2 About Python

Python is currently the most popular programming language among scientists and other programmers. There are a number of reasons leading to its popularity and fame, especially among younger researchers. This chapter introduces the Python programming language and provides an overview on how to install and use the language most efficiently.

2.1 What Is Python?

Python is a general-purpose programming language that is extremely versatile and relatively easy to learn. It is considered a high-level programming language, meaning that the user typically will not have to deal with some typical housekeeping tasks when designing code. This is different from other (especially compiled) languages that heavily rely on the user to do these tasks properly. Python is designed in such a way as to help the user to write easily readable code by following simple guidelines. But Python also implements powerful programming paradigms: it can be used as an object-oriented, procedural, and functional programming language, depending on your needs and use case. Thus Python combines the simplicity of a scripting language with advanced concepts that are typically characteristic for compiled languages. Some of these features – which we will introduce in detail in Chapter 3 – include dynamic typing, built-in object types and other tools, automatic memory management and garbage collection, as well as the availability of a plethora of add-on and third-party packages for a wide range of use cases. Despite its apparent simplicity, these features make Python a very competitive, powerful, and flexible programming language.

Most importantly, Python is open-source and as such freely available to everyone. We detail in Section 2.2 how to obtain and install Python on your computer.

Based on various recent reports and statistics, Python is currently the most popular programming language among researchers and professional software developers for a wide range of applications and problems. This popularity largely stems from the ease of learning Python, as well as the availability of a large number of add-on packages that supplement basic Python and provide easy access to tasks that would otherwise be cumbersome to implement.

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

But there is also a downside: Python is an interpreted language, which makes it slower than compiled languages. However, Python provides some remedies for this issue as we will see in Chapter 9.

For researchers, Python offers a large range of well-maintained open-source packages, many of which are related to or at least based on the SciPy ecosystem. SciPy contains packages for scientific computing, mathematics, and engineering applications. Despite being the backbone of many Python applications, SciPy is completely open-source and funded in some part through NumFocus, a nonprofit organization supporting the development of scientific Python packages. We will get to know some of the packages that are part of the SciPy universe in Chapters 4, 5, and 8.

2.1.1 A Brief History of Python

The Python programming language was conceived by Guido van Rossum, a Dutch computer scientist, in the 1980s. He started the implementation in 1989 as a hobby project over the Christmas holidays. The first release became available in 1991 and Python 1.0 was released in 1994; Python 2.0 became available in 2000. With a growing user base, the development team also started to grow and gradually all the features that we appreciate about this language were implemented. Python 3.0 was released in 2008, which broke the backwards compatibility with Python 2.x due to some design decisions. The existence of two versions of Python that were incompatible with each other generated some confusion, especially with inexperienced users. However, support for Python 2.x ended in 2020, leaving Python 3.x as the only supported version of Python. The example code shown in this book and the accompanying Jupyter Notebooks (see Section 2.4.2) are based on Python version 3.9.12, but this should not matter as future versions should be compatible with that one.

Van Rossum is considered the principal author of Python and has played a central role in its development until 2018. Since 2001, the Python Software Foundation, a nonprofit organization focusing on the development of the core Python distribution, managing intellectual rights, and organizing developer conferences, has played an increasingly important role in the project. Major design decisions within the project are made by a five-person steering council and documented in Python Enhancement Protocols (PEPs). PEPs mainly discuss technical proposals and decisions, but we will briefly look at two PEPs that directly affect users: the Zen of Python (PEP 20, Section 2.1.2) and the Python Style Guide (PEP 8, Section 3.13).

We would also like to note that in 2012, NumFOCUS was founded as a nonprofit organization that supports the development of a wide range of scientific Python packages including, but not limited to, NumPy (see Chapter 4), SciPy (see Chapter 5), Matplotlib (see Chapter 6), SymPy (see Chapter 7), Pandas (see Chapter 8), Project Jupyter, and IPython. The support through NumFOCUS for these projects includes funding that is based on donations to NumFOCUS; for most of these open-source projects, donations are their only source of funding.

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

One detail we have skipped so far is why Van Rossum named his new programming language after a snake. Well, he did not. Python is actually named after the BBC comedy TV show *Monty Python's Flying Circus*, of which Van Rossum is a huge fan. In case you were wondering, this is also the reason why the words "spam" and "eggs" are oftentimes used as metasyntactic variables in Python example code in a reference to their famous "Spam" sketch from 1970.

2.1.2 The Zen of Python

The Zen of Python is an attempt to summarize Van Rossum's guiding principles for the design of Python into 20 aphorisms, only 19 of which have been written down. These guiding principles are very concise and distill many features of Python into a few words. The Zen of Python is so important that it is actually published (PEP 20) and its content is literally built into the Python language and can be accessed as follows:

```
import this
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one -- and preferably only one -- obvious way to
    do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Please note that these guidelines focus on the design of the Python programming language, not necessarily the design of code written in Python. Nevertheless, you are free to follow these guidelines when writing your own code to create truly *pythonic* code. The term *pythonic* is often used within the Python community to refer to code that follows the guiding principles mentioned here.

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

These guiding principles are numerous and some of them might not be immediately clear to the reader, especially if you are new to Python programming. We would summarize the most important Python concepts as follows.

Simplicity Simple code is easier to write and read; it improves readability, shareability, and maintainability, and therefore helps you and others in the short term and long term.

Readability It is nice to write code as compact as possible, but if writing compact code requires some tricks that are hard to understand, you might prefer a more extensive implementation that provides better readability. Why? Imagine that your future self tries to modify some code that you wrote years ago. If your code is well-readable, you will probably have fewer problems understanding what the individual lines of code do.

Explicitness We will explain this idea with an example. Consider you are writing code that is able to read data from different file formats. A decision you have to make is the following: will you create a single function that is able to read all the different file formats, or do you create a number of individual functions, each of which is able to read only a single file format? The *pythonic* way would be the latter: each function that you create will explicitly be able to deal with only a single file format in contrast to a single function that implicitly deals with all file formats. Why is this solution favored? Generally, explicit code is easier to understand and less prone to confusion.

Naturally, these concepts are entangled and closely related to each other. However, there is no need to memorize these concepts. You will internalize those concepts that are relevant to you by writing code and reading code written by others. And, of course, nobody can force you to follow these principles in your own coding; but we hope that this section provides you a better understanding of the Python programming language and its design.

2.2 Installing Python

Depending on the operating system you are using, there are several ways to install Python on your computer, some of which are simpler than others. The easiest and at the same time safest way to install Python is to use the Anaconda environment as detailed below.

Alternatively, you can also install Python from scratch on your computer – unless it is already installed. In the latter case, you should be careful not to interfere with the native Python already installed as it might be required by your operating system. This process might be a bit more complicated, but there are detailed installation guides for all operating systems available online. To be on the safe side, we recommend the installation of Anaconda, which comes with Conda, a tool to set up and utilize virtual environments,

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

in order to prevent interference with other versions of Python that might be installed on your computer. Once Python is installed, additional packages can also be installed using Conda and the *Package installer for Python*, pip.

2.2.1 Anaconda and Conda

Anaconda is a Python distribution package for data science and machine learning applications. Despite this specialization, the Anaconda Individual Edition (also known as the "Anaconda Distribution") constitutes a solid basis for any scientific Python installation.

The Anaconda Distribution is provided and maintained by Anaconda Inc. (previously known as Continuum Analytics). Despite being a for-profit company, Anaconda Inc. distributes the Anaconda Individual Edition for free.

Installing Anaconda

Installing Anaconda is simple and straightforward. All that is required is to download the respective Anaconda Individual Edition installer (see Section 2.6) for your operating system and run it. The installer will walk you through the installation process. Note that you will need to agree to the Anaconda license agreement. At the end of the installation routine, you will be asked whether to make Anaconda Python your default Python version, which is a good idea in most cases. If you now start the Python interpreter (see Section 2.4.1), you will be greeted by Anaconda Python. Congratulations, you have successfully installed Anaconda Python on your computer.

Conda

One advantage of using Anaconda is the availability of Conda, an open-source package and environment manager that was originally developed by Anaconda Inc., but has subsequently been released separately under an open-source license. Although, for a beginner, the simple installation process for Anaconda Python is most likely its most important feature, Conda also solves two problems in the background. As a package manager, it allows you to easily install Python packages with a single command on your command line, e.g.,

```
conda install numpy
```

Almost all major Python packages are available through Conda. Packages are available through Conda-Forge (see Section 2.6), a GitHub (see Section 10.3.1) organization that contains repositories of "Conda recipes" for a wide range of packages. Conda-Forge contains more detailed information on how to install packages through Conda, as well as a list of all packages that are available through Conda.

As an environment manager, Conda allows you to define different environments, each of which can have its own Python installation. Although this is an advanced feature and becomes important when you are dealing with specific versions of your Python packages, there is still some benefit for the Python beginner. Some operating systems use

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

native Python installation to run crucial services; meddling with these Python installations can seriously harm your system. By default, Anaconda creates a *base* environment for the user. Since this environment is independent from your system, there is no danger in meddling with your system Python installation. Thus using Anaconda is safer than using your system Python installation.

It is not complicated to define new Conda environments and to switch between them. However, due to the advanced nature of dealing with different environments, we refer to the Conda documentation to learn more about how to do this.

2.2.2 Pip and PyPI

Pretty much all Python packages are registered with the *Python Package Index*, PyPI, which enables the easy distribution of these packages. Installing packages from PyPI is very easy using the pip package manager, which comes with most Python installations, e.g.,

```
pip install numpy
```

Everybody can publish their code via PyPI; in Section 10.3.2 we will show how this can be achieved. Since PyPI is the official repository of Python packages, pretty much all available packages are installable using pip.

Pip or Conda?

After learning about Conda and pip you might be confused which of these tools you should use to install Python packages. The short answer is, in most cases it does not matter. Especially for beginners, it is perfectly fine and typically also safe to install packages using pip. Pip is typically faster than Conda in installing packages.

This faster installation process comes at a (small) price that won't matter to most users. The price is that Conda is generally safer in installing new packages. Before Conda installs a new package, it will check the version numbers of all packages that are already installed in your current Conda environment and it will check whether these packages in the present versions are compatible with the new package and vice versa. Pip simply checks whether the versions of the installed packages are compatible with the new package – and it will update the already present packages, to make them compatible with the new package. However, pip disregards that there might be requirements by other packages that will break by updating these existing packages. As a result, pip may break packages that were previously installed.

This happens very rarely, since most Python packages are compatible over many different versions. However, in the case of quickly developing projects it is mandatory to use specific versions of packages. In those cases, it is much safer to use Conda to install new packages. For most other users, especially on the beginner level, there should be no major issues.

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

2.3 How Python Works

In Chapter 1, we already introduced compiled and interpreted programming languages. As a brief reminder, compiled languages take the code written by the user in some high-level programming language and translate it into machine-readable code that is written to an executable file. Interpreted languages, on the other hand, do not require the high-level code provided by the user to be compiled. Instead, the *interpreter* reads the code in chunks and translates them sequentially into some less-basic kind of machine-readable *bytecode* that is directly executed. As you can imagine, compiled languages perform faster than interpreted languages, since the *compiler* already does the hard work to translate user code to efficient machine-readable code, whereas an interpreter has to do this on the fly in a less efficient way.

The following sections will detail how to directly provide code to the interpreter in different ways.

2.4 How to Use Python

There are different ways to use Python, the most important of which we will introduce in the following sections. Which of these options you should use depends on your preferences and the problem you are trying to solve.

In the remainder of this book, we assume that you are using Jupyter Notebooks. This choice is mainly driven by the opportunity to publish all code elements from this book as readily accessible Jupyter Notebooks. You can run these Notebooks (as well as your own Notebooks) online in the cloud, or locally on your computer as detailed below. However, we would like to point out that it is not a requirement for the reader to use these Notebooks in order to follow this book in any way. Feel free to use whichever interface to Python you feel most comfortable with.

2.4.1 The Python Interpreter

The easiest way to use Python is to run its interpreter in interactive mode. On most operating systems, this is done by simply typing `python` into a terminal or powershell window. Once started, you can type Python commands and statements into the interpreter, which are then executed line by line (or block by block if you use indentation).

While this might be useful to quickly try something out, it is not really suited to write long scripts or other more or less complex pieces of code. The interpreter also provides only a bare minimum in terms of support and usability.

The Python interpreter also offers a different way to run Python code that is much better suited for running longer pieces of code. Instead of writing your code line by line into the interpreter, you can simply write your code into an ordinary text file and pass that

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

file to the interpreter in your terminal window or on the command line. You can give your code file any name you want, but by convention, you should use the file ending "`.py`." You can use the most basic text editor for this purpose: Emacs, Vim, Nano or Gedit on Linux, TextEdit or Sublime on a Mac, or NotePad on Windows. It is important that the resulting Python code file does not contain any fancy formatting, just clean text.

For example, you could create a file named "`hello.py`" with the following single line of content:

```
print('Hello World!')
```

You can then run this script in a terminal window or powershell by using

```
python hello.py
```

Make sure that Python is properly installed on your system (see Section 2.2) and that you run this command in the same directory where the `hello.py` file resides. If successful, the output that you receive should look like this:

```
Hello World!
```

And this is your first Python program!

2.4.2 IPython and Jupyter

IPython (Interactive Python) is an architecture for interactive computing with Python: it can be considered as the Python interpreter on steroids. The IPython interpreter has been designed and written by scientists with the aim of offering very fast exploration and construction of code with minimal typing effort, and offering appropriate, even maximal, on-screen help when required. It further supports introspection (the ability to examine the properties of any Python object at runtime), tab completion (autocompletion support during typing when hitting the Tab key), history (IPython stores commands that are entered and their results, both of which can be accessed at runtime), as well as support for parallel computing. Most importantly, IPython includes a browser-based Notebook interface with a visually appealing notebook-like appearance.

The first version of IPython was published in 2001. Project Jupyter evolved from IPython around 2014 as a nonprofit, open-source project to support interactive data science and scientific computing. The Notebook interface was subsequently outsourced from IPython and implemented as part of Jupyter, where it was perfected and extended in different ways. Most notably, Jupyter Notebooks are language agnostic and can be used with different programming languages using so-called kernels. The Python kernel is provided and still maintained by the IPython project.

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

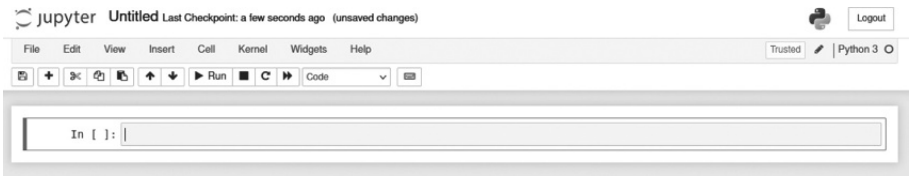

Figure 2.1 A newly created Jupyter Notebook containing a single, empty code cell.

The following sections introduce the most important features of Jupyter.

Jupyter Notebooks

The most relevant feature of Jupyter for you will most likely be the **Jupyter Notebook**, which is an enhanced version of the IPython Notebook. All programming examples are presented in the form of Jupyter Notebooks and imitate their appearance (see Figure 2.1). Furthermore, all code elements shown in this book are available as Jupyter Notebooks online at CoCalc, and also at www.cambridge.org/9781009014809.

Jupyter Notebooks are documents that consist of distinct cells that can contain and run code, formatted text, mathematical equations, and other media. Notebooks are run in your browser through a server that is either hosted locally on your computer or in the cloud (see Section 2.4.4).

To start a Jupyter Notebook server locally, you simply have to run

```
jupyter notebook
```

in a terminal window or powershell. This will run a server in the background that is typically accessible through `http://localhost:8888` (you need to type this into the URL field of your browser to access the Notebook server). You will see a list of files and directories located in the directory you started the server from. From here you can navigate your file system, and open existing Jupyter Notebooks or create new ones. Notebook files use the filename ending `.ipynb`, indicating that they are using the IPython kernel. To open a Notebook, simply click on the file and you will see something that looks like Figure 2.1.

Cloud services hosting Jupyter Notebook servers are a different avenue that allow you to utilize Notebooks without the (minor) hassle of having to install the necessary software on your computer. As a result, Notebooks that run on cloud services (see Section 2.4.4) might look a little bit different to what is shown in Figure 2.1, but rest assured that they can be used in the same way as described in this book.

Notebooks consist of *cells* that are either **code cells** that contain Python code or **markdown cells** that contain text or other media utilizing the markdown language. **Markdown** is a lightweight markup language (pun intended) that enables you to quickly format text and even supports LaTeX inline math. A markdown cheat sheet containing some formatting basics is provided in Table 2.1 for your convenience; for more

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

Table 2.1 Jupyter Notebook markdown cheat sheet

Markdown	Appearance

`# Large Headline`
`### Medium Headline`
`##### Small Headline`
	# Large Headline
	### Medium Headline
	##### Small Headline

`normal text`
`*emphasized/italics* text`
`**bold** text`
`‘simple one-line code sample‘`
	normal text
	emphasized/italics text
	bold text
	`simple one-line code sample`

‘‘‘python
`print('multi-line syntax')`
`print('highlighting', '!')`
‘‘‘
	`print('multi-line syntax')`
	`print('highlighting', '!')`

`* unordered list item 1`
`* unordered list item 2`
`1. ordered list item 1`
`2. ordered list item 2`
	• unordered list item 1
	• unordered list item 2
	1. ordered list item 1
	2. ordered list item 2

`[https://www.python.org](link)`	link
`![alt text][https://...image.jpg]` `![alt text][/path/to/image.jpg]`	insert image from url insert image from file
`$\frac{1}{2} x^2 = \int x\, dx$`	$\frac{1}{2}x^2 = \int x\,dx$

information on how to use the markdown language, please consult your favorite internet search engine.

To run any cell, i.e., to run the code in a code cell or to render the text in a markdown cell, you can click the corresponding “run” button or simply use the keyboard shortcut `shift + enter`. If your code cell generates output, this output will be displayed underneath the code cell. Note that each executed code cell will be numbered (e.g., `[1]` for the code cell that was executed first) at the beginning of that cell and that the corresponding output will carry the same number. This number is stored in the history of the Notebook and can be utilized and indexed during runtime. Note that in the case of markdown cells, the raw input that you provided is simply replaced by the rendered text upon execution.

For as long as a Notebook is open and running, the memory is shared between all cells. That means that if you define an object in one cell and execute that cell, you can access that object from any other cell. This also means that if you change the object in one cell, its state changes in all other cells, too. Thus the order in which cells are executed might be important.

There is no rule for how many lines of code should go into a single code cell. When testing code or performing explorative data analysis, you might end up with a number of single-line code cells. If you develop large-scale numerical models, you might end

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

up with cells containing hundreds of lines of code. The same applies to the number of cells in a Notebook.

We encourage the reader to take full advantage of the features of a Notebook: combine code cells and markdown cells in such a way as to treat the Notebook as a self-explanatory document that contains runnable Python code.

While code cells generally expect to receive valid Python commands, they can also execute commands on the underlying operating system's terminal or command line environment. **Magic commands** provide the user with a way to interact with the operating system and file system from within a Jupyter Notebook. A very long, very detailed description of magic commands can be found by typing `%magic`, and a compact list of available commands is given by typing `%lsmagic`. Note that there are two types of magic: **line magic** and **cell magic**. Line magic commands are prefixed by `%` and operate only on a single line of a cell. Cell magic, on the other hand, is prefixed by `%%` and operates on the entire cell; cell magic should always appear at the beginning of a cell.

A harmless line magic example is `pwd`, which comes from the Unix operating system and prints the name of the current directory (**p**resent **w**orking **d**irectory). With magic, `pwd` can be called by invoking

```
%pwd
```

An example for a cell magic command is `%%timeit`, which we introduce in Section 9.1.1 to accurately measure the time it takes to run a specific cell.

Magic commands provide a useful set of commands, but this set is limited. There is also the possibility to execute commands directly on the operating system level without having to leave a running Notebook environment. This can be achieved in code cells by prepending an exclamation mark (`!`). For instance, you can use this mechanism to install missing Python packages from within a Notebook:

```
!pip install <package name>
```

(see Section 2.2.2 for an introduction on pip).

Within this book, we display Notebook code cells as follows:

```
This is a code cell.
```

The output of a cell is displayed differently:

```
This is the code cell's output.
```

Please be aware that the output that you might receive on your computer may differ from the output provided in this book. This is especially true for code elements that rely on random numbers, but also other examples. Finally, please be aware that we had to edit

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

the output provided by Python manually in a few cases to have it displayed properly in this book.

JupyterLab

JupyterLab is an advanced version of the Jupyter Notebook. It provides you with an interface to arrange multiple files – e.g., documents, Jupyter Notebooks, text editors, consoles – on a single screen. The idea behind the JupyterLab environment is to support data analysis by combining different tools into a single environment. To start a JupyterLab server locally, all you need to do is to run

```
jupyter lab
```

in your Linux or Mac terminal or your Windows powershell. The combination of Notebooks and data visualization tools makes JupyterLabs powerful for tasks involving the analysis of data and other tasks. We encourage readers to experiment with this system, but we will not require its use in the following.

JupyterHub

JupyterHub runs a multi-user server for Jupyter Notebooks. This means that multiple users can log into a server to run and share Notebooks. Some research institutes maintain their own JupyterHub to provide their researchers a collaborative work environment. A detailed discussion of JupyterHub is outside the scope of this book, but we would like the reader to be aware of its existence.

2.4.3 Integrated Development Environments

If you prefer a more sophisticated environment for coding, you should have a look at integrated development environments (IDEs), which support you in your software development endeavors by providing online help, checking your syntax on the fly, highlighting relevant code elements, integrating with and supporting version control software, providing professional debugging tools, and many other things.

A wide range of IDEs for Python is available. Here we briefly introduce a small number of freely available open-source IDEs that run on all major operating systems (Linux, Mac OS, and Windows).

A very simple IDE for beginners is **Thonny**. One feature of Thonny that might appeal to beginners is that it comes with an option to install its own Python interpreter; the user will not have to install Python themselves (although this is of course still possible). Furthermore, Thonny provides features that will help you code in Python, better understand your code, and find mistakes in your code during typing.

Spyder is a much more advanced IDE that is tailored to scientific applications with a focus on data science. Spyder is written in Python and for Python. It comes with many features of professional IDEs, like a debugger, and it allows you to work with Jupyter Notebooks.

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

PyCharm and **VSCode** (Visual Studio Code) are two rather professional IDEs providing all of the aforementioned features plus a wide range of plugins for a variety of use cases. While VSCode, although provided by Microsoft, is completely free of charge, PyCharm has two different versions: a free-to-use Community Edition that comes with all the bells and whistles for Python programming, and a Professional Version that is not free, but comes with additional support for the development of scientific software and web applications.

Finding the right IDE that fits your needs is mostly a matter of taste, habit, and expectations. Feel free to try all of these IDEs and pick the one that suits your needs. Be aware that especially the more professional environments will typically feel less comfortable in the beginning and that it takes some time to get used to setting them up and working with them. However, at some point, you will get used to them and enjoy some of their more advanced features.

However, always keep in mind that there is no requirement to use an IDE to become a good programmer. There are plenty of people out there that write excellent and extremely complex code, using simple text editors like Vim or Emacs (even those can be customized into very efficient programming tools by installing a few extensions) or Jupyter Notebooks. Our bottom line is this: feel free to use whatever tool you feel most comfortable with!

2.4.4 Cloud Environments

Finally, we would like to point out that it is possible to run Jupyter Notebooks in cloud environments. For instance, all Notebooks utilized in this book are available online and can be run on different cloud services. Free computing is available through a number of providers; we would like to point out three examples: Binder, CoCalc, and Google Colab. **Binder** enables you to run Notebooks hosted in Github repositories (see Section 10.3.1) and does, as of writing this, not require any form of registration or user authentication. **CoCalc** provides similar functionality; all Jupyter Notebooks related to this book are hosted at CoCalc. **Google Colab** requires registration with Google services; usage is free to some reasonable extent. The advantage of Colab is its integration in the Google services environment (e.g., it is possible to connect to Google Drive for storing Notebooks and data files) and the option to ask for additional computational resources like GPU support at no charge (as of this writing).

2.5 Where to Find Help?

Before we get started on actual programming with Python, we would like to share a few words on how to get help when you are stuck with an issue. First of all, do not panic: there are many ways for you to get help, depending on your situation.

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

In case you are unsure about how to use a function or method, or you are trying to find the right function or method for your purpose, you can consult the corresponding **Reference**, which describes its syntax and semantics in detail. The Python Language Reference describes the syntax and core semantics of the language. On the other hand, the Python Standard Library contains details on the workings of the built-in functionality of Python. Both references are important and you might want to browse them to get an idea of their content and utility. In addition to these resources, each major Python package has its own reference document. As a practical example, let us consider the `math` package reference that is part of the Python Standard Library. For each function in the `math` module, the reference provides a detailed *docstring* (see Section 3.1) that describes the function's general functionality as well as its arguments. For instance, the *docstring* of `math.perm()` looks like this:

```
math.perm(n, k=None)

Return the number of ways to choose k items from n items without
    repetition and with order.

Evaluates to n! / (n - k)! when k <= n and evaluates to zero when k >
     n.

If k is not specified or is None, then k defaults to n and the
    function returns n!.

Raises TypeError if either of the arguments are not integers. Raises
    ValueError if either of the arguments are negative.

New in version 3.8.
```

`math.perm` takes two arguments: a (required) positional argument (see Section 3.8.2), `n`, and an (optional) keyword argument (see Section 3.8.3), `k`. The *docstring* defines the functionality of the function and what it returns. Furthermore, it contains information on the exceptions (see Section 3.10.2) that it may raise and a note about when it was implemented into the `math` module. Based on this information – and after reading the next chapter of this book – it should be straightforward to utilize this function. References can be accessed online by utilizing your favorite search engine, by using the `help()` function in your Python interpreter, or through your favorite IDE.

If you prefer looking things up in printed books instead of browsing the Internet, you may, of course, also refer to literature (see, e.g., Section 3.14). The advantage here is that function descriptions might be less technical and easier to understand – but on the downside, these descriptions might be incomplete. Nevertheless, literature is definitely a good resource if you are looking for help.

Even if you are perfectly sure about how to use a function, errors may occur. When an error occurs during runtime, Python will tell you about it. It will not only tell you

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

on which line of code what type of error occurred, but also how the program reached that point in your code (this is called the *traceback*; see Section 3.10.1 for more details on this). The latter might sound trivial, but it is actually very useful when dealing with highly modular programs that consist of hundreds or thousands of lines of code.

Every once in a while, every programmer will encounter a problem that they cannot solve without help from others. A perfectly legitimate approach to solving this problem would be to search for a solution on the Internet. An excellent resource, and likely the most common website to pop up as a result of search engine queries, is *StackOverflow*, which is used by beginners and professional programmers alike. You can ask questions on *StackOverflow*, but it is more than likely that the specific problem that you encountered has already been addressed and answered by the community and can therefore be found with most internet search engines. For instance, the last line of your *traceback* (the actual error message; see Section 3.10.1) would be a good candidate to enter into a search engine, potentially leading to a number of cases in which other coders experienced similar issues and presumably were able to solve them. While this sounds trivial, this process of finding the solution to a problem online should be in no way stigmatized. On the contrary, we encourage users in this process, since the potential to learn from others cannot be underestimated.

2.6 References

Online resources

- Python project resources

  Python project homepage
  `https://python.org`

  Python Developer's Guide
  `https://python.org/dev/`

  Python PEP Index
  `https://python.org/dev/peps`

  NumFOCUS project homepage
  `https://numfocus.org`

  NumPy project homepage
  `https://numpy.org`

  SciPy project homepage
  `https://scipy.org`

  Matplotlib project homepage
  `https://matplotlib.org`

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

SymPy project homepage
`https://sympy.org`

Pandas project homepage
`https://pandas.pydata.org`

Jupyter project homepage
`https://jupyter.org`

IPython project homepage
`https://ipython.org`

- Installation resources

  Anaconda homepage
  `https://anaconda.com/`

  Anaconda Individual Edition download homepage
  `https://anaconda.com/products/individual`

  Conda project homepage
  `https://conda.io`

  Conda-Forge project homepage
  `https://conda-forge.org/`

  Conda documentation
  `https://docs.conda.io/`

  Python Package Index (PyPI) homepage
  `https://pypi.org/`

- Integrated development environments

  Thonny
  `https://thonny.org/`

  Spyder
  `https://spyder-ide.org/`

  PyCharm
  `https://jetbrains.com/pycharm/`

  Visual Studio Code
  `https://spyder-ide.org/`

- Cloud environments

  Binder
  `https://mybinder.org/`

  CoCalc
  `https://cocalc.com/`

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press

Google Colab
https://colab.research.google.com/

- Finding help

Python Standard Library
https://docs.python.org/3/library/

Python Language Reference
https://docs.python.org/3/reference/

StackOverflow
https://stackoverflow.com/

https://doi.org/10.1017/9781009029728.003 Published online by Cambridge University Press