Tomas Beuzen and Tiffany Timbers

Python Packages

To you, the Reader.

Never stop learning. You are capable of anything.

Contents

List of Tables

List of Figures

Preface

Python packages are the fundamental unit of shareable code in Python. Packages make it easy to reuse and maintain your code, and are how you share your code with your colleagues and the wider Python community. *Python Packages* is an open source textbook that describes modern and efficient workflows for creating Python packages. The focus of this text is overwhelmingly practical; we will demonstrate methods and tools you can use to develop and maintain packages quickly, reproducibly, and with as much automation as possible - so you can focus on writing and sharing code!

Why read this book?

Python packages are a core element of the Python programming language and are how you write reuseable and shareable code in Python. Despite their importance, packages can be difficult to understand and cumbersome to create for beginners and seasoned developers alike. This book aims to describe the packaging process at an accessible level for data scientists, developers and hobbyist programmers. Along the way, we'll walk through the development of a real Python package and we will explore all the key elements of Python packaging, including; package structure, when and why to write tests and documentation, and how to maintain and update your package with the help of automated CI/CD pipelines.

By reading this book, you will: - Understand what Python packages are, and when and why you should use them. - Be able to build your own Python package from scratch. - Learn how to document your Python code and packages, and how to render this documentation into a coherent, shareable document. - Write automated and formal tests for your code. - Learn how to release your package on the Python Package Index (PyPI) and discover best practices for updating and versioning your code. - Implement automation and CI/CD pipelines to build, test, and deploy your package and update its dependencies. - Get tips on Python coding style, best-practice packaging workflows, and other useful development tools.

Structure of the book

Chapter 1: Introduction first gives a brief introduction to packages in Python and why you should know how to make them. **Chapter 2: System setup** describes how to set up your development environment to develop packages and follow along with the remainder of the book. In **Chapter 3:** [**How to package a Python**], we will develop a small toy package from end-to-end to get a feel for the steps involved in the packaging process and to understand the final product we are aiming for. The remaining chapters then unpack this workflow and go into more details about each step in the packaging process, organised roughly in their order in the workflow:

- **Chapter 4:** [**Package structure and state**]
- **Chapter 5:** [**Testing**]
- **Chapter 6:** [**Documentation**]
- **Chapter 7:** [**Releasing and versioning**]
- **Chapter 8:** [**Continuous integration and deployment**]
- **Appendix 1:** [**Packages with a command line interface**]
- **Appendix 2:** [**Python packaging cheat sheet**]

Conventions

Throughout this book we use `foo()` to refer to functions, and `bar` to refer to variables and function parameters.

Larger code snippets in the text appear as below and we will use type hints and the so-called "Numpy-style" for docstrings:

```python
def palindrome(word: str) -> str:
    """Turns a string into a palindrome by concatenating it with a reversed version of itself.

    Parameters
    ----------
    word : str
        Word to turn into a palindrome.

    Returns
    -------
    str
        Palindrome of word.
    Examples
    --------
    >>> palindrome("Tomas")
    'TomassamoT'
    >>> palindrome("Python")
    'PythonnohtyP'
    """
    return word + word[::-1]
```

Code executed in the interactive Python interpreter to produce an output will appear as below:

```
a = 1
b = 2
a + b
```

```
## 3
```

If you have an electronic version of the book, e.g., `https://py-pkgs.org`, all code is rendered in such a way that you can easily copy and paste directly from your browser to your Python interpreter or editor.

Persistence

The Python software ecosystem is constantly evolving. While the packaging workflows and concepts discussed in this book are effectively tool-agnostic, the tools we do use in the book may have been updated by the time you read it. If the maintainers of these tools are doing the right thing by documenting, versioning, and properly deprecating their code (we'll explore these concepts ourselves in **Chapter 7:** [**Releasing and versioning**]), then it should be straightforward to adapt any outdated code in the book.

Colophon

This book was written in JupyterLab[1], compiled using Jupyter Book[2], is hosted on GitHub[3], and the online version[4] is deployed with Netlify[5]. The complete source is available from GitHub[6], and is automatically updated after edits by GitHub Actions[7].

Acknowledgements

We'd like to thank everyone that has contributed to the development of *Python Packages*[8]. This is an open source textbook that began as supplementary material for the University of British Columbia's Master of Data Science program and was subsequently developed openly on GitHub where it has been read, revised, and supported by many students, educators, practioners and hobbyists. Without you all, this book wouldn't be nearly as good as it is, and we are deeply grateful. A special thanks to those who directly contributed to the text via GitHub (in alphabetical order): `@Carreau`, `@dcslagel`.

We would also like to acknowledge the software used to develop this book. This book was written in JupyterLab[9], compiled using Jupyter Book[10], is hosted on GitHub[11], and the online version[12] is deployed with Netlify[13].

[1] https://jupyterlab.readthedocs.io/en/stable/index.html
[2] https://jupyterbook.org/intro.html
[3] https://github.com/
[4] https://py-pkgs.org/
[5] https://www.netlify.com/
[6] https://github.com/
[7] https://github.com/features/actions
[8] https://py-pkgs.org/
[9] https://jupyterlab.readthedocs.io/en/stable/index.html
[10] https://jupyterbook.org/intro.html
[11] https://github.com/
[12] https://py-pkgs.org/
[13] https://www.netlify.com/

About the authors

Tomas Beuzen is a Postdoctoral Teaching & Learning Fellow in the Department of Statistics at the University of British Columbia and a data science consultant. In these roles he teaches courses and provides expert advice about Python and R programming, data wrangling, and machine learning. Tomas has a background in coastal engineering and enjoys the practical application of data science to solving problems in the natural and engineered world. He is currently interested in developing open-source, educational data science material.

FIGURE 1: Tomas Beuzen.

Tiffany Timbers is an Assistant Professor of Teaching in the Department of Statistics and an Co-Director for the Master of Data Science program (Vancouver Option) at the University of British Columbia. In these roles she teaches and develops curriculum around the responsible application of Data Science to solve real-world problems. One of her favourite courses she teaches is a graduate course on collaborative software development, which focuses on teaching how to create R and Python packages using modern tools and workflows.

FIGURE 2: Tiffany Timbers.

1

Introduction

Python packages are a core element of the Python programming language and are how you write reuseable and shareable code in Python. If you're reading this book, chances are you already know how to use packages with the help of the `import` statement in Python. For example, importing and using the `numpy` package to round pi to 3 decimal places is as simple as:

```
import numpy as np

np.round(np.pi, decimals=3)
```

```
## 3.142
```

At a minimum, a package simply bundles together code (such as functions, classes, variables, or scripts) so that it can be easily reused across different projects. However, packages may also include things like documentation and tests, which become exponentially more important if you wish to share your package with others.

As of January 2021, there are over 280,000 packages available on the Python Package Index (PyPI). Packages are a key reason why Python is such a powerful and widely used programming language. The chances are that someone has already solved a problem that you're working on, and you can benefit from their work by downloading and installing their package (which they have kindly developed and shared) by, for example, using Python's native package manager `pip` and a simple `pip install <package-name>` at the command line. Put simply, packages are how you make it as easy as possible to share, maintain and collaborate on Python code with others; whether they be your friends, work colleagues, or the world!

Even if you never intend to share your code with others, making packages will ultimately save you time. Creating Python packages will make it significantly easier for you to access, reuse and maintain your code within a project and across different projects. At some point, all of us have wanted to reuse code from one project in another; this is something often accomplished through the reprehensible method of copy-and-pasting your existing code into the new project. Despite being obviously inefficient, this practice also makes it difficult to improve and maintain your code and its dependenices across projects. Creating a simple Python package will solve these problems.

Regardless of your motivation, the goal of this book is to show you how to easily develop Python packages. The focus is overwhelmingly practical - we will leverage modern methods and tools to develop and maintain packages efficiently, reproducibly, and with as much automation as possible; so you can focus on writing and sharing code. Along the way, we'll also enlighten some of the lower-level details of Python packaging and the Python programming language.

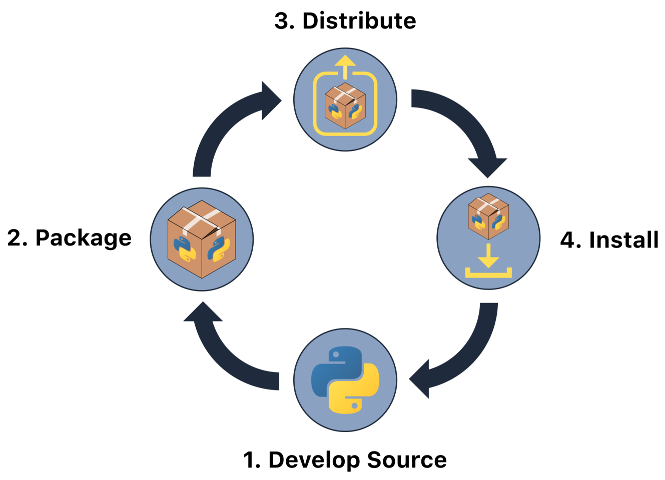


FIGURE 1.1: The Python packaging workflow.

1.1 Why you should create packages

As discussed above, there are many reasons why you should develop Python packages! Let's summarise the key reasons below:

- To effectively share your code with others.
- They save you time. Even if you don't intend to share your code, packages help you easily reuse and maintain your code across multiple projects.
- They force you to organise and document your code, such that it can be more easily understood and used at a later time.
- They isolate dependencies for your code and improve its reproducibility.
- They are a good way to practice writing good code.
- Finally, developing and distributing packages supports the Python ecosystem and other Python users who can benefit from your work.

2

System setup

If you intend to follow along with the code presented in this book, we recommend you follow these setup instructions so that you will run into fewer technical issues.

2.1 Installing Python

We recommend installing the latest version of Python via the Anaconda distribution by following the instructions in the Anaconda documentation[1]. Anaconda is a Python distribution that includes Python, the `conda` package and environment manager, and a number of commonly used Python libraries and dependencies. If you'd prefer to install packages as you need them, feel free to install Python via the Miniconda[2] distribution instead, which is a lightweight version of Anaconda that essentially comes with Python, `conda` and only a few key packages. If you have previosuly installed the Anaconda or Miniconda distribution, ensure that Python and `conda` are up to date by running the following command in your terminal:

```
conda update --all
```

If you're not familiar with `conda`, it is a piece of software that simplifies the process of installing and updating software (like Python packages). It is also an environment manager which is the key function we'll be using it for in this text. An environment manager helps you create "virtual environments" on your machine where you can safely install different packages and their dependencies in an isolated location. Installing all the packages you need in the same place (i.e., the system default location) can be problematic, because different packages have different dependencies, and as you install more packages, you'll inevitably get conflicts between packages and your code will start to break. Virtual environments help you compartmentalise and isolate the packages you are using for different projects to avoid this issue. While alternative package

[1] `https://docs.anaconda.com/anaconda/install/`
[2] `https://docs.conda.io/en/latest/miniconda.html`

and environment managers exist, we choose to use `conda` in this book because of its popularity, ease-of-use, and ability to handle any software stack (not just Python!).

Once you've installed the Anaconda distribution, use `conda` to install `poetry` (a package[3] that will help us build our own Python packages) and `cookiecutter` (a package[4] that will help us create packages from pre-made templates) with the following command:

```
conda install -c conda-forge poetry cookiecutter
```

2.2 Register for a PyPI account

PyPI is the official online software repository for Python. A software repository is a storage location for downloadable software, like Python packages. In this book we'll be publishing a package on PyPI. Before publishing packages on PyPI, it is typical to "test drive" their publication on TestPyPI which is a test version of PyPI. To follow along with this book, you should register for a TestPyPI account on the TestPyPI website[5] and a PyPI account on the PyPI website[6].

2.3 Install Git and register for a GitHub account

If you're not using a version control system, we highly recommend you get into the habit! A version control system tracks changes to the file(s) of your project in a clear and organised way (no more "document_1.doc", "document_1_new.doc", "document_final.doc", etc.). A VCS works by storing a master copy of your project in a "repository" which you don't edit directly. Instead, you edit a copy of the project file(s), and then *commit* changes back to the repository. In this way, a VCS contains a full history of all the revisions made to your project which you can view and retrieve at any time.

There are many version control systems available, but the most common is Git and we'll be using it throughout this book. You can download Git by

[3] https://python-poetry.org/
[4] https://github.com/cookiecutter/cookiecutter
[5] https://test.pypi.org/account/register/
[6] https://pypi.org/account/register/

following the instructions in the Git documentation[7]. Git will help us track changes to our project on our local computers, but what if we want to collaborate with others? Or, what happens if our computer crashes? That's where GitHub comes in. GitHub is one of many online services for hosting Git-managed projects. GitHub helps you create an online version of your local Git repository which acts as a back-up of your local work and allows others to easily and transparently collaborate on your project. You can sign up for a GitHub account on the GitHub website[8].

2.4 Python integrated development environments

A Python integrated development environment (IDE) will make the process of creating Python packages significantly easier. An IDE is a piece of software that provides advanced functionality for code development such as directory and file creation and navigation, code refactoring, autocomplete, debugging, and syntax highlighting, to name a few. Put simply, an IDE will save you time and help you write better code. Commonly used free Python IDEs include Visual Studio Code[9], Atom[10], Sublime Text[11], Spyder[12], and PyCharm Community Edition[13]. For those more familiar with the popular Jupyter ecosystem, JupyterLab[14] is an alternative browser-based IDE. Finally, for the R community, the RStudio IDE[15] also supports Python.

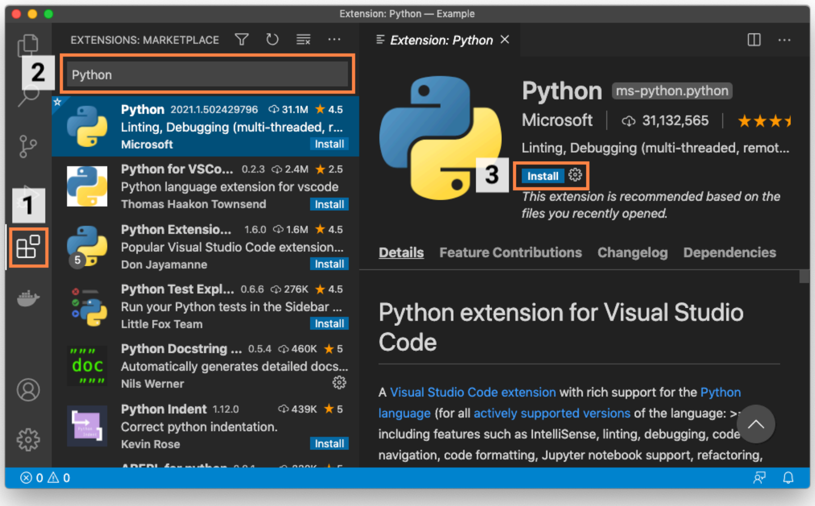
You'll be able to follow along with the examples presented in this book regardless of what IDE you choose to develop your Python code in. Below, we will briefly describe how to set up Visual Studio Code, JupyterLab, and RStudio as Python IDEs (these are the IDEs we personally use in our day-to-day work). If you don't know which IDE to use, we recommend starting with Visual Studio Code.

2.4.1 Visual Studio Code

You can download Visual Studio Code (VS Code) from the Visual Studio Code website[16]. Once you've installed VS Code, you should install the "Python"

[7] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
[8] https://www.github.com
[9] https://code.visualstudio.com/
[10] https://atom.io/
[11] https://www.sublimetext.com/
[12] https://www.spyder-ide.org/
[13] https://www.jetbrains.com/pycharm/
[14] https://jupyter.org/
[15] https://rstudio.com/products/rstudio/download/
[16] https://code.visualstudio.com/

extension from the VS Code *Marketplace*. To do this, you should: 1. Open the *Marketplace* by clicking the *Extensions* tab on the VS Code activity bar; 2. Search for "Python" in the search bar; 3. Select the extension named "Python" and then click *Install*.



Python 2021.1.502429796 31.1M ★ 4.5
Linting, Debugging (multi-threaded, r...
Microsoft Install

Python for VSCo... 0.2.3 2.4M ★ 2.5
Python language extension for vscode
Thomas Haakon Townsend Install

Python Extensio... 1.6.0 1.6M ★ 4.5
Popular Visual Studio Code extension...
Don Jayamanne Install

Python Test Expl... 0.6.6 276K ★ 4.5
Run your Python tests in the Sidebar ...
Little Fox Team Install

Python Docstring ... 0.5.4 460K ★ 5
Automatically generates detailed docs...
Nils Werner

Python Indent 1.12.0 439K ★ 5
Correct python indentation.
Kevin Rose Install

Linting, Debugging (multi-threaded, remot...

This extension is recommended based on the files you recently opened.

Python extension for Visual Studio Code

A Visual Studio Code extension with rich support for the Python language (for all actively supported versions of the language: >= including features such as IntelliSense, linting, debugging, code navigation, code formatting, Jupyter notebook support, refactoring,

FIGURE 2.1: Installing the Python extension in Visual Studio Code.

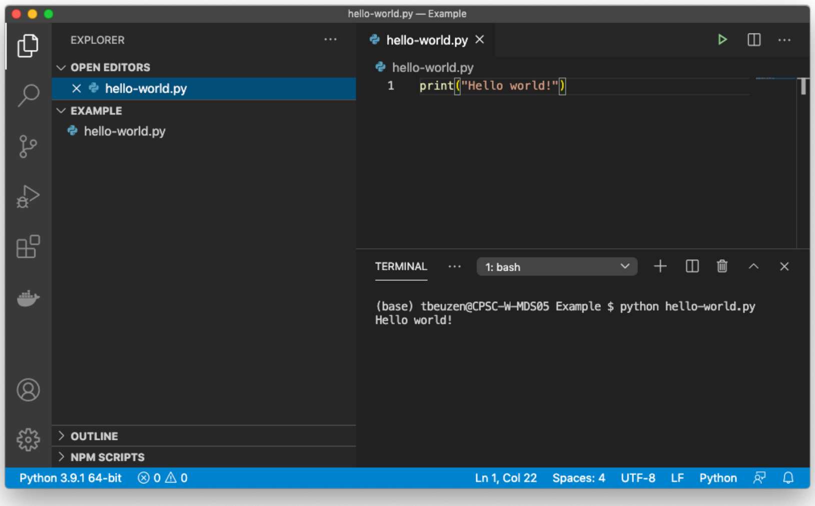
Once this is done, you have everything you need to start creating packages! For example, you can create files and directories from the *File Explorer* tab on the VS Code activity bar and you can open up an integrated terminal by clicking selecting *Terminal* from the *View* menu.

We recommend you take a look at the VS Code Getting Started Guide[17] to learn more about VS Code. While you don't need to install any additional extensions to start creating packages in VS Code, there are many extensions available that can support and streamline your programming workflows in VS Code. Here's a few we currently recommend installing (you can search for and install these from the *Marketplace* just as we did earlier): - Python Docstring Generator - Bracket Pair Colorizer 2 - Markdown All in One - markdownlint - reStructuredText - Material Icon Theme - Material Theme

2.4.2 JupyterLab

For those comfortable in the Jupyter ecosystem feel free to stay there. JupyterLab is a browser-based IDE that supports all of the core functionality we need

[17] https://code.visualstudio.com/docs



```python
print("Hello world!")
```

```
(base) tbeuzen@CPSC-W-MDS05 Example $ python hello-world.py
Hello world!
```

FIGURE 2.2: Executing a simple Python file called hello-world.py from the integrated terminal in Visual Studio Code.

to create packages. As per the JupyterLab installation instructions[18], you can install JupyterLab with:

```
conda install -c conda-forge jupyterlab
```

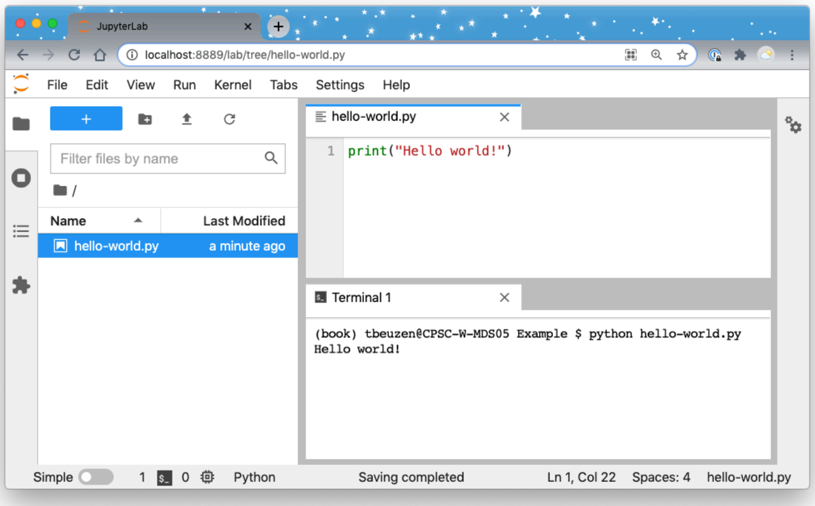
Once installed, you can launch JupyterLab from your current directory using:

```
jupyter-lab
```

At a minimum for package development, we need to be able to create files and directories and have access to a terminal. In JupyterLab, you can create files and directories from the *File Browser* and can open up an integrated terminal from the *File* menu or a *Launcher* pane.

We recommend you take a look at the JupyterLab documentation[19] to learn more about how to use Jupyterlab. In particular, we'll note that, like VS Code, JupyterLab also supports an ecosystem of extensions to support the IDE. We won't install any here, but you can browse them in the JupyterLab *Extension Manager* if you're interested.

[18] https://jupyterlab.readthedocs.io/en/stable/getting_started/installation.html
[19] https://jupyterlab.readthedocs.io/en/stable/index.html



```
print("Hello world!")
```

```
(book) tbeuzen@CPSC-W-MDS05 Example $ python hello-world.py
Hello world!
```

FIGURE 2.3: Executing a simple Python file called hello-world.py from a terminal in JupyterLab.

2.4.3 RStudio

Users with an R background may prefer to stay in the RStudio IDE. We recommend installing the most recent version of the IDE from the RStudio site[20] and then installing the most recent version of R from CRAN[21]. To use Python in RStudio, you will need to install the reticulate[22] R package by typing the following in the R console inside RStudio:

```
install.packages("reticulate")
```

When installing reticulate, you may be prompted to install the Anaconda distribution. If you have already installed the Anaconda distribution of Python (which we did earlier), answer "no" to installing Anaconda distribution at this prompt. Depending on your operating system and how your Python and R environments are set up, you may need to configure `reticulate` in different ways. The `reticulate` documentation[23] can help you with this set up.

[20] https://rstudio.com/products/rstudio/download/preview/
[21] https://cran.r-project.org/
[22] https://rstudio.github.io/reticulate/
[23] https://rstudio.github.io/reticulate/

2.4.4 Collaborative package development and packaging in the cloud

Git and GitHub make it easy for others to collaborate on your Python packages, and we will use these tools in this book. In this workflow, collaborators create a copy of a package's code from GitHub on their local machine, make changes to it, and then "push" those changes back to the remote GitHub repository. This is an effective way of making and tracking changes to a code base over time in a collaborative environment.

There also exists other options for collaborative software development that allow collaborators to directly edit source code (without having to make a copy of it), potentially simultaneously. Such a workflow might be particularly suited to collaborators that want to develop together in real-time, leverage cloud resources, or keep their work in private servers/hardware. Some current options for this kind of collaborative software development include: - Visual Studio Live Share[24]: real-time collaborative development in VS Code. - Google Colab[25]: Jupyter notebooks that can be shared with collaborators via Google Drive. Collaborators can then view, edit or comment on these notebooks. While Python .py files can't be be created in Google Colab, they can be shared via Google Drive and accessed in the Google Colab environment. - JupyterHub[26]: facilitates groups of users to interact within a shared Jupyter ecosystem that runs on private hardware or the cloud. A key advantage of JupyterHub is that it gives users access to collaborative computational environments without burdening the users with software installation.

[24]https://visualstudio.microsoft.com/services/live-share/
[25]https://colab.research.google.com/
[26]https://jupyter.org/hub