Chapter 1. Introduction to Python

Python is a well-established general-purpose programming language, first released by its creator, Guido van Rossum, in 1991. This stable and mature language is high-level, dynamic, object-oriented, and cross-platform—all very attractive characteristics. Python runs on macOS, most current Unix variants including Linux, Windows, and, with some tweaks, mobile platforms.[1]

Python offers high productivity for all phases of the software life cycle: analysis, design, prototyping, coding, testing, debugging, tuning, documentation, and, of course, maintenance. The language's popularity has seen steadily increasing growth for many years, becoming the **TIOBE index** leader in October 2021. Today, familiarity with Python is a plus for every programmer: it has snuck into most niches, with useful roles to play in any software solution.

Python provides a unique mix of elegance, simplicity, practicality, and sheer power. You'll quickly become productive with Python, thanks to its consistency and regularity, its rich standard library, and the many third-party packages and tools that are readily available for it. Python is easy to learn, so it is quite suitable if you are new to programming, yet is also powerful enough for the most sophisticated expert.

The Python Language

The Python language, while not minimalist, is spare, for good pragmatic reasons. Once a language offers one good way to express a design, adding other ways has, at best, modest benefits; the cost of language complexity, though, grows more than linearly with the number of features. A complicated language is harder to learn and master (and to implement efficiently and without bugs) than a simpler one. Complications and quirks in a language hamper productivity in software development, particularly in large projects, where many developers cooperate and, often, maintain code originally written by others.

Python is fairly simple, but not simplistic. It adheres to the idea that, if a language behaves a certain way in some contexts, it should ideally work similarly in all contexts. Python follows the principle that a language should not have "convenient" shortcuts, special cases, ad hoc exceptions, overly subtle distinctions, or mysterious and tricky under-the-covers optimizations. A good language, like any other well-designed artifact, must balance general principles with taste, common sense, and a lot of practicality.

Python is a general-purpose programming language: its traits are useful in almost any area of software development. There is no area where Python cannot be part of a solution. "Part" is important here; while many developers find that Python fills all of their needs, it does not have to stand alone. Python programs can cooperate with a variety of other software components, making it the right language for gluing together components in other languages. A design goal of the language is, and has long been, to "play well with others."

Python is a very high-level language (VHLL). This means that it uses a higher level of abstraction, conceptually further away from the underlying machine, than classic compiled languages such as C, C++, and Rust, traditionally called "high-level languages." Python is simpler, faster to process (both for humans and for tools), and more regular than classic high-level languages. This affords high programmer productivity, making Python a strong development tool. Good compilers for classic compiled languages can generate binary code that runs faster than Python. In most cases, however, the performance of Python-coded applications is sufficient. When it isn't, apply the optimization techniques covered in **"Optimization"** to improve your program's performance while keeping the benefit of high productivity.

In terms of language level, Python is comparable to other powerful VHLLs like JavaScript, Ruby, and Perl. The advantages of simplicity and regularity, however, remain on Python's side.

Python is an object-oriented programming language, but it lets you program in both object-oriented and procedural styles, with a touch of functional programming too, mixing and matching as your application requires. Python's object-oriented features are conceptually similar to those of C++ but simpler to use.

The Python Standard Library and Extension Modules

There is more to Python programming than just the language: the standard library and other extension modules are almost as important for Python use as the language itself. The Python standard library supplies many well-designed, solid Python modules for convenient reuse. It includes modules for such tasks as representing data, processing text, interacting with the operating system and filesystem, and web programming, and works on all platforms supported by Python.

Extension modules, from the standard library or elsewhere, let Python code access functionality supplied by the underlying operating system or other software components, such as graphical user interfaces (GUIs), databases, and networks. Extensions also afford great speed in computationally intensive tasks such as XML parsing and numeric array computations. Extension modules that are not coded in Python, however, do not necessarily enjoy the same cross-platform portability as pure Python code.

You can write extension modules in lower-level languages to optimize performance for small, computationally intensive parts that you originally prototyped in Python. You can also use tools such as Cython, ctypes, and CFFI to wrap existing C/C++ libraries into Python extension modules, as covered in "Extending Python Without Python's C API" in **Chapter 25** (available **online**). You can also embed Python in applications coded in other languages, exposing application functionality to Python via app-specific Python extension modules.

This book documents many modules, from the standard library and other sources, for client- and server-side network programming, databases, processing text and binary files, and interacting with operating systems.

Python Implementations

At the time of this writing, Python has two full production-quality implementations (CPython and PyPy) and several newer, high-performance ones in somewhat earlier stages of development, such as **Nuitka**, **RustPython**, **GraalVM Python**, and **Pyston**, which we do not cover further. In **"Other Developments, Implementations, and Distributions"** we also mention some other, even earlier-stage implementations.

This book primarily addresses CPython, the most widely used implementation, which we often call just "Python" for simplicity. However, the distinction between a language and its implementations is important!

CPython

Classic Python—also known as CPython, often just called Python—is the most up-to-date, solid, and complete production-quality implementation of Python. It is the "reference implementation" of the language. CPython is a bytecode compiler, interpreter, and set of built-in and optional modules, all coded in standard C.

CPython can be used on any platform where the C compiler complies with the ISO/IEC 9899:1990 standard[2] (i.e., all modern, popular platforms). In **"Installation"**, we explain how to download and install CPython. All of this book, except a few sections explicitly marked otherwise, applies to CPython. As of this writing, CPython's current version, just released, is 3.11.

PyPy

PyPy is a fast and flexible implementation of Python, coded in a subset of Python itself, able to target several lower-level languages and virtual machines using advanced techniques such as type inferencing. PyPy's greatest strength is its ability to generate native machine code "just in time" as it runs your Python program; it has substantial advantages in execution speed. PyPy currently implements 3.8 (with 3.9 in beta).

Choosing Between CPython, PyPy, and Other Implementations

If your platform, as most are, is able to run CPython, PyPy, and several of the other Python implementations we mention, how do you choose among them? First of all, don't choose prematurely: download and install them all. They coexist without problems, and they're all free (some of them also offer commercial versions with added value such as tech support, but the respective free versions are fine, too). Having them all on your development machine costs only some download time and a little disk space, and lets you compare them directly. That said, here are a few general tips.

If you need a custom version of Python, or high performance for long-running programs, consider PyPy (or, if you're OK with versions that are not quite production-ready yet, one of the others we mention).

To work mostly in a traditional environment, CPython is an excellent fit. If you don't have a strong alternative preference, start with the standard CPython reference implementation, which is most widely supported by third-party add-ons and extensions and offers the most up-to-date version.

In other words, to experiment, learn, and try things out, use CPython. To develop and deploy, your best choice depends on the extension modules you want to use and how you want to distribute your programs. CPython, by definition, supports all Python extensions; however, PyPy supports most extensions, and it can often be faster for long-running programs thanks to just-in-time compilation to machine code—to check on that, benchmark your CPython code against PyPy (and, to be sure, other implementations as well).

CPython is most mature: it has been around longer, while PyPy (and the others) are newer and less proven in the field. The development of CPython versions proceeds ahead of that of other implementations.

PyPy, CPython, and other implementations we mention are all good, faithful implementations of Python, reasonably close to each other in terms of usability and performance. It is wise to become familiar with the strengths and weaknesses of each, and then choose optimally for each development task.

Other Developments, Implementations, and Distributions

Python has become so popular that several groups and individuals have taken an interest in its development and have provided features and implementations outside the core development team's focus.

Nowadays, most Unix-based systems include Python—typically version 3.*x* for some value of *x*—as the "system Python." To get Python on Windows or macOS, you usually download and run an **installer** (see also **"macOS"**.) If you are serious about software development in Python, the first thing you should do is *leave your system-installed Python alone!* Quite apart from anything else, Python is increasingly used by some parts of the operating system itself, so tweaking the Python installation could lead to trouble.

Thus, even if your system comes with a "system Python," consider installing one or more Python implementations to freely use for your development convenience, safe in the knowledge that nothing you do will affect the operating system. We also strongly recommend the use of *virtual environments* (see **"Python Environments"**) to isolate projects from each other, letting them have what might otherwise be conflicting dependencies (e.g., if two of your projects require different versions of the same third-party module). Alternatively, it is possible to locally install multiple Pythons side by side.

Python's popularity has led to the creation of many active communities, and the language's ecosystem is very active. The following sections outline some of the more interesting developments: note that our failure to include a project here reflects limitations of space and time, rather than implying any disapproval!

Jython and IronPython

Jython, supporting Python on top of a **JVM**, and **IronPython**, supporting Python on top of **.NET**, are open source projects that, while offering production-level quality for the Python versions they support, appear to be "stalled" at the time of this writing, since the latest versions they support are substantially behind CPython's. Any "stalled" open source project could, potentially, come back to life again: all it takes is one or more enthusiastic, committed developers to devote themselves to "reviving" it. As an alternative to Jython for the JVM, you might also consider GraalVM Python, mentioned earlier.

Numba

Numba is an open source just-in-time (JIT) compiler that translates a subset of Python and NumPy. Given its strong focus on numeric processing, we mention it again in **Chapter 16**.

Pyjion

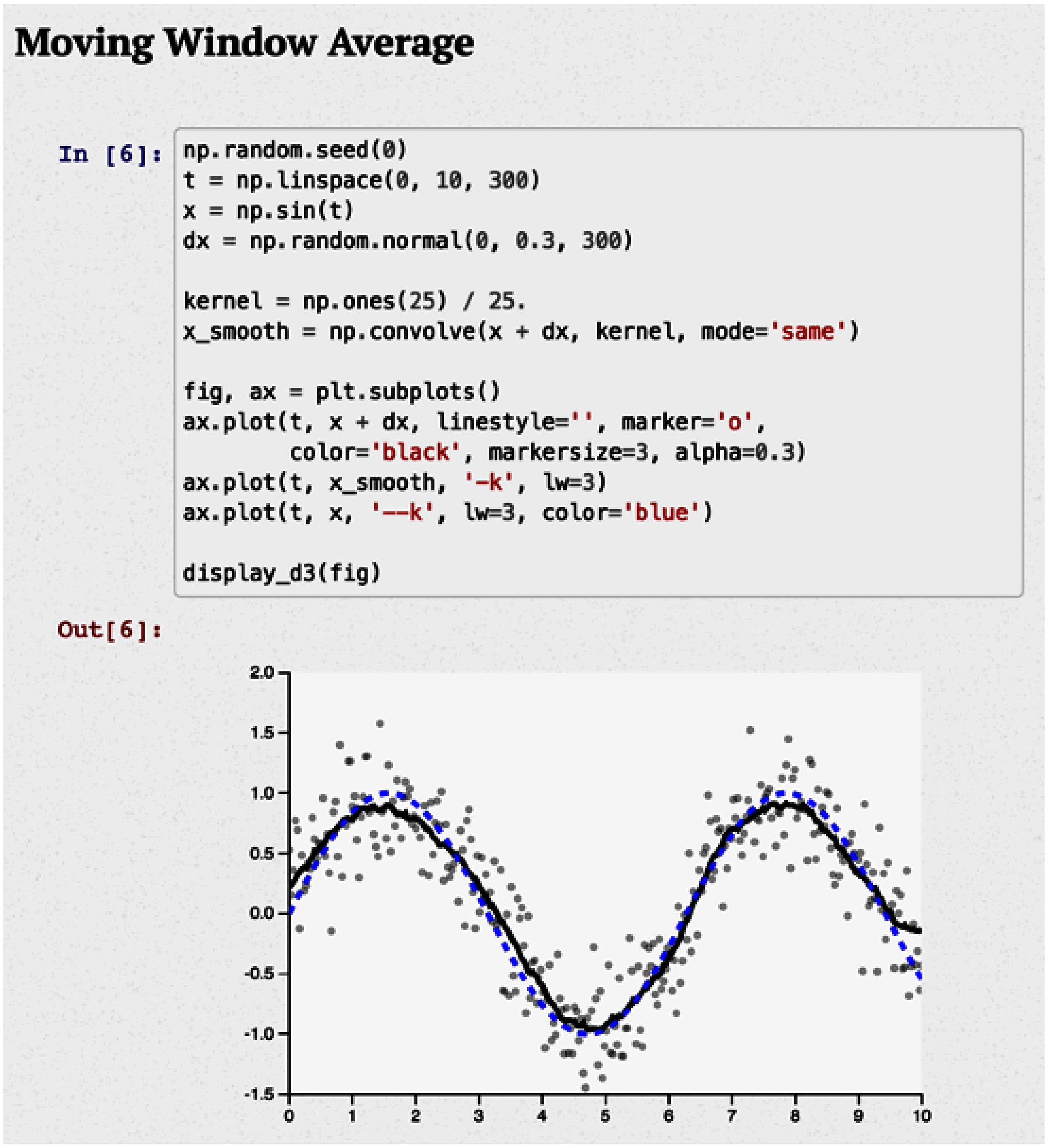
Pyjion is an open source project, originally started by Microsoft, with the key goal of adding an API to CPython to manage JIT compilers. Secondary goals include offering a JIT compiler for Microsoft's open source **CLR** environment (which is part of .NET) and a framework to develop JIT compilers. Pyjion does not *replace* CPython; rather, it is a module that you import from CPython (it currently requires 3.10) that lets you translate CPython's bytecode, "just in time," into machine code for several different environments. Integration of Pyjion with CPython is enabled by **PEP 523**;

however, since building Pyjion requires several tools in addition to a C compiler (which is all it takes to build CPython), the Python Software Foundation (PSF) will likely never bundle Pyjion into the CPython releases it distributes.

IPython

IPython enhances CPython's interactive interpreter to make it more powerful and convenient. It allows abbreviated function call syntax, and extensible functionality known as *magics* introduced by the percent (`%`) character. It also provides shell escapes, allowing a Python variable to receive the result of a shell command. You can use a question mark to query an object's documentation (or two question marks for extended documentation); all the standard features of the Python interactive interpreter are also available.

IPython has made particular strides in the scientific and data-focused world, and has slowly morphed (through the development of IPython Notebook, now refactored and renamed Jupyter Notebook, discussed in **"Jupyter"**) into an interactive programming environment that, among snippets of code,[3] also lets you embed commentary in **literate programming** style (including mathematical notation) and show the output of executing code, optionally with advanced graphics produced by such subsystems as `matplotlib` and `bokeh`. An example of `matplotlib` graphics embedded in a Jupyter Notebook is shown in the bottom half of **Figure 1-1**. Jupyter/IPython is one of Python's prominent success stories.



Moving Window Average

In [6]:
```
np.random.seed(0)
t = np.linspace(0, 10, 300)
x = np.sin(t)
dx = np.random.normal(0, 0.3, 300)

kernel = np.ones(25) / 25.
x_smooth = np.convolve(x + dx, kernel, mode='same')

fig, ax = plt.subplots()
ax.plot(t, x + dx, linestyle='', marker='o',
        color='black', markersize=3, alpha=0.3)
ax.plot(t, x_smooth, '-k', lw=3)
ax.plot(t, x, '--k', lw=3, color='blue')

display_d3(fig)
```

Out[6]:

Figure 1-1. An example Jupyter Notebook with embedded matplotlib graph

MicroPython

The continued trend in miniaturization has brought Python well within the range of the hobbyist. Single-board computers like the **Raspberry Pi** and **Beagle boards** let you run Python in a full Linux environment. Below this level, there is a class of devices known as *microcontrollers*—programmable chips with configurable hardware—that extend the scope of hobby and professional projects, for example by making analog and digital sensing easy, enabling such applications as light and temperature measurements with little additional hardware.

Both hobbyists and professional engineers are making increasing use of these devices, which appear (and sometimes disappear) all the time.

Thanks to the **MicroPython** project, the rich functionality of **many such devices** (**micro:bit**, **Arduino**, **pyboard**, **LEGO® MINDSTORMS® EV3**, **HiFive**, etc.) can now be programmed in (limited dialects of) Python. Of note at the time of writing is the introduction of the **Raspberry Pi Pico**. Given the success of the Raspberry Pi in the education world, and Pico's ability to run MicroPython, it seems that Python is consolidating its position as the programming language with the broadest range of applications.

MicroPython is a Python 3.4 implementation ("with selected features from later versions," to quote **its docs**) producing bytecode or executable machine code (many users will be happily unaware of the latter fact). It fully implements Python 3.4's syntax, but lacks most of the standard library. Special hardware driver modules let you control various parts of built-in hardware; access to Python's socket library lets devices interact with network services. External devices and timer events can trigger code. Thanks to MicroPython, the Python language can fully play in the Internet of Things.

A device typically offers interpreter access through a USB serial port, or through a browser **using the WebREPL protocol** (we aren't aware of any fully working `ssh` implementations yet, though, so, take care to firewall these devices properly: *they should* **not** *be directly accessible across the internet without proper, strong precautions!*). You can program the device's power-on bootstrap sequence in Python by creating a *boot.py* file in the device's memory, and this file can execute arbitrary MicroPython code of any complexity.

Anaconda and Miniconda

One of the most successful Python distributions[4] in recent years is **Anaconda**. This open source package comes with a vast number[5] of preconfigured and tested extension modules in addition to the standard library. In many cases, you might find that it contains all the necessary dependencies for your work. If your dependencies aren't supported, you can also install modules with `pip`. On Unix-based systems, it installs very simply in a single directory: to activate it, just add the Anaconda *bin* subdirectory at the front of your shell `PATH`.

Anaconda is based on a packaging technology called `conda`. A sister implementation, **Miniconda**, gives access to the same extensions but does not come with them preloaded; it instead downloads them as required, making it a better choice for creating tailored environments. `conda` does not use the standard virtual environments, but contains equivalent facilities to allow separation of the dependencies for multiple projects.

pyenv: Simple support for multiple versions

The basic purpose of `pyenv` is to make it easy to access as many different versions of Python as you need. It does so by installing so-called *shim* scripts for each executable, which dynamically compute the version required by looking at various sources of information in the following order:

1. The `PYENV_VERSION` environment variable (if set).
2. The *.pyenv_version* file in the current directory (if present)—you can set this with the **`pyenv local`** command.
3. The first *.pyenv_version* file found when climbing the directory tree (if one is found).
4. The *version* file in the `pyenv` installation root directory—you can set this with the **`pyenv global`** command.

`pyenv` installs its Python interpreters underneath its home directory (normally *~/.pyenv*), and, once available, a specific interpreter can be installed as the default Python in any project directory. Alternatively (e.g., when testing code under multiple versions), you can use scripting to change the interpreter dynamically as the script proceeds.

The **`pyenv install -list`** command shows an impressive list of over 500 supported distributions, including PyPy, Miniconda, MicroPython, and several others, plus every official CPython implementation from 2.1.3 to (at the time of writing) 3.11.0rc1.

Transcrypt: Convert your Python to JavaScript

Many attempts have been made to make Python into a browser-based language, but JavaScript's hold has been tenacious. The **Transcrypt** system is a `pip`-installable Python package to convert Python code (currently, up to version 3.9) into browser-executable JavaScript. You have full access to the browser's DOM, allowing your code to dynamically manipulate window content and use JavaScript libraries.

Although it creates minified code, Transcrypt provides full **sourcemaps** that allow you to debug with reference to the Python source rather than the generated JavaScript. You can write browser event handlers in Python, mixing it freely with HTML and JavaScript. Python may never replace JavaScript as the embedded browser language, but Transcrypt means you might no longer need to worry about that.

Another very active project that lets you script your web pages with Python (up to 3.10) is **Brython**, and there are others yet: **Skulpt**, not quite up to Python 3 yet but moving in that direction; **PyPy.js**, ditto; **Pyodide**, currently supporting Python 3.10 and many scientific extensions, and centered on **Wasm**; and, most recently, Anaconda's **PyScript**, built on top of Pyodide. We describe several of these projects in more detail in **"Running Python in the Browser"**.

Licensing and Price Issues

CPython is covered by the **Python Software Foundation License Version 2**, which is GNU Public License (GPL) compatible but lets you use Python for any proprietary, free, or other open source software development, similar to BSD/Apache/MIT licenses. Licenses for PyPy and other implementations are similarly liberal. Anything you download from the main Python and PyPy sites won't cost you a penny. Further, these licenses do not constrain what licensing and pricing conditions you can use for software you develop using the tools, libraries, and documentation they cover.

However, not everything Python-related is free from licensing costs or hassles. Many third-party Python sources, tools, and extension modules that you can freely download have liberal licenses, similar to that of Python itself. Others are covered by the GPL or Lesser GPL (LGPL), constraining the licensing conditions you can place on derived works. Some commercially developed modules and tools may require you to pay a fee, either unconditionally or if you use them for profit.[6]

There is no substitute for careful examination of licensing conditions and prices. Before you invest time and energy into any software tool or component, check that you can live with its license. Often, especially in a corporate environment, such legal matters may involve consulting lawyers. Modules and tools covered in this book, unless we explicitly say otherwise, can be taken to be, at the time of this writing, freely downloadable, open source, and covered by a liberal license akin to Python's. However, we claim no legal expertise, and licenses can change over time, so double-checking is always prudent.

Python Development and Versions

Python is developed, maintained, and released by a team of core developers led by Guido van Rossum, Python's inventor, architect, and now "ex" Benevolent Dictator for Life (BDFL). This title meant that Guido had the final say on what became part of the Python language and standard library. Once Guido decided to retire as BDFL, his decision-making role was taken over by a small "Steering Council," elected for yearly terms by PSF members.

Python's intellectual property is vested in the PSF, a nonprofit corporation devoted to promoting Python, described in **"Python Software Foundation"**. Many PSF Fellows and members have commit privileges to Python's **reference source repositories**, as documented in the **"Python Developer's Guide"**, and most Python committers are members or Fellows of the PSF.

Proposed changes to Python are detailed in public docs called **Python Enhancement Proposals (PEPs)**. PEPs are debated by Python developers and the wider Python community, and finally approved or rejected by the Steering Council. (The Steering Council may take debates and preliminary votes into account but are not bound by them.) Hundreds of people contribute to Python development through PEPs, discussion, bug reports, and patches to Python sources, libraries, and docs.

The Python core team releases minor versions of Python (3.*x* for growing values of *x*), also known as "feature releases," currently at a pace of **once a year**.

Each minor release (as opposed to bug-fix microreleases) adds features that make Python more powerful, but also takes care to maintain backward compatibility. Python 3.0, which was allowed to break backward compatibility in order to remove redundant "legacy" features and simplify the language, was first released in December 2008. Python 3.11 (the most recent stable version at the time of publication) was first released in October 2022.

Each minor release 3.*x* is first made available in alpha releases, tagged as 3.*x*a0, 3.*x*a1, and so on. After the alphas comes at least one beta release, 3.*x*b1, and after the betas, at least one release candidate, 3.*x*rc1. By the time the final release of 3.*x* (3.*x*.0) comes out, it is solid, reliable, and tested on all major platforms. Any Python programmer can help ensure this by downloading alphas, betas, and release candidates, trying them out, and filing bug reports for any problems that emerge.

Once a minor release is out, part of the attention of the core team switches to the next minor release. However, a minor release normally gets successive point releases (i.e., 3.*x*.1, 3.*x*.2, and so on), one every two months, that add no functionality but can fix errors, address security issues, port Python to new platforms, enhance documentation, and add tools and (100% backward compatible!) optimizations.

Python's backward compatibility is fairly good within major releases. You can find code and documentation **online** for all old releases of Python, and the **Appendix** contains a summary list of changes in each of the releases covered in this book.

Python Resources

The richest Python resource is the web: start at Python's **home page**, which is full of links to explore.

Documentation

Both CPython and PyPy come with good documentation. You can read CPython's manuals **online** (we often refer to these as "the online docs"), and various downloadable formats suitable for offline viewing, searching, and printing are also available. The Python **documentation page** contains additional pointers to a large variety of other documents. There is also a **documentation page** for PyPy, and you can find online FAQs for both **Python** and **PyPy**.

Python documentation for nonprogrammers

Most Python documentation (including this book) assumes some software development knowledge. However, Python is quite suitable for first-time programmers, so there are exceptions to this rule. Good introductory, free online texts for nonprogrammers include:

- Josh Cogliati's **"Non-Programmers Tutorial for Python 3"** (currently centered on Python 3.9)
- Alan Gauld's **"Learning to Program"** (currently centered on Python 3.6)
- Allen Downey's ***Think Python*, 2nd edition** (centered on an unspecified version of Python 3.*x*)

An excellent resource for learning Python (for nonprogrammers, and for less experienced programmers too) is the **"Beginners' Guide to Python" wiki**, which includes a wealth of links and advice. It's community curated, so it will stay up-to-date as available books, courses, tools, and so on keep evolving and improving.

Extension modules and Python sources

A good starting point to explore Python extension binaries and sources is the **Python Package Index** (still fondly known to a few of us old-timers as "The Cheese Shop," but generally referred to now as PyPI), which at the time of this writing offers more than 400,000 packages, each with descriptions and pointers.

The standard Python source distribution contains excellent Python source code in the standard library and in the *Tools* directory, as well as C source for the many built-in extension modules. Even if you have no interest in building Python from source, we suggest you download and unpack the Python source distribution (e.g., the latest stable release of **Python 3.11**) for the sole purpose of studying it; or, if you so choose, peruse the current bleeding-edge version of Python's standard library **online**.

Many Python modules and tools covered in this book also have dedicated sites. We include references to such sites in the appropriate chapters.

Books

Although the web is a rich source of information, books still have their place (if you didn't agree with us on this, we wouldn't have written this book, and you wouldn't be reading it). Books about Python are numerous. Here are a few we recommend (some cover older Python 3 versions, rather than current ones):

- If you know some programming but are just starting to learn Python, and you like graphical approaches to instruction, ***Head First Python, 2nd edition***, by Paul Barry (O'Reilly) may serve you well. Like all the books in the Head First series, it uses graphics and humor to teach its subject.
- ***Dive Into Python 3***, by Mark Pilgrim (Apress), teaches by example in a fast-paced and thorough way that is quite suitable for people who are already expert programmers in other languages.
- ***Beginning Python: From Novice to Professional***, by Magnus Lie Hetland (Apress), teaches both via thorough explanations and by fully developing complete programs in various application areas.

- ***Fluent Python***, by Luciano Ramalho (O'Reilly), is an excellent book for more experienced developers who want to use more Pythonic idioms and features.

Community

One of the greatest strengths of Python is its robust, friendly, welcoming community. Python programmers and contributors meet at conferences, "hackathons" (often known as ***sprints*** in the Python community), and local user groups; actively discuss shared interests; and help each other on mailing lists and social media. For a complete list of ways to connect, visit ***https://www.python.org/community***.

Python Software Foundation

Besides holding the intellectual property rights for the Python programming language, the PSF promotes the Python community. It sponsors user groups, conferences, and sprints, and provides grants for development, outreach, and education, among other activities. The PSF has dozens of **Fellows** (nominated for their contributions to Python, including all of the Python core team, as well as three of the authors of this book); hundreds of members who contribute time, work, and money (including many who've earned **Community Service Awards**); and dozens of **corporate sponsors**. Anyone who uses and supports Python can become a member of the PSF.[7] Check out the **membership page** for information on the various membership levels, and on how to become a member of the PSF. If you're interested in contributing to Python itself, see the **"Python Developer's Guide"**.

Workgroups

Workgroups are committees established by the PSF to do specific, important projects for Python. Here are some examples of active workgroups at the time of writing:

- The **Python Packaging Authority (PyPA)** improves and maintains the Python packaging ecosystem and publishes the **"Python**

Packaging User Guide”.
- The **Python Education workgroup** promotes education and learning with Python.
- The **Diversity and Inclusion workgroup** supports and facilitates the growth of a diverse and international community of Python programmers.

Python conferences

There are lots of Python conferences worldwide. General Python conferences include international and regional ones, such as **PyCon** and **EuroPython**, and other more local ones such as **PyOhio** and **PyCon Italia**. Topical conferences include **SciPy** and **PyData**. Conferences are often followed by coding sprints, where Python contributors get together for several days of coding focused on particular open source projects and abundant camaraderie. You can find a listing of conferences on the Community **Conferences and Workshops page**. More than 17,000 videos of talks about Python, from more than 450 conferences, are available at the **PyVideo site**.

User groups and organizations

The Python community has local user groups on every continent except Antarctica[8]—more than 1,600 of them, according to the list on the **LocalUserGroups wiki**. There are Python **meetups** around the world. **PyLadies** is an international mentorship group, with local chapters, to promote women in Python; anyone with an interest in Python is welcome. **NumFOCUS**, a nonprofit charity promoting open practices in research, data, and scientific computing, sponsors the PyData conference and other projects.

Mailing lists

The Community **Mailing Lists page** has links to several Python-related mailing lists (and some Usenet groups, for those of us old enough to remember **Usenet**!). Alternatively, search **Mailman** to find active mailing lists covering a wide variety of interests. Python-related official announcements are posted to the **python-announce list**. To ask for help with specific problems, write to ***help@python.org***. For help learning or teaching Python, write to ***tutor@python.org***, or, better yet, join the **list**. For a useful weekly roundup of Python-related news and articles, subscribe to **Python Weekly**. You can also follow Python Weekly at *@python_discussions@mastodon.social*.

Social media

For an **RSS feed** of Python-related blogs, see **Planet Python**. If you're interested in tracking language developments, check out ***discuss.python.org***—it sends useful summaries if you don't visit regularly. On Twitter, follow @ThePSF. **Libera.Chat** on **IRC** hosts several Python-related channels: the main one is `#python`. **LinkedIn** has many Python groups, including **Python Web Developers**. On Slack, join the **PySlackers** community. On Discord, check out **Python Discord**. Technical questions and answers about Python programming can also be found and followed on **Stack Overflow** under a variety of tags, including **[python]**. Python is currently the **most active** programming language on Stack Overflow, and many useful answers with illuminating discussions can be found there. If you like podcasts, check out Python podcasts, such as **Python Bytes**.

Installation

You can install the classic (CPython) and PyPy versions of Python on most platforms. With a suitable development system (C for CPython; PyPy, coded in Python itself, only needs CPython installed first), you can install Python versions from the respective source code distributions. On popular platforms, you also have the recommended alternative of installing prebuilt binary distributions.

INSTALLING PYTHON IF IT COMES PREINSTALLED

If your platform comes with a preinstalled version of Python, you're still best advised to install a separate up-to-date version for your own code development. When you do, do *not* remove or overwrite your platform's original version: rather, install the new version alongside the first one. This way, you won't disturb any other software that is part of your platform: such software might rely on the specific Python version that came with the platform itself.

Installing CPython from a binary distribution is faster, saves you substantial work on some platforms, and is the only possibility if you have no suitable C compiler. Installing from source code gives you more control and flexibility, and is a must if you can't find a suitable prebuilt binary distribution for your platform. Even if you install from binaries, it's best to also download the source distribution, since it can include examples, demos, and tools that are usually missing from prebuilt binaries. We'll look at how to do both next.

Installing Python from Binaries

If your platform is popular and current, you'll easily find prebuilt, packaged binary versions of Python ready for installation. Binary packages are typically self-installing, either directly as executable programs or via appropriate system tools, such as the Red Hat Package Manager (RPM) on some versions of Linux, and the Microsoft Installer (MSI) on Windows. After downloading a package, install it by running the program and choosing installation parameters, such as the directory where Python is to be installed. In Windows, select the option labeled "Add Python 3.10 to PATH" to have the installer add the install location into the PATH in order to easily use Python at a command prompt (see **"The python Program"**).

You can get the "official" binaries from the **Downloads page** on the Python website: click the button labeled "Download Python 3.11.x" to download the most recent binary suitable for your browser's platform.

Many third parties supply free binary Python installers for other platforms. Installers exist for Linux distributions, whether your distribution is **RPM-based** (Red Hat, Fedora, Mandriva, SUSE, etc.) or **Debian-based** (including Ubuntu, probably the most popular Linux distribution at the time of this writing). The **Other Platforms page** provides links to binary distributions for now somewhat exotic platforms such as AIX, OS/2, RISC OS, IBM AS/400, Solaris, HP-UX, and so forth (often not the latest Python versions, given the now "quaint" nature of such platforms), as well as one for the very current **iOS platform**, the operating system of the popular **iPhone** and **iPad** devices.

Anaconda, mentioned earlier in this chapter, is a binary distribution including Python, plus the **conda** package manager, plus hundreds of third-party extensions, particularly for science, math, engineering, and data analysis. It's available for Linux, Windows, and macOS. **Miniconda**, also mentioned earlier in this chapter, is the same package but without all of those extensions; you can selectively install subsets of them with `conda`.

MACOS

The popular third-party macOS open source package manager **Homebrew** offers, among many other open source packages, excellent versions of **Python**. `conda`, mentioned in **"Anaconda and Miniconda"**, also works well in macOS.

Installing Python from Source Code

To install CPython from source code, you need a platform with an ISO-compliant C compiler and tools such as `make`. On Windows, the normal way to build Python is with Visual Studio (ideally **VS 2022**, currently available to developers **for free**).

To download the Python source code, visit the **Python Source Releases** page (on the Python website, hover over Downloads in the menu bar and select "Source code") and choose your version.

The file under the link labeled “Gzipped source tarball” has a *.tgz* file extension; this is equivalent to *.tar.gz* (i.e., a *tar* archive of files, compressed by the popular `gzip` compressor). Alternatively, you can use the link labeled “XZ compressed source tarball” to get a version with an extension of *.tar.xz* instead of *.tgz*, compressed with the even more powerful `xz` compressor, if you have all the needed tools to deal with XZ compression.

Microsoft Windows

On Windows, installing Python from source code can be a chore unless you are familiar with Visual Studio and used to working in the text-oriented window known as the *command prompt*[9]—most Windows users prefer to simply download the prebuilt **Python from the Microsoft Store**.

If the following instructions give you any trouble, stick with installing Python from binaries, as described in the previous section. It’s best to do a separate installation from binaries anyway, even if you also install from source. If you notice anything strange while using the version you installed from source, double-check with the installation from binaries. If the strangeness goes away, it must be due to some quirk in your installation from source, so you know you must double-check the details of how you chose to build the latter.

In the following sections, for clarity, we assume you have made a new folder called *%USERPROFILE%\py* (e.g., *c:\users\tim\py*), which you can do, for example, by typing the **`mkdir`** command in any command window. Download the source *.tgz* file—for example, *Python-3.11.0.tgz*—to that folder. Of course, you can name and place the folder as it best suits you: our name choice is just for expository purposes.

Uncompressing and unpacking the Python source code

You can uncompress and unpack a *.tgz* or *.tar.xz* file with, for example, the free program **7-Zip**. Download the appropriate version from the **Download page**, install it, and run it on the *.tgz* file (e.g., *c:\users\alex\py\Python-3.11.0.tgz*) that you downloaded from the Python website. Assuming you downloaded this file into your *%USERPROFILE%\py* folder (or moved it there from *%USERPROFILE%\downloads*, if necessary), you will now have a folder called *%USERPROFILE%\py\Python-3.11.0* or similar, depending on the version you downloaded. This is the root of a tree that contains the entire standard Python distribution in source form.

Building the Python source code

Open the *readme.txt* file located in the *PCBuild* subdirectory of this root folder with any text editor, and follow the detailed instructions found there.

Unix-Like Platforms

On Unix-like platforms, installing Python from source code is generally simple.[10] In the following sections, for clarity, we assume you have created a new directory named *~/py* and downloaded the source *.tgz* file—for example, *Python-3.11.0.tgz*—to that directory. Of course, you can name and place the directory as it best suits you: our name choice is just for expository purposes.

Uncompressing and unpacking the Python source code

You can uncompress and unpack a *.tgz* or *.tar.xz* file with the popular GNU version of `tar`. Just type the following at a shell prompt:

```
$ cd ~/py && tar xzf Python-3.11.0.tgz
```

You now have a directory called *~/py/Python-3.11.0* or similar, depending on the version you downloaded. This is the root of a tree that contains the entire standard Python distribution in source form.

Configuring, building, and testing

You’ll find detailed notes in the *README* file inside this directory, under the heading “Build instructions,” and we recommend you study those notes. In the simplest case, however, all you need may be to give the following commands at a shell prompt:

```
$ cd ~/py/Python-3.11/0
$ ./configure
    [configure writes much information, snipped here]
$ make
    [make takes quite a while and emits much information, snipped here]
```

If you run **make** without first running **./configure**, **make** implicitly runs **./configure**. When **make** finishes, check that the Python you have just built works as expected:

```
$ make test
    [takes quite a while, emits much information, snipped here]
```

Usually, **make test** confirms that your build is working, but also informs you that some tests have been skipped because optional modules were missing.

Some of the modules are platform-specific (e.g., some may work only on machines running SGI’s ancient IRIX operating system); you don’t need to worry about them. However, other modules may be skipped because they depend on other open source packages that are currently not installed on your machine. For example, on Unix, the module `_tkinter`—needed to run the Tkinter GUI package and the IDLE integrated development environment, which come with Python—can be built only if **./configure** can find an installation of Tcl/Tk 8.0 or later on your machine. See the *README* file for more details and specific caveats about different Unix and Unix-like platforms.

Building from source code lets you tweak your configuration in several ways. For example, you can build Python in a special way that helps you debug memory leaks when you develop C-coded Python extensions, covered in “Building and Installing C-Coded Python Extensions” in **Chapter 25**. `./configure --help` is a good source of information about the configuration options you can use.

Installing after the build

By default, `./configure` prepares Python for installation in */usr/local/bin* and */usr/local/lib*. You can change these settings by running `./configure` with the option `--prefix` before running `make`. For example, if you want a private installation of Python in the subdirectory *py311* of your home directory, run:

```
$ cd ~/py/Python-3.11.0
$ ./configure --prefix=~/py311
```

and continue with `make` as in the previous section. Once you’re done building and testing Python, to perform the actual installation of all files, run the following command:[11]

```
$ make install
```

The user running `make install` must have write permissions on the target directories. Depending on your choice of target directories, and the permissions on those directories, you may need to `su` to *root*, *bin*, or some other user when you run `make install`. The common idiom for this purpose is `sudo make install`: if `sudo` prompts for a password, enter your current user’s password, not *root*’s. An alternative, and recommended, approach is to install into a virtual environment, as covered in **“Python Environments”**.

1 For Android, see ***https://wiki.python.org/moin/Android***, and for iPhone and iPad, see **Python for iOS and iPadOS**.

2 Python versions from 3.11 use “C11 without optional features” and specify that “the public API should be compatible with C++.”

3 Which can be in many programming languages, not just Python.

4 In fact, `conda`’s capabilities extend to other languages, and Python is simply another dependency.

5 250+ automatically installed with Anaconda, 7,500+ explicitly installable with **`conda install`**.

6 A popular business model is *freemium*: releasing both a free version and a commercial “premium” version with tech support and, perhaps, extra features.

7 The Python Software Foundation runs significant infrastructure to support the Python ecosystem. Donations to the PSF are always welcome.

8 We need to mobilize to get more **penguins** interested in our language!

9 Or, in modern Windows versions, the vastly preferable **Windows Terminal**.

10 Most problems with source installations concern the absence of various supporting libraries, which may cause some features to be missing from the built interpreter. The “Python Developers’ Guide” explains **how to handle dependencies on various platforms**. ***build-python-from-source.com*** is a helpful site that shows you all the commands necessary to download, build, and install a specific version of Python, plus most of the needed supporting libraries on several Linux platforms.

11 Or **`make altinstall`**, if you want to avoid creating links to the Python executable and manual pages.