录。表17-9则列出了build支持的选项。

表17-9. build的选项

选项	说明
--sdist, -s	创建dist包。
--wheel, -w	创建轮子。
--outdir *dir*, -o *dir*	将分发包放在dir目录下。
--skip-denpendency-check, -x	不检查安装时依赖是否被满足。
--no-isolation, -n	不通过被隔离的虚拟环境创建分发包。
--config-setting *config*, -C *config*	临时更改配置项设置。

表17-9列出的选项中最重要的是--sdist和--wheel，分别表示创建dist包和创建轮子。此外，--outdir选项指定将创建的dist包和/或轮子放到哪个目录下，如果省略则默认在项目根目录下创建dist子目录来储存它们。但不论是否给出了--outdir选项，都会在项目根目录下创建*.egg-info子目录。build默认会创建一个隔离的虚拟环境（忽略所有环境变量）来创建分发包，如果需要安装依赖则安装到该虚拟环境下。关于build更详细的说明请查阅其官方手册[33]。

需要强调的是，当既没有添加--sdist又没有添加--wheel时，build的默认行为是先创建dist包，再基于该dist包创建轮子；而只要添加了--wheel，则轮子将直接基于项目代码创建，哪怕同时添加了--sdist。一般而言，建议同时创建dist包和轮子，这样能确保以后从该dist包创建的轮子与本次创建的轮子保持一致。在创建轮子时，build会自动判断该轮子的兼容性，并设置最合适的兼容性标签。由于我们的分发包中不涉及扩展模块，所以该轮子是跨平台的，故ABI标签为none，平台标签为any。当涉及扩展模块时，必须在所有主流平台上分别创建轮子才能完美创建该分发包。

下面让我们来观察dist包和轮子的结构。从后缀名.tar.gz可以看出，dist包是一个通过tar归档然后通过gzip压缩的压缩归档文件。请通过如下命令行将其解压（假设你使用的是Unix或类Unix操作系统，下同）：

```
$ cd dist
$ tar -xzf wwy_demo-1.0.0.tar.gz
```

这样在test_env/dist目录下会出现wwy_demo-1.0.0子目录，其内除了包含MANIFEST.in指定的所有文件之外，还多出了wwy_demo.egg-info子目录和PKG-INFO文件。我们已经知道了前者和egg包有关，而后者则以标准的pkginfo格式提供元数据，以使该分发布能被更多标准化的工具使用。

轮子的后缀名虽然是.whl，但它其实本质上是一个ZIP归档文件。请通过如下命令行将其解压：

```
$ unzip wwy_demo-1.0.0-py3-none-any.whl
```

这样在test_env/dist目录下会出现demo子目录和wwy_demo-1.0.0.dist-info目录，前者就是wwy_demo分发包所包含的demo包，后者则用于存放轮子的元数据，包括下列文件：

- WHEEL：记录该轮子被创建的方式和兼容性。
- RECORD：记录组成该轮子的文件，以及哈希值。
- LICENSE：许可证文件。
- METADATA：元数据文件，综合了pyproject.toml和README.md。
- top_level.txt：该轮子提供了哪些包和独立脚本。
- entry_points.txt：该轮子提供了哪些入口点。

可见虽然轮子的元数据与dist包的元数据格式不同，但内容本质上是相同的。

下面切换到example_env虚拟环境，测试wwy_demo分发包是否能正确安装。首先切换工作目录并激活example_env：

```
$ cd
$ source example_env/bin/activate
```

然后安装wwy_demo的dist包：

```
(example_env)$ pip install test_env/dist/wwy_demo-1.0.0.tar.gz

(example_env)$ pip show wwy_demo
Name: wwy-demo
Version: 1.0.0
Summary: Demonstration of how to make a distribution package.
Home-page:
Author:
Author-email: wwy <james_wwy@yahoo.com>
License: MIT License
```

```
Location: /Users/wwy/example_env/lib/python3.11/site-packages
Requires:
Required-by:
```

接下来是测试wwy_demo能否正常使用：

```
(example_env)$ demo
这是一个生成阶乘序列的工具。输入“quit”退出。

请输入一个正整数: 3
[1, 2, 6]
请输入一个正整数: quit


(example_env)$ demo-intro
Demo是一个用于展示规范格式的包。

它提供了函数factorial_sequence()来生成阶乘序列，该函数可被其他模块导入。

通过命令“demo”可以直接执行该包。

而通过命令“demo-intro”可以显示本介绍文档。

(example_env)$ python3
>>> from demo import *
>>> factorial_sequence(4)
[1, 2, 6, 24]
>>> ^D
```

最后卸载wwy_demo包：

```
(example_env)$ pip uninstall wwy_demo

(example_env)$ deactivate
```

请重复上述过程来验证wwy_demo的轮子也可以被本地安装。

至此，我们已经可以通过U盘或电子邮件与他人分享我们制作的分发包了。如果你还想将该分发包上传到PyPI，则需要申请一个PyPI账户，并使用twine来完成上传。然而需要强调，直接向PyPI上传没什么实用价值的分发包是非常不礼貌的，为了帮助新手学习如何使用PyPI，PyPA非常贴心地提供了TestPyPI，其网址是https://test.pypi.org/。TestPyPI是一个PyPI的仿制品，维护着独立的测试用仓库和用户账户系统，但界面与PyPI保持一致。下面的讨论将以TestPyPI为例子，在你熟悉了这些技巧以后可以方便地切换到PyPI。

首先你需要在TestPyPI上注册一个用户账户，这可以直接在https://test.pypi.org/上完成，本书不详细讨论。但要注意，TestPyPI会周期性地清理仓库和用户账户系统，所以你上传的分发包可能会突然消失，用户账户也可能突然不能登录。我申请的账户用户名为wwy，这就是为什么将分发包命名为wwy_demo的原因，这样能够避免与其他用户的项目发生名字冲突。当然，正式的PyPI项目不会这样命名，只要确保不发生冲突即可。

接下来安装twine：

```
$ pip install --upgrade twine
```

twine可以被当成shell命令使用，有三个子命令：register、check和upload。register子命令用于在上传分发包前向指定仓库注册一个名称，但PyPI并不需要这一步骤。check子命令用于检查分发包的README.md或README.rst是否能在PyPI上正常显示。upload子命令用于上传分发包。本书不讨论register，只讨论check和upload。关于twine的详细信息请查阅其官方手册[34]。

check子命令的语法为：

```
twine check [--strict] dist ...
```

其中dist代表分发包的路径，可以包含通配符；--strict选项的作用是将警告转换为异常。

下面分别检查我们制作的dist包和轮子：

```
$ rm -Rf test_env/dist/demo test_env/dist/wwy_demo-1.0.0 test_env/dist/
wwy_demo-1.0.0.dist-info/

$ twine check test_env/dist/*
Checking test_env/dist/wwy_demo-1.0.0-py3-none-any.whl: PASSED
Checking test_env/dist/wwy_demo-1.0.0.tar.gz: PASSED
```

该结果说明我们制作的分发包的README.md格式符合要求。

upload子命令的语法为：

```
twine upload [options] dist ...
```

其中dist含义同上，而表17-10则列出了此时可使用的选项。下面只讨论较重要的选项。

表17-10. twine upload的选项

选项	说明
-r *repo*, --repository *repo*	上传到名为repo的仓库。
--repository-url *repo_url*	上传到URL为repo_url的仓库。
-u *user*, --username *user*	登录用的用户名是user。

选项	说明
-p *pswd*, --password *pswd*	登录用的口令是pswd。
--non-interactive	如果没有提供用户名和口令，则直接报错，不弹出交互式shell。
--config-file *file*	指定当成.pypirc使用的文件。
--cert *path*	指定用于建立SSL/TLS连接的服务器端证书。
--client-cert *path*	指定用于建立SSL/TLS连接的客户端证书。
-s, --sign	对分发包进行GPG签名。
--sign-with *gpg_program*	指定用来对分发包进行GPG签名的程序。
-i *id*, --identity *id*	指定用来对分发包进行GPG签名的ID。
-c *str*, --comment *str*	将字符串str作为该分发包的注释。
--skip-existing	如果仓库中已有该分发包则跳过。仅对PyPI有效。
--verbose	执行过程中输出更多信息。
--disable-progress-bar	执行过程中不显示进度条。

默认情况下，twine会将分发包上传到PyPI。如果要上传到其他仓库，则或者通过-r选项提供该仓库的名称（例如-r testpypi），或者通过--repository-url选项提供该仓库的URL（例如--repository-url https://test.pypi.org/legacy）。

默认情况下，twine会弹出一个交互式shell，要求我们输入关于指定仓库的用户名和口令。我们也可以通过-u选项提供用户名，通过-p选项提供口令，但这会使口令以明文出现，存在安全隐患。若添加了--non-interactive选项，则在任何情况下都不会弹出交互式shell，无法获得用户名和口令会直接报错。

如果你觉得每次都通过选项指定仓库，并提供用户名和口令太麻烦，也可以在家目录下创建一个名为.pypirc的文件，将上述信息保存在该文件中。如果你想使用其他文件名的文件来储存这些信息，则每次都需要通过--config-file选项指定使用该文件。而为了避免泄露用户口令，你可能需要使用keyring。这些技巧请查询Python打包用户导引[35]。

下面将我们制作的分发包上传到TestPyPI：

```
$ twine upload -r testpypi test_env/dist/*
Uploading distributions to https://test.pypi.org/legacy/
Enter your username: wwy
Enter your password:
Uploading wwy_demo-1.0.0-py3-none-any.whl
100% ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10.8/10.8 kB • 00:02 • 17.6 MB/s
Uploading wwy_demo-1.0.0.tar.gz
100% ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9.3/9.3 kB • 00:00 • 5.6 MB/s

View at:
https://test.pypi.org/project/wwy_demo/1.0.0/
```

等一小段时间，就可以在https://test.pypi.org/project/wwy_demo/1.0.0/上查看到wwy_demo项目了。项目主页会在中间完整显示README.md的内容，而pyproject.toml包含的核心元数据则会在左边栏显示。

不论是PyPI还是TestPyPI，都不允许使用重复的文件名，这意味着一个分发包被上传后，哪怕将分发包乃至整个项目完全删除，都不能重复上传该分发包。对分发包的任何修改都导致必须使用一个新的版本号。这意味着如果发现元数据存在错误将很难处理，所以在上传之前最好通过pip在本地安装并检查核心元数据是否完全正确，然后用twine的check子命令确保README.md或README.rst不存在语法错误。

在将分发包上传到TestPyPI后，就可以远程下载并安装它了：

```
$ source example_env/bin/activate

(example_env)$ pip install -i https://test.pypi.org/simple/ wwy_demo
Looking in indexes: https://test.pypi.org/simple/
Collecting wwy_demo
  Downloading https://test-files.pythonhosted.org/packages/57/4a/
d20e16d44edeb79d16882c5fa33c2a58f976b754233937487ab1a0388695/wwy_demo-1.0.0-py3-
none-any.whl (4.9 kB)
Installing collected packages: wwy_demo
Successfully installed wwy_demo-1.0.0

(example_env)$ deactivate
```

注意这里通过-i选项指定了使用TestPyPI而非PyPI。请自行验证wwy_demo安装成功。

至此我们已经讨论完了关于由PyPA维护的Python生态系统的所有必须掌握的知识。然而随着你对这个生态系统越来越熟悉，你会发现更多的有用技巧。下面举几个例子。

制作分发包时将所有的.py文件都替换成.pyc文件，然后通过MANIFEST.in指定分发包需要包含.pyc文件，就构成了无源文件发型版。对于上面的例子来说，只需要将demo目录下的.py文件全部删除，然后删除__pycache__子目录下的.pyc文件的文件名中的“.cpython-311”部分，最后在MANIFEST.in中添加如下命令：

```
# Include .pyc Files
recursive-include demo/__pycache__ *.pyc
```

就能制作出无源文件发型版。然而需要强调的是，由于Python的字节码能够被反编译成源代码，所以无源文件发型版并不能为代码保护提供任何安全性。换句话说，不涉及扩展模块的Python项目必然是以开源的形式发布的，只能通过许可证从法律上来限制用户对其的使用方式。

在开发大型Python项目的过程中，不同小组通常各自独立开发若干分发包，而这些分发包相互依赖，小组A需要将小组B开发的分发包安装到自己的虚拟环境中来测试，反之亦然。此

时每做一点修改都创建一个分发包会浪费大量时间，因此需要使用pip的-e选项直接从本地目录或远程版本控制系统进行安装，跳过创建分发包的步骤。

很多Python项目都是托管在GitHub上的，而GitHub提供了“GitHub Actions CI/CD”接口，可以为事件注册自动执行的操作，使得我们能够实现自动将新版本发送到PyPI。这其实是通过向push事件注册执行twine的操作实现的，具体细节请查阅Python打包用户导引[36]。

当升级了Python项目的次版本号，严谨的开发者需要在准备支持的所有Python环境中进行测试，而这是一项枯燥沉闷的工作。tox或nox可以自动完成这些工作，有兴趣的读者请自行查阅它们的官网[37][38]。

17-7. 制作PEX文件

我们已经讨论完了分发包这种针对开发者的软件分发方式。现在让我们把目光转向针对最终用户的软件分发方式。本节将讨论PEX文件。

第2章已经说明，在Unix或类Unix操作系统上，通过给Python脚本最开头添加Shebang，然后将该脚本的文件模式修改为可执行，就可以在CLI下通过输入该脚本的文件路径来执行它，不需要使用python3命令。事实上，如果该Python脚本是一个GUI脚本（使用了支持GUI的模块），那么在GUI下还可以通过点击相应图标来执行它。另外，.py后缀名并不是必须的，可以去掉。这意味着可以通过单个Python脚本实现的简单程序使用上述技巧就能转化为可执行文件，直接分发给最终用户。然而对于需要通过多个Python脚本实现的复杂程序，即便将其组织成一个包也无法通过上述技巧转化为可执行文件，因为无法给目录添加Shebang，其模式的解读方式也与文件不同。

第6章已经说明，标准库中的zipimport模块使得Python解释器可以将一个ZIP归档文件等同于一个目录。而第7章已经说明，具有__main__.py的包可以被Python解释器通过-m选项直接执行。那么，如果将一个包含__main__.py的包转换为一个ZIP归档文件，然后在该文件的开头添加Shebang，并将其模式修改为可执行，理论上同样能得到一个可执行文件，而执行它的效果等价于执行该包。这是PEP 441所引入的理念[39]，也是实现PEX文件的核心思路。

然而PEX文件并不是PEP 441所描述的ZIP归档文件。它的确是一个ZIP归档文件，但代表的其实是一个虚拟环境而非一个简单的包，所以要复杂得多。PEX文件是基于分发包构建的，其理念是先创建一个临时虚拟环境，安装指定的分发包和它所依赖的所有其他分发包，然后在该虚拟环境下自动创建一个__main__.py文件来执行指定的入口点，最后将整个虚拟环境（不包括Python解释器）制作成一个ZIP归档文件，给该文件添加Shebang并修改文件模式为可执行。这样只要一台计算机提供了能够被该虚拟环境使用的Python解释器，就可以执行该PEX文件。

有许多工具可以制作PEX文件，然而最流行的还是pex[40]。pex是PyPI上的一个项目，可以通过如下方式安装：

```
$ pip install --upgrade pex
```

在完成安装后，通过如下命令行基于上节制作的wwy_demo分发包制作一个PEX文件：

```
$ cd test_env

$ pex -o demo.pex -e demo.main:main -f dist wwy_demo
```

这样在test_env目录下会生成demo.pex。可以通过如下命令行验证demo.pex可以被直接执行：

```
$ ./demo.pex
这是一个生成阶乘序列的工具。输入“quit”退出。

请输入一个正整数: 5
[1, 2, 6, 24, 120]
请输入一个正整数: quit
```

下面让我们研究一下PEX文件的结构。由于它本质上是一个ZIP归档文件，所以可以通过unzip命令拆包。下面将demo.pex拷贝到一个空目录下，然后将其拆包：

```
$ cd
$ mkdir tmp
$ cp test_env/demo.pex tmp
$ cd tmp
$ unzip -q demo.pex
$ ls -a
.           .bootstrap  PEX-INFO    __pex__
..          .deps       __main__.py demo.pex
```

这说明demo.pex被拆包后会产生下列内容：

- PEX-INFO：提供元数据的文本文件。
- __main__.py：自动生成的入口点脚本。
- __pex__：一个目录，其内包含一个__init__.py文件，使得PEX文件可以像包一样被专门编写的导入工具直接导入。
- .deps：一个目录，存放需要安装的分发包。
- .bootstrap：一个目录，存放重现虚拟环境所需的Python脚本。

可见PEX文件还是比PEP 441描述的ZIP归档文件要复杂得多。

上面的例子已经说明，pex可以被当成shell命令使用。pex的基本语法可以概括为：

```
pex [-o pex_file] [options] [package ...]
pex {--help|-h}
pex {--version|-V}
```

当创建PEX文件是必须提供-o（也可以写作--output-file）选项以指定被创建的PEX文件的路径，注意.pex后缀不是必须的。package参数列表指定临时虚拟环境需要安装哪些分发包，每个package都代表一个符合PEP 508的依赖规格。options代表的选项非常多，本书只介绍其中最重要的一些，其余选项的用法请通过“pex --help”查看。

表17-11列出的选项被用于控制PEX文件的内容。注意-e、-c和--exe选项三者必有其一。

表17-11. pex的内容选项

选项	说明
-p *file*, --preamble-file *file*	指定生成PEX文件时使用的前导文件。
-D *dir*, --sources-directory *dir*	指定PEX文件包含dir目录下的Python脚本和数据文件。
-r *file_or_url*, --requirement *file_or_url*	指定提供依赖的文件。该选项可以重复出现多次。
-m *module*[:*symbol*], -e *module*[:*symbol*], --entry-point *module*[:*symbol*]	指定PEX文件的入口点是其包含的模块或标准库中的模块module，或者作为其包含的某个分发包的入口点的模块module中的可调用对象symbol。
-c *script*, --script *script*, --console-script *script*	指定PEX文件的入口点是作为其包含的某个分发包的入口点的script脚本。
--exe *executable*, --executable *executable*, --python-script *executable*	指定一个可直接执行的Python脚本executable作为PEX文件的入口点。该脚本会自动被PEX文件包含。

➤ -p选项指定的文件必须包含一段Python代码，这段代码会被插入PEX文件，紧随Shebang，使得执行PEX文件时会先执行这段代码。

➤ -D选项指定一个目录，以该目录为根的子目录树会被PEX文件所包含。我们会通过该目录提供额外的Python脚本和数据文件，它们就好像位于实现临时虚拟环境的目录下，却没有被制作成分发包，也没有被安装到该临时虚拟环境中。

➤ -r选项指定的本地或远程文本文件中每行都是一个依赖规格。该选项能够出现多次，与package参数列表共同决定了临时虚拟环境需要安装哪些分发包。

➤ -e选项可以指定某个标准库中的模块，通过-D选项添加的模块，或被安装到临时虚拟环境中的分发包内的模块作为PEX文件的入口点。此时只能给出module，而执行PEX文件相当于Python解释器通过-m选项执行了module模块。-e选项也可以指定某个被安装到临时虚拟环境中的分发包的入口点作为PEX文件的入口点。此时必须同时给出module和symbol，而symbol的格式是object[.attr[...]]，以保持与该分发包中通过setup()的entry_points关键字定义的入口点一致。

➤ -c选项指定以某个被安装到临时虚拟环境中的分发包的入口点作为PEX文件的入口点，而该分发包是通过setup()的scripts关键字定义了该入口点。

➤ --exe选项指定以某个可以被直接执行的Python脚本作为PEX文件的入口点，该Python脚本具有Shebang且模式被设置为可执行，不属于任何分发包，也没有通过-D选项被添加。这样PEX文件会自动包含该Python脚本。

当通过package参数列表和/或-r选项指定将某些分发包安装到虚拟环境时，应通过表17-12列出的选项控制获得这些分发包的方式。注意它们中有些与pip的通用选项含义相同。在默认情况下，相当于添加了--pypi选项，且没有添加-i选项和-f选项，这意味着所有分发包都是从PyPI下载。

表17-12. pex的分发包选项

选项	说明
--pypi	允许在PyPI中寻找指定的分发包。与--no-pypi矛盾。
--no-pypi	不允许在PyPI中寻找指定的分发包。与--pypi矛盾。
-i *url*, --index *url*	额外在url指定的仓库中寻找指定的分发包。
-f *path_or_url*, --find-links *path_or_url*	在path_or_url指定的本地或远程目录下寻找指定的分发包。
--proxy *proxy*	当访问仓库和/或远程目录时使用proxy指定的代理。
--retries *retries*	指定尝试建立TCP连接的最大次数。默认为5次。
--timeout *sec*	指定建立TCP连接的最大等待时间。默认为15秒。
--cert *path*	指定服务器端的CA证书，用于建立SSL/TLS连接。
--client-cert *path*	指定客户端的CA证书，用于建立SSL/TLS连接。
--wheel, --binary	允许下载和安装轮子。与--no-wheel等矛盾。
--no-wheel, --no-binary, --no-use-wheel, --no-use-binary	不允许下载和安装轮子。与--wheel等矛盾。
--build	允许下载和安装dist包。与--no-build矛盾。

选项	说明
--no-build	不允许下载和安装dist包。与--build矛盾。

现在，你可以明白上面的例子中创建PEX文件时将分发包wwy_demo安装到了临时虚拟环境中；入口点为该分发包内的demo.main模块内的main()函数；该分发包在当前工作目录下的dist子目录内寻找；创建的PEX文件的文件名为“demo.pex”，位于当前工作目录下。我们也可以通过如下命令行创建具有相同功能的PEX文件：

```
$ pex -o demo.pex -e demo.__main__ -f dist wwy_demo
```

但此时该PEX文件的入口点是wwy_demo分发包内的demo.__main__模块。

PEX文件携带有运行所需的所有分发包、Python脚本和数据文件，对计算机的唯一要求是Python解释器。pex会记录下其自身被安装时使用的pip所对应的Python解释器为默认解释器，而其创建的PEX文件默认仅支持该解释器。但显然，这样的PEX文件局限性太大。

为了解决这一问题，pex提供了很多环境选项来描述该PEX文件适用于哪些Python解释器，表17-13列出了其中最重要的一些。--python选项用于限定Python版本，每出现一次就增加一个支持的Python版本，而不添加该选项意味着仅支持默认解释器的Python版本。--interpreter-constraint选项用于限定Python解释器的类型，同样每出现一次就增加一个支持的类型，而不添加该选项意味着仅支持默认解释器的类型。这两个选项共同确定了PEX文件所支持的Python解释器范围，然而有时候这与默认添加的Shebang矛盾，此时就需要通过--python-shebang选项修改Shebang。注意默认添加的Shebang通常是“#!/usr/bin/env python3”。而搜索Python解释器的路径默认取创建PEX文件时PATH环境变量的取值，而这可以通过--python-path选项修改。

表17-13. pex的环境选项

选项	说明
--python *python*	指定允许使用哪些Python版本。python参数格式如“Python3.11”。该选项可以重复出现多次。
--interpreter-constraint *constraint*	指定允许使用哪些Python解释器类型。constraint参数的格式如“CPython>=2.7,<3”。该选项可以重复出现多次。
--python-shebang *shebang*	指定以“#!”加上shebang作为PEX文件的Shebang，进而指定启动哪个Python解释器。
--python-path *paths*	指定在哪些目录下搜索Python解释器。paths参数的格式为一个冒号分隔的路径列表。

除了制作PEX文件外，pex还可用于创建一个临时虚拟环境，然后在该虚拟环境下运行Python解释器，此时其语法为：

```
pex [options] [package ...] [-- arg ...]
```

也就是说，不给出-o选项。在这种情况下，通过-e、-c或--exe指定了入口点后，会直接执行入口点，而“--”后面的arg参数列表将作为命令行参数传递给入口点，例如：

```
$ cd
$ pex mypy -m mypy -- typing1.pyi
typing1.pyi:9: error: Incompatible types in assignment (expression has type
"List[object]", variable has type "Tuple[int, str]")
typing1.pyi:11: error: Incompatible types in assignment (expression has
type "Tuple[int]", variable has type "Tuple[int, str]")
typing1.pyi:13: error: Incompatible types in assignment (expression has
type "Tuple[int, str, None]", variable has type "Tuple[int, str]")
typing1.pyi:15: error: Incompatible types in assignment (expression has
type "Tuple[float, bytes]", variable has type "Tuple[int, str]")
Found 4 errors in 1 file (checked 1 source file)
```

注意这会先给临时虚拟环境安装mypy和它依赖的mypy_extensions和typig_extensions，然后再执行mypy模块，并以typing1.pyi作为命令行参数。而如果不指定入口点，则不能提供arg参数列表，此时pex会以交互式启动Python解释器，例如：

```
$ pex wwy_demo -f test_env/dist

>>> from demo import *
>>> factorial_sequence(3)
[1, 2, 6]
>>> ^D
now exiting InteractiveConsole...
```

注意这会先给临时虚拟环境安装wwy_demo，然后再以交互式启动Python 3.11，因此我们可以导入demo模块。

最后值得一提，不论是创建PEX文件，还是创建临时虚拟环境，在默认情况下pex都会通过一个缓存区来储存下载的分发包和动态生成的文件，而该缓冲区的默认位置是你的家目录下的.pex子目录。缓存区可以提高pex的执行速度，但随着时间的推移会占用越来越多的磁盘空间。在必要时，你可以整个删除作为pex缓存区的目录，以释放磁盘空间。

17-8. 制作冻结包袱

PEX文件的局限性是明显的：它只能在Unix和类Unix操作系统上使用，而且要求预先安装它所支持的范围内的某款Python解释器。这样的要求对很多最终用户过于苛刻。冻结包袱可以在Windows上使用，且由于内置了Python解释器本身，所以对操作系统没有额外的要求。制作冻结包袱的工具被称为“冻结器（freezer）”，有很多种，且每种制作出的冻结包袱的结构还各不相同。最流行的冻结器是cx_Freeze和PyInstaller，它们是流行程度仅次于

Nuitka的Python打包工具[41]。PyInstaller使用起来较复杂，而且其制作出的冻结包袱存在代码保护，不容易分析其结构，因此本节以cx_Freeze为例讨论冻结包袱的制作。

首先安装cx_Freeze。注意cx_Freeze依赖于setuptools，但很有可能不支持最新版的setuptools，安装后可能导致setuptools版本降低，因此推荐将cx_Freeze安装到某个虚拟环境。下面将cx_Freeze安装到虚拟环境freezer_env下：

```
$ python3.11 -m venv freezer_env
$ source freezer_env/bin/activate
(freezer_env)$ pip install --upgrade pip
(freezer_env)$ pip install --upgrade cx_Freeze
```

如下则是本书用于测试cx_Freeze的虚拟环境：

```
(freezer_env)$ pip list
Package    Version
---------- -------
cx-Freeze  6.11.1
packaging  21.3
pip        22.2.2
pyparsing  3.0.9
setuptools 60.10.0
```

下面将第7章中的例子demo.py拷贝到freezer_env目录下，然后通过如下命令行以默认方式基于demo.py创建冻结包袱：

```
(freezer_env)$ cxfreeze demo.py
```

这会导致在freezer_env目录下自动创建build子目录，而该目录下有名为“exe.macosx-10.9-universal2-3.11”的冻结包袱。该冻结包袱本质上是一个结构如下的目录：

```
└─ exe.macosx-10.9-universal2-3.11/
    ├─ demo
    ├─ libcrypto.1.1.dylib
    ├─ libncursesw.5.dylib
    ├─ libssl.1.1.dylib
    ├─ libtcl8.6.dylib
    ├─ libtk8.6.dylib
    └─ lib/
        ├─ Python
        ├─ library.zip
        └─ ...
```

其中可执行文件是demo，其余.dylib文件是macOS X专用的动态链接库文件，而lib目录则储存着一个名为Python的可执行文件，一个名为library.zip的ZIP归档文件，一些.so文件（Unix和类Unix操作系统通用的动态链接库文件）和大量的子目录。

下面验证该冻结包袱可以被执行：

```
(freezer_env)$ deactivate

$ ./build/exe.macosx-10.9-universal2-3.11/demo
这是一个生成阶乘序列的工具。输入“quit”退出。

请输入一个正整数: 3
[1, 2, 6]
请输入一个正整数: quit
```

该冻结包袱直接通过U盘拷贝到其他运行macOS X的平台后就实现了分发。如果要通过Internet分发则需要进一步制作成.dmg文件。

需要强调的是cx_Freeze不支持交叉编译，在哪个平台上制作的冻结包袱就只能在哪个平台上运行。例如上面例子中的冻结包袱只能在CPU的指令集架构为x86-64，操作系统为macOS X 10.9的平台上运行。要想制作能在其他平台上运行的冻结包袱，就需要在其他平台上使用cx_Freeze制作该冻结包袱，而在该平台上的冻结包袱的组成也会与上面例子中的不同。

然而通过cx_Freeze制作的冻结包袱的结构是固定的。在冻结包袱对应目录下有唯一一个可执行文件，该文件被称为“基础可执行文件（base executable）”。在该目录下还有若干基础可执行文件运行时必须的动态链接库文件。上述文件是由cx_Freeze提供的，内容不会因被打包的Python脚本而改变，但格式因平台不同而不同（例如在Windows上可执行文件的后缀名是.exe，动态链接库文件的后缀名是.dll），而基础可执行文件的文件名则会被被设置为能反映被打包的Python脚本。

被打包的Python脚本、该脚本所导入的其他模块和包，以及Python解释器本身（包括标准库），都储存在冻结包袱目录的lib子目录下。能够在ZIP归档中被正常使用的模块，都以.pyc文件的方式被储存在library.zip中；包、扩展模块和不能在ZIP归档中被正常使用的模块，则直接储存在lib子目录下。冻结包袱中的.pyc文件其实就是第6章提到过的冻结模块。可执行文件Python则用于启动Python解释器。对于上面的例子，打开library.zip后可以发现其内有demo__init__.pyc和demo__main__.pyc，对应着demo.py文件。

cx_Freeze有两种使用方式。第一种使用方式是编写一个提供元数据的脚本setup.py，在该脚本内不调用setuptools模块定义的setup()，而是调用cx_Freeze模块定义的setup()，最后通过执行该setup.py脚本来创建冻结包袱。虽然这种使用方式可以实现最精确的控制，但与制作分发包冲突。本书不详细讨论这种使用cx_Freeze的方式，如果你感兴趣请自行查阅其官方手册[42]。

第二种使用方式是用cx_Freeze提供的cxfreeze命令直接处理脚本和包，如上面的例子所示。cxfreeze命令的语法为：

```
cxfreeze [options] main_module.py
```

注意cxfreeze只能打包可以作为主模块执行的Python脚本，不能直接处理包。但该脚本导入的所有其他模块，不论它来自另外的脚本、包还是分发包都会被视为其“依赖”，进而被添加到冻结包袱中。

用cxfreeze打包一个可直接执行的包需要一点小技巧，即创建一个主模块脚本导入该包，然后打包该脚本。请删除freezer_env目录下的demo.py，将第7章中制作的demo包拷贝到该目录下，然后在该目录下创建demo1.py脚本：

```
from demo import *
from demo.main import main as _main
import sys

if __name__ == '__main__':
    sys.exit(_main())
```

这样设置后执行demo1.py等同于执行demo包，而导入demo1.py也基本上等同于导入demo包（但会额外导入一个_main标识符）。

现在我们可以通过如下命令行将demo包制作成冻结包袱：

```
$ source bin/activate
(freezer_env)$ cxfreeze demo1.py
(freezer_env)$ deactivate
```

这同样会在build目录下生成exe.macosx-10.9-universal2-3.11子目录，但该目录下的基本可执行文件变成了demo1。lib子目录下将多出demo目录，其内容是__init__.pyc、main.pyc和factorial_sequence.pyc，这是对应demo包的冻结模块。可以验证按上述方法生成的冻结包袱可以被执行：

```
$ ./build/exe.macosx-10.9-universal2-3.11/demo1
这是一个生成阶乘序列的工具。输入“quit”退出。

请输入一个正整数: 3
[1, 2, 6]
请输入一个正整数: quit
```

表17-14列出了cxfreeze的基本选项。--target-name用于指定基本可执行文件的文件名，默认取被打包脚本的文件名（不包括.py）。--target-dir用于指定存放冻结包裹的目录，默认为build目录下自动基于平台和Python解释器版本创建的子目录。

表17-14. cxfreeze的基本选项

选项	说明
--target-name=*file*	指定基本可执行文件的文件名。
--target-dir=*dir*, --install-dir=*dir*	指定存放冻结包袱的目录。

一般而言，cx_Freeze会根据被打包的脚本自动识别它的依赖，并将这些依赖最合理的添加到冻结包袱的lib目录下。然而如果你发现默认设置下生成的冻结包袱效果不尽人意，可以通过表17-15列出的选项手工控制冻结包袱的内容。--base-name用于指定以哪个文件作为基础可执行文件。--pacakges、--include-files、--includes、--excdlues、--bin-includes和--excludes共同决定了除了被打包的脚本之外还有哪些包、模块、数据文件和扩展模块被添加到lib目录下。--zip-include-packages、--zip-exclude-packages和-z共同决定了哪些包、模块和数据文件被存放在library.zip。-c选项使得library.zip被压缩。-O和-OO选项用于强制优化冻结包袱中的.pyc文件。

表17-15. cxfreeze的内容选项

选项	说明
--base-name=*file*	指定基础可执行文件。
--packages=*packages*	指定包含哪些包。packages是用逗号分隔的包名列表。
--include-files=*files*	指定包含哪些数据文件。files是用逗号分隔的路径列表。
--includes=*modules*, --include-modules=*modules*	指定包含哪些模块。modules是用逗号分隔的模块名列表。
--excludes=*modules*, --excludes-modules=*modules*	指定不包含哪些模块。modules是用逗号分隔的模块名列表。
--bin-includes=*files*	指定包含哪些动态链接库。files是逗号分隔的文件名列表。
--bin-excludes=*files*	指定不包含哪些动态链接库。files是逗号分隔的文件名列表。
--zip-include-packages=*packages*	指定library.zip中包含哪些包。packages是用逗号分隔的包名列表。
--zip-exclude-packages=*packages*	指定library.zip中不包含哪些包。packages是用逗号分隔的包名列表。
-z *file*, --zip-include=*file*	指定library.zip中包含哪些脚本和数据文件。file是一个指向文件的路径。该选项可以出现多次。
-c, --compress	压缩library.zip。
-O	优化.pyc文件中的字节码。
-OO	去掉.pyc文件中的文档字符串。

当打包的是GUI版Python程序时，我们可以使用表17-16列出的选项指定仅在GUI下有意义的资源。其中--icon用于指定程序图标，是所有平台通用的。其余选项则专门用来提供Windows程序需要的资源。

表17-16. cxfreeze的GUI选项

选项	说明
--icon=*icon*	指定GUI界面下显示的图标。
--manifest *file*	指定清单文件。仅适用于Windows。
--uac-admin	创建一个指定需要管理员权限的清单文件。仅适用于Windows。
--shortcut-name *name*	为msi包指定快捷方式名。仅适用于Windows。
--shortcut-dir *dir*	为msi包指定存放快捷方式的目录。仅适用于Windows。
--copyright	指定在版本信息中包含版权信息。仅适用于Windows。
--trademarks	指定在版本信息中包含商标。仅适用于Windows。

最后，我们可以通过表17-17列出的选项控制cxfreeze在执行过程中的行为细节。然而除非你对cx_Freezer的运行机制非常熟悉，否则没有必要使用这些选项。这些选项中最常使用的是-s，它能过滤掉打包过程中产生的提示性信息，仅保留错误和警告。

表17-17. cxfreeze的行为选项

选项	说明
--init-script=*file*	指定启动cxfreeze时自动执行的脚本。
--default-path=*dirs*	指定搜索包和模块时使用的sys.path的初始值。dirs是用冒号分隔的路径列表。
--include-path=*dirs*	指定搜索包和模块时向sys.path添加哪些目录。dirs是用冒号分隔的路径列表。
--bin-path-includes=*dirs*	指定搜索动态链接库时包含哪些目录。dirs是用冒号分隔的路径列表。
--bin-path-excludes=*dirs*	指定搜索动态链接库时不包含哪些目录。dirs是用冒号分隔的路径列表。
--replace-paths=*directives*	进行路径替换。directives是用冒号分隔的指令列表。每条指令都是格式为path1=path2的名值对，表示将path1替换成path2。最后一条指令可以是*=pathN，表示将所有没明确给出的路径都替换为pathN。
-s, --silent	仅显示错误和警告，不显示其他输出。

由于demo.py和demo包的结构过于简单，所以下面给出一个较复杂的例子来说明如何用cx_Freezer制作冻结包袱。请将freezer_env下的demo包和demo1.py都删除，然后将前面下载的mypy的dist包中包含的mypy包拷贝到freezer_env下，再创建mypy.py脚本：

```
from mypy import *
from mypy.__main__ import console_entry as _console_entry

if __name__ == '__main__':
    _console_entry()
```

同样执行mypy.py就等同于执行mypy包，而导入mypy.py也基本上等同于导入mypy包（但会额外导入一个_console_entry标识符）。接下来尝试创建基于mypy的冻结包袱：

```
     $ source bin/activate
     (freezer_env)$ cxfreeze --target-name=mypy-bin --target-dir=mypy_bundle -c
-s mypy.py
```

你会发现冻结包袱创建成功，被存放在mypy_bundle目录下，可执行文件名为mypy-bin。然而尝试执行mypy-bin会报错，因为缺少mypy_extensions和typing_extensions。

我们给freezer_env虚拟环境安装mypy_extensions和typing_extensions，然后再次尝试创建基于mypy的冻结包袱：

```
     (freezer_env)$ pip install --upgrade mypy_extensions typing_extensions
     (freezer_env)$ cxfreeze --target-name=mypy-bin --target-dir=mypy_bundle -c
-s mypy.py
     (freezer_env)$ deactivate
```

你会发现这次创建的冻结包袱能够被成功执行。检查lib目录，你会发现它直接包含mypy目录，而library.zip除了包含mypy_bin__init__.pyc和mypy_bin__main__.pyc之外，还包括mypy_extensions.pyc和typing_extensions.pyc，以及mypy_extensions-0.4.3.dist-info和typing_extensions-4.3.0.dist-info目录。这说明分发包mypy_extensions和typing_extensions被自动打包进了该冻结包袱。

17-9. 制作C可执行程序

由于每个冻结包袱都需要完整包含一个Python解释器，所以其大小不会小于Python解释器的大小，哪怕只有一个Python脚本。这导致了冻结包袱会占用较多磁盘空间。此外，由于本质上依然是在通过Python解释器来解释执行Python脚本，所以其执行速度不会很快。对于最终用户来说，分发Python程序的最好方式是将它们先转换成C程序，再编译为可执行程序。我们已经知道可以通过Nuitka做到这点，本节讨论如何使用Nuitka。

由于在编写本书时Nuitka支持的最高Python版本是3.10，所以先要在系统中安装Python 3.10。注意在按照官网提供的安装方式安装完Python 3.10后，python3命令将启动Python 3.10而非Python 3.11，只能通过python3.11命令来启动Python 3.11。这是因为在PATH环境变量的值中，指向Python 3.10的目录位于指向Python 3.11的目录前面，因而前者屏蔽了后者。如果你不满足于这种状态，则需要修改一些配置文件。假如你使用的是Unix或类Unix操作系统，只需要修改家目录下的.bash_profile（或等价文件），注释掉下面的命令行：

```
     PATH="/Library/Frameworks/Python.framework/Versions/3.10/bin:${PATH}"
     export PATH
```

这样重新启动shell后，python3命令将重新启动Python 3.11，只能通过python3.10来启动Python 3.10。用类似的办法，系统中可以有任意多个Python版本并存。

下面将Nuitka和支持它的额外功能的ordered-set和zstandard都安装到使用Python 3.10的虚拟环境nuitka_env下：

```
$ cd
$ python3.10 -m venv nuitka_env
$ source nuitka_env/bin/activate
(nuitka_env)$ pip install --upgrade pip setuptools
(nuitka_env)$ pip install --upgrade Nuitka ordered-set zstandard
```

这样本书用于测试Nuitka的虚拟环境就是：

```
(nuitka_env)$ pip list
Package    Version
---------- -------
Nuitka     1.0.7
ordered-set 4.1.0
pip        22.2.2
setuptools 65.3.0
zstandard   0.18.0
```

此外Nuitka需要使用C编译器，所以请按照如下方法满足该条件：

1. 如果你使用的是macOS X，则系统中已经安装有clang，但还需要在终端中执行“xcode-select --install”来安装xcrun。
2. 如果你使用的是其他Unix或类Unix操作系统，则请自行安装gcc。
3. 如果你使用的是Windows，则请安装MinGW64（必须基于11.2以上版本的gcc），或者2022以上版本的Visual Studio（基于MSVC）。

现在把demo包和上一节编写的demo1.py拷贝到nuitka_env目录下，然后用Nuitka对demo1.py进行打包：

```
(nuitka_env)$ cd nuitka_env
(nuitka_env)$ python3 -m nuitka --quiet demo1.py
Nuitka-Options:WARNING: You did not specify to follow or include anything
but
Nuitka-Options:WARNING: main program. Check options and make sure that is
Nuitka-Options:WARNING: intended.
```

这样在nuitka_env目录下会多出一个存放C源代码的目录demo1.build，以及可执行文件demo1.bin（如果是在Windows上则是demo1.exe，下同）。可以验证该可执行文件放在当前目录下时可以正常执行：

```
(nuitka_env)$ ./demo1.bin
这是一个生成阶乘序列的工具。输入“quit”退出。
```

```
请输入一个正整数: 3
[1, 2, 6]
请输入一个正整数: quit
```

然而将demo1.bin移动到其他目录时就不再能正常执行了，而会报错找不到demo模块。事实上上面例子中的警告信息已经说明，demo1.bin中只包含了demo1.py本身，没有包含该脚本导入的其他模块。

Nuitka的语法可以概括为：

```
python3 -m nuitka [options] main_module.py
python3 -m nuitka {--help|-h}
python3 -m nuitka --version
```

其中main_module.py同样是主模块脚本，与在cxfreeze中含义相同。事实上Nuitka还提供了一些shell命令，例如我们可以用nuitka3代替“python3 -m nuitka”，但建议总是通过-m启动Nuitka，因为这样可以让Nuitka明确使用哪个Python解释器。

Nuitka提供了非常多的选项，可以通过“python3 -m nuitka --help”查看。表17-18和表17-19列出的选项是最常用的。此外上面例子中的--quiet选项是使Nuitka在运行过程中只输出错误和警告。

表17-18列出的选项明确了Nuitka需要完成什么任务。选项--standalone、--onefile和--module相互矛盾，至多只能有一个被添加。当这三个选项没有一个被添加时，Nuitka将创建一个需要系统中预先安装了Python解释器才能执行的可执行文件，与PEX文件的区别仅在于所有模块都被转换成了扩展模块。--disable-console和--enable-console也是矛盾的，如果都不添加默认相当于添加了--enable-console。-o和--output则共同控制着最终输出。

表17-18. Nuitka的基本选项

选项	说明
--standalone	创建不需要Python解释器就可以执行的可执行文件。隐含了--follow-imports和--python-flag=no_site。结果是一个包含可执行文件的.dist归档文件。
--onefile	隐含了--standalone，在.dist归档文件的基础上创建一个更小的可执行文件。
--module	创建扩展模块，而非可执行文件。
--disable-console	指定创建GUI程序。
--enable-console	指定创建CLI程序。
-o *file*	指定可执行文件的文件名。不能与--standalone或--module联合使用。
--output-dir=*dir*	指定将生成的中间结果（C源代码）和最终结果（可执行文件、.dist归档文件或扩展模块）存放在哪个目录。

在默认情况下，Nuitka仅会将main_module.py翻译为C源代码，而忽略它导入的任何模块。这一行为可以通过表17-19中的选项改变，但它们的含义在Nuitka执行不同任务时略有

差异，所以不能随意搭配。需要强调的是，当添加了--standalone或--onefile时，相当于自动添加了--folow-imports。

表17-19. Nuitka的跟踪选项

选项	说明
--follow-imports	递归地跟踪主模块导入的所有模块。
--nofollow-imports	不跟踪主模块导入的任何模块。
--follow-import-to=*module*	递归地跟踪指定的模块。可以重复出现多次。
--nofollow-import-to=*module*	不跟踪指定的模块。可以重复出现多次。
--follow-stdlib	将标准库中的模块当成普通模块处理。

现在你应该明白了，在上面的例子中Nuitka仅将demo1.py翻译为C源代码，然后再编译为可执行文件。也就是说demo1.bin仅对应demo1.py本身，与demo包没有关系，导致它离开nuitka_env就失效。这也是Nuitka会给出警告的原因。此时如果添加--nofollow-imports选项，则不会给出警告，但创建的可执行文件依然没有用处。要想使创建的可执行文件能够在文件系统中移动，应通过--follow-imports指定跟踪主模块导入的所有非标准库模块，例如：

```
(nuitka_env)$ python3 -m nuitka --follow-imports demo1.py
```

可以验证这次生成的demo1.bin移动到其他目录后依然能正常执行，且要比前一次生成的demo1.bin要大约50KB。

Nuitka的上述用法可以与pex类比。然而可执行文件比PEX文件小得多，执行速度也要快得多。同样是打包demo包，demo.pex有504KB，而demo1.bin只有242KB。但另一方面，如果被打包的Python程序依赖于其他分发包，那么PEX文件可以包含整个虚拟环境，而以上述方法创建的可执行文件只能在重现了该虚拟环境的情况下可以被正常执行。

当添加了--standalone后，Nuitka将创建一个后缀名为.dist的归档文件，相当于一个目录。请通过如下命令行来获得这样的.dist文件：

```
(nuitka_env)$ python3 -m nuitka --standalone demo1.py
```

这样在nuitka_env目录下同样会生成一个储存C源代码的demo1.build目录，但demo1.bin被demo1.dist代替。在Unix和类Unix操作系统上可以通过如下命令行查看demo1.dist的内容：

```
(nuitka_env)$ ls demo1.dist/
Python                  _elementtree.so         _struct.so
_asyncio.so             _hashlib.so             _uuid.so
_bisect.so              _heapq.so               _zoneinfo.so
_blake2.so              _json.so                array.so
_bz2.so                 _lzma.so                audioop.so
_codecs_cn.so           _multibytecodec.so      binascii.so
_codecs_hk.so           _multiprocessing.so     demo1
_codecs_iso2022.so      _opcode.so              fcntl.so
_codecs_jp.so           _pickle.so              grp.so
_codecs_kr.so           _posixshmem.so          libcrypto.1.1.dylib
_codecs_tw.so           _posixsubprocess.so     libssl.1.1.dylib
_contextvars.so         _queue.so               math.so
_crypt.so               _random.so              mmap.so
_csv.so                 _scproxy.so             pyexpat.so
_ctypes.so              _sha512.so              select.so
_datetime.so            _socket.so              termios.so
_dbm.so                 _ssl.so                 unicodedata.so
_decimal.so             _statistics.so          zlib.so
```

可以看出该.dist文件与用cx_Freeze制作的冻结包袱很相似。事实上该.dist文件除了demo1相当于demo1.bin外，其余文件都来自Python解释器（包括标准库）。.dist文件中不包含任何.pyc文件，特殊文件Python也要比冻结包袱中的同名可执行文件小得多（因为它只包含libpython）。由于上述原因，.dist文件比冻结包袱小很多，在本书的例子中demo1.dist只有20.6MB，而上节基于demo包创建的冻结包袱则有60.4MB。

可以通过如下命令行验证demo1.dist文件内的demo1是可执行的：

```
(nuitka_env)$ ./demo1.dist/demo1
这是一个生成阶乘序列的工具。输入“quit”退出。

请输入一个正整数: 3
[1, 2, 6]
请输入一个正整数: quit
```

demo1.dist被拷贝到任何操作系统是MacOS X的计算机上都能通过这种方法被执行。

在创建.dist文件时，默认会打包整个Python标准库，但可以通过添加--follow-stdlib进行精简，即只包含主模块直接或间接导入的标准库模块。下面重新制作demo1.dist：

```
(nuitka_env)$ python3 -m nuitka --standalone --follow-stdlib demo1.py
```

你会发现这次demo1.dist的大小只有9.8MB。通过如下命令行可以发现demo1.dist中少了很多动态链接库文件：

```
(nuitka_env)$ ls demo1.dist/
Python                  _codecs_kr.so           binascii.so
_bz2.so                 _codecs_tw.so           demo1
_codecs_cn.so           _lzma.so                grp.so
_codecs_hk.so           _multibytecodec.so      unicodedata.so
_codecs_iso2022.so      _opcode.so              zlib.so
_codecs_jp.so           _struct.so
```

如果你不满意于生成一个.dist文件，想要得到一个单独的可执行文件，且它的运行不依赖于预安装的Python解释器，则应添加--onefile。请执行如下命令行：

```
(nuitka_env)$ python3 -m nuitka --onefile demo1.py
```

这同样会先生成demo1.build目录和demo1.dist文件，然后再生成demo1.onefile-build目录和demo1.bin文件，前者储存的是额外的C源代码，后者是包含了Python解释器的可执行文件。可以验证demo1.bin可以在任何安装有MacOS X的计算机上运行，而它的大小只有6.4MB。

在使用--onefile时，也可以通过--follow-stdlib进一步缩小可执行文件的大小：

```
(nuitka_env)$ python3 -m nuitka --onefile --follow-stdlib demo1.py
```

这次生成的demo1.bin的大小只有3.2MB。

以上就是用Nuitka创建可执行文件的方法。而我们还可以通过--module用Nuitka将脚本或包转换为扩展模块，这些扩展模块能够以轮子的形式被发布，而相应的C源代码则能够以dist包的形式被发布。在制作扩展模块时，需要用--follow-import-to和--nofollow-import-to精确说明跟踪哪些模块，如果只需要主模块也必须用--nofollow-imports说明。对于上面的例子来说，必须跟踪demo包：

```
(nuitka_env)$ python3 -m nuitka --module --follow-import-to=demo demo1.py
```

这会使我们得到储存C源代码的demo1.build目录、代表扩展模块的demo1.cpython-310-darwin.so文件、以及说明该模块导入了哪些其他模块的demo1.pyi文件。demo1.pyi本质上是一个存根文件，目前其内容为：

```
import factorial_sequence
import re
import demo
import demo.main

__name__ = ...
```

以后还可能包含被导入标识符的类型信息。可以验证demo1.cpython-310-darwin.so能够被导入：

```
(nuitka_env)$ python3

>>> import demo1
>>> demo1._main
<compiled_function main at 0x102fb7880>
>>> demo1.factorial_sequence
<compiled_function factorial_sequence at 0x102fb73d0>
>>>
```

拷贝该扩展模块到其他安装有MacOS X的计算机上后也能正常使用，其大小只有221KB。

可以看出，Nuitka的任何操作都需要用C编译器将作为中间结果的C源代码编译为最终结果。我们可以通过表17-20列出的选项对C编译器进行更精确的控制。此外Nuitka创建的可执行文件被执行时本质上是通过Python解释器来执行扩展模块，而Python解释器启动时的设置会影响到对扩展模块的执行方式，因此在创建可执行文件或扩展模块时需要控制和记录这些设置。默认情况下，当通过“python3 --python-options -m nuitka”启动Nuitka时，后者会自动从--python-options代表的Python选项中提取相关信息，但我们也可以通过--python-flag选项提供这些信息。

表17-20. Nuitka的编译选项

选项	说明
--clang	指定使用clang编译C源代码。
--mingw64	指定使用MinGW64编译C源代码。
--msvc=*version*	指定使用版本号为version的MSVC编译C源代码。
--python-flag=*flags*	指定启动Python解释器时添加的选项和其他行为。flags是一个用逗号分隔的列表，列表中可以包括： no_site：添加-S选项。 no_asserts：添加-O选项。 no_warnings：不抛出警告。 no_docstrings：不包含文档字符串。 unbuffered：添加-u选项。 static_hashes：计算哈希值时不使用随机盐值。 该选项可以重复出现多次。

注意由于Nuitka制作可执行文件时无法处理已经安装的第三方模块，所以--standalone和--onefile会自动添加“--python-flag=no_site”。这意味着被打包的Python程序如果依赖于某些分发包，则需要下载相应的dist包，找出相关包或模块的源代码与该Python程序放在同一目录下。

最后，如果用Nuitka打包的Python程序是一个GUI程序，则还需要进行一些额外的设置。表17-21列出了其中最常用的一些选项。

表17-21. Nuitka的GUI选项

选项	说明
**`--windows-icon-from-ico=`*`path`*	指定Windows图标。该选项可重复出现多次。
`--window-uac-admin`	指定Windows用户控制要求管理员权限。
**`--macos-app-icon=`*`path`*	指定macOS X图标。
**`--linux-icon=`*`path`*	指定Linux图标。

现在我们已经完成了对Nuitka的讨论。如果你想掌握Nuitka的其他选项，请自行查阅其官方手册[43]，并结合“python3 -m nuitka --help”的输出。

17-10. conda简介

至此本书已经完成了对Python官方生态系统的讨论。在本章的最后一节，我们讨论基于conda的第三方生态系统，这包括Anaconda和conda-forge。首先需要再次强调，之所以需要建立Anaconda，是因为PyPI只负责维护Python项目，而AI研究者们需要综合使用不同编程语言编写的工具；而之所以需要建立conda-forge，是因为Anaconda是一家盈利性的公司，不是一个开源组织。

基于conda的第三方生态系统是独立于Python官方生态系统的，但两者之间存在互动，例如可以通过pip安装conda（但通常不是最新版本），在conda环境中也可以使用pip（但推荐优先使用conda）。由于本书已经用大量篇幅详细讨论了Python官方生态系统，限于篇幅我们不可能再对基于conda的第三方生态系统进行同等程度的详细讨论，所以下面仅澄清后者的关键概念，并介绍conda命令的基本用法。如果你想了解更多请自行查阅conda的官方手册[44]。

Python官方生态系统的基础是分发包，就连第三方工具pex、cx_Freeze和Nuitka都支持分发包。然而分发包只能处理纯Python项目，而基于conda的第三方生态系统需要支持包括FORTRAN、C、C++、Java、JavaScript、R、Ruby、Scala和Lua在内的多种编程语言，这使得该生态系统不的不放弃了分发包，而以所谓的“conda包（conda packages）”作为基础。下面介绍conda包的结构。

旧版本的conda包是用tar打包用bzip2压缩的压缩归档文件，后缀名是.tar.bz2；新版本的conda包是一个ZIP归档文件，其内包含两个用tar打包用zstd压缩（将来可能替换成更新的压缩算法）的归档文件，后缀名是.conda。目前两种conda包都可以使用，推荐使用后者，但conda版本小于4.7时只能使用前者。

不论哪种conda包，其内容的逻辑结构都如下所示：

```
    ├─ bin/
    ├─ lib/
    └─ info/
        ├─ files
        ├─ paths.json
        ├─ index.json
        ├─ has_prefix (Optional)
        ├─ no_link (Optional)
        ├─ about.json (Optional)
        ├─ LICENSE.txt (Optional)
        └─ recipe/
```

其中bin目录存放可执行文件和动态链接库文件，lib目录存放脚本文件，info目录存放元数据文件。而必须的元数据文件包括：

- files：一个清单，每行代表一个该conda包中的非元数据文件的路径。
- paths.json：该conda包中的所有非元数据文件的路径、哈希值和大小。JSON格式。
- index.json：该conda包的基本元数据，包括包名、版本、许可证、支持的平台、依赖关系和生成时间等。JSON格式。

可选的元数据文件包括：

- has_prefix：用于实现路径转换，每行都将files中的若干路径映射到各自的安装路径。
- no_link：用于指定必须拷贝的文件，每行都对应files中的某个路径。
- about.json：meta.yaml文件中的“about节”。JSON格式。

此外，info/recipe目录下会储存该conda包的“创建配方（build recipe）”，以指导该包该被如何安装。

显然，基于conda的第三方生态系统涉及的仓库储存的都是conda包，它们与PyPI一样通过HTTP协议提供conda包的下载。然而与pip不同，conda将每个仓库都抽象为一个“通道（channels）”。conda有唯一的一个默认通道，但还可以具有任意多个可选通道。

在系统中安装conda有三种方式：

1. 安装Miniconda：这会导致以Anaconda为默认通道，但只会安装conda本身和它的依赖。此后用户可以自己选择安装Anaconda中的conda包，但其中有些是免费的，有些是收费的。
2. 安装Anaconda：与安装Miniconda的唯一区别是会自动从Anaconda下载并安装所有7500个conda包，以形成“数据科学家”所需的完整软件环境。这必然是收费的。
3. 安装Miniforge：与安装Miniconda的唯一区别是以conda-forge为默认通道。

本书以Miniconda为例，请根据conda的官方手册在你的系统中安装Miniconda。简单来说，你需要下载Miniconda安装器，然后运行它。具体来说，对于Windows这是一个exe文件，安装过程与安装其他Windows应用是一样的，卸载方式也相同。对于Unix和类Unix操作

系统来说这是一个shell脚本（特别针对macOS X还提供了.pkg文件），执行该脚本即完成安装，但卸载时需要手工删除如下目录和文件：

➤ Miniconda的基目录：可以通过执行“conda info --base”找到，在我的系统中是/Users/wwy/opt/miniconda3。
➤ .conda目录：位于家目录下，用于储存环境和系统信息。
➤ .condarc文件：需要自己创建，位于家目录下，conda的配置文件。

需要强调的是，如果你像本书一样先安装CPython，然后再安装Miniconda，那么在安装完Miniconda后你会发现python3命令不会再启动Python 3.11，而会启动Python 3.9。这是因为在安装Miniconda的过程中会执行“conda init”，会自动将下列代码添加到shell的启动脚本（例如.bash_profile）中：

```
# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/Users/wwy/opt/miniconda3/bin/conda' 'shell.bash' 'hook'
2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/Users/wwy/opt/miniconda3/etc/profile.d/conda.sh" ]; then
        . "/Users/wwy/opt/miniconda3/etc/profile.d/conda.sh"
    else
        export PATH="/Users/wwy/opt/miniconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<
```

这使得当启动shell时会自动配置好conda，并激活base环境。建议在这段代码后面添加如下命令行：

```
conda deactivate
```

这样在启动shell脚本时依然会自动配置好conda，但会先激活base环境然后再退出，使得python3命令依然启动Python 3.11。

现在我们可以使用conda命令了，该命令的语法可以概括为：

```
conda command [options] [args]
conda {--help|-h} [command]
conda {--version|-V}
```

表17-22列出了所有的conda命令，表17-23～表17-26则列出了常用的conda选项。下面会讨论这些命令中大部分，但不可能详细讨论每个命令的细节。对于本书未覆盖的内容，你可以通过执行“conda --help”查看关于conda的帮助信息，通过执行“conda xxx --help”查看关于conda命令xxx的帮助信息，还可以查阅conda官方手册。

表17-22. conda命令

命令	说明
search	在指定的频道或本地搜索conda包。
create	创建一个新conda环境，并安装指定的conda包。
install	从指定的频道下载指定的conda包，并将它们安装在指定conda环境下。
upgrade, update	升级指定conda环境下的指定conda包。
uninstall, remove	从指定conda环境卸载指定的conda包。
list	显示指定conda环境已经安装的conda包。
run	在指定的conda环境中执行指定的可执行文件。
compare	比较conda环境。
clean	清理conda系统。
info	显示conda系统统计信息。
config	修改.condarc文件。

表17-23. conda命令的通用选项

选项	说明
-h, --help	显示指定命令的帮助信息。
--json	以JSON格式给出输出信息，以便于自动处理。
-v, --verbose	显示更多输出信息。可以出现1～3次，分别代表info、debug和trace级别的信息。
-q, --quiet	不显示进度条。

表17-24. conda命令的频道选项

选项	说明
-c *chan*, --channel *chan*	除了搜索默认频道和在.condarc中添加的频道外，还在chan中搜索。可以重复出现多次。chan还可以取特殊值local，表示在本地搜索。
--user-local	等价于“-c local”。
--override-channels	不搜索默认频道和在.condarc中添加的频道。必须与-c联合使用。

表17-25. conda命令的环境选项

选项	说明
-n *env*, --name *env*	指定conda环境的名字。
-p *path*, --prefix *path*	指定conda环境的路径。

表17-26. conda命令的运行方式选项

选项	说明
-d, --dry-run	仅显示即将执行哪些操作，但不实际执行。
-y, --yes	执行命令过程中不需要向用户寻求确认。

conda提供了search命令以在指定的频道中搜索，这是pip所不具备的。search命令接受一个参数作为搜索关键字，可以是完整的包名（例如“numpy”），可以是用通配符*表示的包名的一部分（例如“num*”、“*py”和“*ump*”），还可以指定版本（例如“numpy>=1.20”）。默认情况下，search命令仅会在默认频道和.condarc中指定的频道搜索，并合并搜索结果。我们可以用表17-14列出的选项对搜索范围进行精确控制。

由于我们没有创建.condarc文件，所以下面的例子仅在Anaconda仓库中搜索pytorch：

```
$ conda search pytorch
Loading channels: done
# Name                       Version           Build  Channel
pytorch                        1.3.1 cpu_py27h0c87eb2_0  pkgs/main
pytorch                        1.3.1 cpu_py36h0c87eb2_0  pkgs/main
pytorch                        1.3.1 cpu_py37h0c87eb2_0  pkgs/main
pytorch                        1.4.0 cpu_py36hf9bb1df_0  pkgs/main
pytorch                        1.4.0 cpu_py37hf9bb1df_0  pkgs/main
pytorch                        1.4.0 cpu_py38hf9bb1df_0  pkgs/main
pytorch                        1.6.0 cpu_py37hd70000b_0  pkgs/main
pytorch                        1.6.0 cpu_py38hd70000b_0  pkgs/main
pytorch                        1.7.1 cpu_py37hb87dcc5_0  pkgs/main
pytorch                        1.7.1 cpu_py38hb87dcc5_0  pkgs/main
pytorch                        1.7.1 cpu_py39h7e2095a_0  pkgs/main
pytorch                       1.10.2 cpu_py310h30e64cd_0  pkgs/main
pytorch                       1.10.2 cpu_py37h903acac_0  pkgs/main
pytorch                       1.10.2 cpu_py38h903acac_0  pkgs/main
pytorch                       1.10.2 cpu_py39h903acac_0  pkgs/main
```

下面的例子同时在Anaconda和conda-forge中搜索pytorch，并限制版本为1.10.2：

```
$ conda search -c conda-forge pytorch==1.10.2
Loading channels: done
# Name                       Version           Build  Channel
pytorch                       1.10.2 cpu_py310h30e64cd_0  pkgs/main
pytorch                       1.10.2 cpu_py37h3ff094a_1  conda-forge
pytorch                       1.10.2 cpu_py37h903acac_0  pkgs/main
```

```
pytorch                        1.10.2 cpu_py38h2cb30f8_1  conda-forge
pytorch                        1.10.2 cpu_py38h903acac_0  pkgs/main
pytorch                        1.10.2 cpu_py39h19568cb_1  conda-forge
pytorch                        1.10.2 cpu_py39h903acac_0  pkgs/main
```

而下面的例子仅在conda-forge中搜索以“tensorl”开头的conda包：

```
$ conda search -c conda-forge --override-channels tensorl*
Loading channels: done
# Name                       Version           Build  Channel
tensorly                       0.4.2            py_0  conda-forge
tensorly                       0.4.2            py_1  conda-forge
tensorly                       0.4.4            py_0  conda-forge
tensorly                       0.4.5            py_0  conda-forge
tensorly                       0.5.0    pyh9f0ad1d_0  conda-forge
tensorly                       0.5.1    pyhd3deb0d_0  conda-forge
tensorly                       0.7.0    pyhd8ed1ab_0  conda-forge
tensorly-quantum               0.1.0    pyhd8ed1ab_0  conda-forge
tensorly-torch                 0.3.0    pyhd8ed1ab_0  conda-forge
```

“conda环境（conda environment）”是使用conda时必须时刻注意的一个概念。它与虚拟环境类似，但不是通过venv模块创建的。当安装了Miniconda后，会自动创建一个并为“base”的conda环境，对应的目录即Miniconda的基目录。理论上我们也可以向base环境安装conda包，但这是极不推荐的。强烈建议不改变base环境的任何设置，而为任何项目都创建新的conda环境。

创建conda环境的命令是create，它需要通过表17-25中的选项之一指定新创建的conda环境的名称或路径（它们可以相互推导），例如下面的例子创建了一个名为“example”的conda环境：

```
$ conda create -n example
```

而该conda环境对应的目录在我的系统中是/Users/wwy/opt/miniconda3/envs/example，也就是说在Miniconda基目录中的envs子目录下创建名为“exmaple”的子目录。目前这是一个空的conda环境，其目录结构为：

```
└─ example/
    └─ conda-meta/
        └─ history
```

其中history是储存该conda环境的修改记录的日志文件。

conda环境的使用方式与虚拟环境类似，需要激活。然而激活conda环境的方式不是执行相应的activate脚本，而是执行“conda activate xxx”，其中xxx是conda环境名。例如下面的命令行激活example环境：

```
$ conda activate example
```

而example环境被激活后，在命令行提示符前会多出“(example)”，这与激活虚拟环境是类似的。在激活一个conda环境的情况下，可以直接激活另一个conda环境，例如：

```
(example)$ conda activate base
```

取消激活conda环境的是执行“conda deactivate”，注意这会退出当前conda环境，回到之前激活的conda环境，直到退出，例如：

```
(base)$ conda deactivate
(example)$ conda deactivate
$
```

conda环境和虚拟环境并不冲突。在激活某个conda环境的情况下，可以激活虚拟环境，例如：

```
$ conda activate example
(example)$ source example_env/bin/activate
(example_env)(example)$ deactivate
(example)$ conda deactivate
$
```

在激活某个conda环境的情况下，PATH环境变量会被添加该conda环境对应目录下的bin子目录，这就是conda环境的意义。然而一个空的conda环境不会造成任何影响。我们需要通过install命令将conda包安装到指定的conda环境。该命令也需要通过表17-24列出的选项指定在哪些频道搜索，且需要通过表17-25列出的选项指定conda环境，如果省略了后者则将conda包安装到当前被激活的conda环境。install命令接受任意多个conda包名作为参数，且包名之后还可以用“=”指定版本号。下面的例子将Python 3.8安装到example环境下：

```
$ conda install -n example python=3.8
```

注意CPython所依赖的其他conda包（不一定与Python有关，例如zlib和openssl），也会自动被安装。此时example环境的目录结构变成了：

```
└─ example/
    ├─ conda-meta/
    ├─ bin/
    ├─ include/
    ├─ lib/
    ├─ share/
    └─ ssl/
```

现在激活example环境，执行“python3”，你会发现启动的是Python 3.8。

```
$ conda activate example
(example)$ python3
```

下面的例子则将numpy的最新版本安装到了example环境中：

```
(example)$ conda install numpy
(example)$ python3

>>> import numpy
>>> numpy
<module 'numpy' from '/Users/wwy/opt/miniconda3/envs/example/lib/python3.8/
site-packages/numpy/__init__.py'>
>>> ^D
```

list命令被用于显示指定conda环境中安装了哪些conda包，同样需要通过表17-25列出的选项指定conda环境，如果省略则作用于当前被激活的conda环境。下面的例子显示了当前安装到example环境中的conda包：

```
(example)$ conda list
# packages in environment at /Users/wwy/opt/miniconda3/envs/example:
#
# Name                    Version                   Build  Channel
blas                      1.0                         mkl
ca-certificates           2022.07.19           hecd8cb5_0
certifi                   2022.9.14         py38hecd8cb5_0
intel-openmp              2021.4.0           hecd8cb5_3538
libcxx                    14.0.6               h9765a3e_0
libffi                    3.3                  hb1e8313_2
mkl                       2021.4.0           hecd8cb5_637
mkl-service               2.4.0             py38h9ed2024_0
mkl_fft                   1.3.1             py38h4ab4a9b_0
mkl_random                1.2.2             py38hb2f4e1b_0
ncurses                   6.3                  hca72f7f_3
numpy                     1.23.1            py38h2e5f0a9_0
numpy-base                1.23.1            py38h3b1a694_0
openssl                   1.1.1q               hca72f7f_0
pip                       22.1.2            py38hecd8cb5_0
python                    3.8.13               hdfd78df_0
readline                  8.1.2                hca72f7f_1
setuptools                63.4.1            py38hecd8cb5_0
six                       1.16.0              pyhd3eb1b0_1
sqlite                    3.39.2               h707629a_0
tk                        8.6.12               h5d9f67b_0
wheel                     0.37.1              pyhd3eb1b0_0
xz                        5.2.5                hca72f7f_1
zlib                      1.2.12               h4dc903c_3
```

upgrade命令用于升级已经安装的conda包到不破坏依赖关系的最新版本，其用法与install命令一样。下面的例子将安装在example环境下的Python 3.8升级到Python 3.10（编写本书时Miniconda支持的最大Python版本）：

```
(example)$ conda upgrade python
```

注意这会自动调整CPython所依赖的其他conda包，包括升级、降级、安装和卸载。

注意我们无法用upgrade命令将某个conda包升级到指定的版本，这是install命令才能够完成的。然而用upgrade命令结合--all选项（此时不需要指定任何conda包）可以方便的整体升级指定的conda环境中的所有conda包，这是pip没有的功能：

```
(example)$ conda upgrade --all
```

需要强调的是，这样做并不会升级conda命令本身，因为它是安装在base环境下的，只能通过下面的方式升级：

```
(example)$ conda upgrade -n base conda
```

uninstall命令用于卸载已经安装的conda包。它同样需要用表17-25列出的选项指定作用于哪个conda环境，如果省略则作用于当前被激活的conda环境。与pip的uninstall不同，conda的uninstall不仅会卸载指定的conda包，还会自动找到卸载这些conda包后不再被需要的conda包并卸载它们。下面的例子从example环境卸载numpy以及不再需要的依赖：

```
(example)$ conda uninstall numpy
```

而下面的例子则会从example环境卸载Python 3.8以及不再需要的依赖：

```
(example)$ conda deactivate
$ conda uninstall -n example python
```

这会导致example环境只剩下与SSL/TLS相关的conda包：

```
$ conda list -n example
# packages in environment at /Users/wwy/opt/miniconda3/envs/example:
#
# Name                    Version                   Build  Channel
ca-certificates           2022.07.19           hecd8cb5_0
openssl                   1.1.1q               hca72f7f_0
```

以上就是使用conda的基本方式。然而我们还有一些快捷方法，例如在创建虚拟环境时同时指定安装哪些conda包：

```
$ conda create -n example1 python=3.9 mypy
```

以及先卸载所有的conda包，然后再删除conda环境本身：

```
$ conda uninstall -n example1 --all
```

对于create、install、upgrade和uninstall来说，都可以通过表17-26列出的选项来改变其默认执行过程。

需要强调的是，在用户创建的conda环境下安装conda包时，实际上是将conda包安装到base环境下，然后在其他conda环境中创建符号链接，这可以有效节省磁盘空间。然而从这些conda环境删除conda包时，通常也只会删除符号链接，导致base环境会逐渐积累不需要的conda包。所以我们需要周期性执行“conda clean --all”来清理这些conda包。这同时也会清理下载缓存、仓库索引和临时文件。

我们可以通过执行“conda info --base”取得Miniconda的基目录，通过执行“conda info --envs”了解系统中有多少个conda环境。

最后，我们可以通过conda安装conda-build，编写meta.yaml文件，然后将我们用任何编程语言编写的项目打包成一个conda包。我们还可以自己架设基于conda包的仓库，然后以它为可选频道。然而对它们的详细讨论超出了本书的范围。

本书到此结束，感谢你的阅读，祝好运！

---

[1] Mahmoud Hashemi, "The Many Layers of Packaging". https://sedimental.org/the_packaging_gradient.html.

[2] "Python Packaging Authority". https://www.pypa.io/en/latest/.

[3] "Anaconda". https://www.anaconda.com/.

[4] "Miniconda". https://docs.conda.io/en/latest/miniconda.html.

[5] "conda-forge". https://conda-forge.org/.

[6] PyPA. "Python Packaging User Guide". https://packaging.python.org/en/latest/.

[7] PyPA. "Project Summaries". https://packaging.python.org/en/latest/key_projects/.

[8] PyPA. "Installing pip/setuptools/wheel with Linux Package Managers". https://packaging.python.org/en/latest/guides/installing-using-linux-tools/.

[9] PyPA. "pip documentation". https://pip.pypa.io/en/stable/.

[10] "PEP 440 – Version Identification and Dependency Specification". https://peps.python.org/pep-0440/.

[11] "PEP 425 – Compatibility Tags for Built Distributions". https://peps.python.org/pep-0425/.

[12] PyPA. "Pipenv: Python Dev Workflow for Humans". https://pipenv.pypa.io/en/latest/.

[13] PyPA. "pipx — Install and Run Python Applications in Isolated Environments". https://pypa.github.io/pipx/.

[14] "PEP 345 – Metadata for Python Software Packages 1.2". https://peps.python.org/pep-0345/.

[15] "PEP 376 – Database of Installed Python Distributions". https://peps.python.org/pep-0376/.

[16] "PEP 427 – The Wheel Binary Package Format 1.0". https://peps.python.org/pep-0427/.

[17] "PEP 517 – A build-system independent format for source trees". https://peps.python.org/pep-0517/.

[18] "PEP 518 – Specifying Minimum Build System Requirements for Python Projects". https://peps.python.org/pep-0518/.

[19] Paul Ganssle. "Why you shouldn't invoke setup.py directly". https://blog.ganssle.io/articles/2021/10/setup-py-deprecated.html.

[20] "PEP 621 – Storing project metadata in pyproject.toml". https://peps.python.org/pep-0621/.

[21] PyPA. "Setuptools". https://setuptools.pypa.io/en/latest/.

[22] "PEP 503 – Simple Repository API". https://peps.python.org/pep-0503/.

[23] W3docs. "MIME-Types". https://www.w3docs.com/learn-html/mime-types.html.

[24] "PEP 301 – Package Index and Metadata for Distutils". https://peps.python.org/pep-0301/.

[25] PyPI. "Classifiers". https://pypi.org/classifiers/.

[26] "PEP 508 – Dependency specification for Python Software Packages". https://peps.python.org/pep-0508/.

[27] "TOML - Tom's Obvious Minimal Language". https://toml.io/en/.

[28] RFC 3339. "Date and Time on the Internet: Timestamps". https://www.rfc-editor.org/rfc/rfc3339.

[29] PyPA. "Declaring project metadata". https://packaging.python.org/en/latest/specifications/declaring-project-metadata/.

[30] GitHub. "Basic writing and formatting syntax". https://docs.github.com/cn/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax.

[31] GitHub. "Markdown and reStructuredText". https://gist.github.com/javiertejero/4585196.

[32] PyPA. "Including files in source distributions with MANIFEST.in". https://packaging.python.org/en/latest/guides/using-manifest-in/.

[33] PyPA. "build". https://pypa-build.readthedocs.io/en/stable/index.html.

[34] PyPA. "Twine". https://twine.readthedocs.io/en/latest/.

[35] PyPA. "The .pypirc file". https://packaging.python.org/en/latest/specifications/pypirc/.

[36] PyPA. "Publishing package distribution releases using GitHub Actions CI/CD workflows". https://packaging.python.org/en/latest/guides/publishing-package-distribution-releases-using-github-actions-ci-cd-workflows/.

[37] "tox automation project". https://pypi.org/project/tox/.

[38] "Welcome to Nox". https://nox.thea.codes/en/stable/index.html.

[39] "PEP 441 – Improving Python ZIP Application Support". https://peps.python.org/pep-0441/.

[40] "PEX". https://pypi.org/project/pex/.

[41] "What are the best tools for creating packaged executables for Python?" https://www.slant.co/topics/2668/~best-tools-for-creating-packaged-executables-for-python.

[42] "Welcome to cx_Freeze’ s documentation!". https://cx-freeze.readthedocs.io/en/latest/.

[43] "Nuitka User Manual". https://www.nuitka.net/doc/user-manual.html.

[44] "Conda". https://docs.conda.io/projects/conda/en/latest/.