Chapter 2. The Python Interpreter

To develop software systems in Python, you usually write text files that contain Python source code. You can do this using any text editor, including those we list in **"Python Development Environments"**. Then you process the source files with the Python compiler and interpreter. You can do this directly, within an integrated development environment (IDE), or via another program that embeds Python. The Python interpreter also lets you execute Python code interactively, as do IDEs.

The python Program

The Python interpreter program is run as **`python`** (it's named *python.exe* on Windows). The program includes both the interpreter itself and the Python compiler, which is implicitly invoked as needed on imported modules. Depending on your system, the program may have to be in a directory listed in your `PATH` environment variable. Alternatively, as with any other program, you can provide its complete pathname at a command (shell) prompt or in the shell script (or shortcut target, etc.) that runs it.[1]

On Windows, press the Windows key and start typing **`python`**. "Python 3.x" (the command-line version) appears, along with other choices, such as "IDLE" (the Python GUI).

Environment Variables

Besides `PATH`, other environment variables affect the **`python`** program. Some of these have the same effects as options passed to **`python`** on the command line, as we show in the next section, but several environment variables provide settings not available via command-line options. The

following list introduces some frequently used ones; for complete details, see the **online docs**:

`PYTHONHOME`
: The Python installation directory. A *lib* subdirectory, containing the Python standard library, must be under this directory. On Unix-like systems, standard library modules should be in *lib/python-3.x* for Python 3.*x*, where *x* is the minor Python version. If `PYTHONHOME` is not set, Python makes an informed guess about the installation directory.

`PYTHONPATH`
: A list of directories, separated by colons on Unix-like systems and by semicolons on Windows, from which Python can import modules. This list extends the initial value for Python's `sys.path` variable. We cover modules, importing, and `sys.path` in **Chapter 7**.

`PYTHONSTARTUP`
: The name of a Python source file to run each time an interactive interpreter session starts. No such file runs if you don't set this variable, or set it to the path of a file that is not found. The `PYTHONSTARTUP` file does not run when you run a Python script; it runs only when you start an interactive session.

How to set and examine environment variables depends on your operating system. In Unix, use shell commands, often within startup shell scripts. On Windows, press the Windows key and start typing **`environment var`**, and a couple of shortcuts appear: one for user environment variables, the other for system ones. On a Mac, you can work just as on other Unix-like systems, but you have more options, including a MacPython-specific IDE. For more information about Python on the Mac, see **"Using Python on a Mac" in the online docs**.

Command-Line Syntax and Options

The Python interpreter's command-line syntax can be summarized as follows:

```
[path]python {options} [-c command | -m module | file | -] {args}
```

Brackets ([]) enclose what's optional, braces ({}) enclose items of which zero or more may be present, and bars (|) mean a choice among alternatives. Python uses a slash (/) for filepaths, as in Unix.

Running a Python script at a command line can be as simple as:

```
$ python hello.py
Hello World
```

You can also explicitly provide the path to the script:

```
$ python ./hello/hello.py
 Hello World
```

The filename of the script can be an absolute or relative filepath, and need not have any specific extension (although it is conventional to use a *.py* extension).

`options` are case-sensitive short strings, starting with a hyphen, that ask **`python`** for nondefault behavior. **`python`** accepts only options that start with a hyphen (-). The most frequently used options are listed in **Table 2-1**. Each option's description gives the environment variable (if any) that, when set, requests that behavior. Many options have longer versions, starting with two hyphens, as shown by **`python -h`**. For full details, see the **online docs**.

Table 2-1. Frequently used `python` command-line options

Option	Meaning (and corresponding environment variable, if any)
`-B`	Don't save bytecode files to disk (`PYTHONDONTWRITEBYTECODE`)
`-c`	Gives Python statements within the command line

Option	Meaning (and corresponding environment variable, if any)
`-E`	Ignores all environment variables
`-h`	Shows the full list of options, then terminates
`-i`	Runs an interactive session after the file or command runs (`PYTHONINSPECT`)
`-m`	Specifies a Python module to run as the main script
`-O`	Optimizes bytecode (`PYTHONOPTIMIZE`)—note that this is an uppercase letter `O`, not the digit `0`
`-OO`	Like `-O`, but also removes docstrings from the bytecode
`-S`	Omits the implicit `import site` on startup (covered in **"Per-Site Customization"**)
`-t`, `-tt`	Issues warnings about inconsistent tab usage (`-tt` issues errors, rather than just warnings, for the same issues)
`-u`	Uses unbuffered binary files for standard output and standard error (`PYTHONUNBUFFERED`)
`-v`	Verbosely traces module import and cleanup actions (`PYTHONVERBOSE`)
`-V`	Prints the Python version number, then terminates
`-W arg`	Adds an entry to the warnings filter (see **"The warnings Module"**)
`-x`	Excludes (skips) the first line of the script's source

Use `-i` when you want to get an interactive session immediately after running some script, with top-level variables still intact and available for inspection. You do not need `-i` for normal interactive sessions, though it does no harm.

`-O` and `-OO` yield small savings of time and space in bytecode generated for modules you import, turning `assert` statements into no-operations, as covered in **"The assert Statement"**. `-OO` also discards documentation strings.[2]

After the options, if any, tell Python which script to run by adding the filepath to that script. Instead of a filepath, you can use `-c` *command* to execute a Python code string command. A *command* normally contains spaces, so you'll need to add quotes around it to satisfy your operating system's shell or command-line processor. Some shells (e.g., **bash**) let you enter multiple lines as a single argument, so that *command* can be a series of Python statements. Other shells (e.g., Windows shells) limit you to a single line; *command* can then be one or more simple statements separated by semicolons (`;`), as we discuss in **"Statements"**.

Another way to specify which Python script to run is with `-m` *module*. This option tells Python to load and run a module named ***module*** (or the *__main__.py* member of a package or ZIP file named *module*) from some directory that is part of Python's `sys.path`; this is useful with several modules from Python's standard library. For example, as covered in **"The timeit module"**, `-m timeit` is often the best way to perform micro-benchmarking of Python statements.

A hyphen (`-`), or the lack of any token in this position, tells the interpreter to read the program source from standard input—normally, an interactive session. You need a hyphen only if further arguments follow. *args* are arbitrary strings; the Python you run can access these strings as items of the list `sys.argv`.

For example, enter the following at a command prompt to have Python show the current date and time:

```
$ python -c "import time; print(time.asctime())"
```

You can start the command with just **python** (you do not have to specify the full path to Python) if the directory of the Python executable is in your PATH environment variable. (If you have multiple versions of Python installed, you can specify the version with, for example, **python3** or **python3.10**, as appropriate; then, the version used if you just say **python** is the one you installed most recently.)

The Windows py Launcher

On Windows, Python provides the **py** launcher to install and run multiple Python versions on a machine. At the bottom of the installer, you'll find an option to install the launcher for all users (it's checked by default). When you have multiple versions, you can select a specific version using **py** followed by a version option instead of the plain **python** command. Common **py** command options are listed in Table 2-2 (use **py -h** to see all the options).

Table 2-2. Frequently used py command-line options

Option	Meaning
-2	Run the latest installed Python 2 version.
-3	Run the latest installed Python 3 version.
-3.*x* or **-3.*x*-*nn***	Run a specific Python 3 version. When referenced as just **-3.10**, uses the 64-bit version, or the 32-bit version if no 64-bit version is available. **-3.10-32** or **-3.10-64** picks a specific build when both are installed.
-0 or **--list**	List all installed Python versions, including an indication of whether a build is 32- or 64-bit, such as **3.10-64**.

Option	Meaning
`-h`	List all **py** command options, followed by standard Python help.

If no version option is given, **py** runs the latest installed Python.

For example, to show the local time using the installed Python 3.9 64-bit version, you can run this command:

```
C:\> py -3.9 -c "import time; print(time.asctime())"
```

(Typically, there is no need to give a path to **py**, since installing Python adds **py** to the system PATH.)

The PyPy Interpreter

PyPy, written in Python, implements its own compiler to generate LLVM intermediate code to run on an LLVM backend. The PyPy project offers some improvements over standard CPython, most notably in the areas of performance and multithreading. (At this writing, PyPy is up-to-date with Python 3.9.)

pypy may be run similarly to **python**:

```
[path]pypy {options} [-c command | file | - ] {args}
```

See the PyPy **home page** for installation instructions and complete up-to-date information.

Interactive Sessions

When you run **`python`** without a script argument, Python starts an interactive session and prompts you to enter Python statements or expressions. Interactive sessions are useful to explore, to check things out, and to use Python as a powerful, extensible interactive calculator. (Jupyter Notebook, discussed briefly at the end of this chapter, is like a "Python on steroids" specifically for interactive session usage.) This mode is often referred to as a *REPL*, or read–evaluate–print loop, since that's pretty much what the interpreter then does.

When you enter a complete statement, Python executes it. When you enter a complete expression, Python evaluates it. If the expression has a result, Python outputs a string representing the result and also assigns the result to the variable named _ (a single underscore) so that you can immediately use that result in another expression. The prompt string is `>>>` when Python expects a statement or expression, and `...` when a statement or expression has been started but not completed. In particular, Python prompts with `...` when you have opened a parenthesis, bracket, or brace on a previous line and haven't closed it yet.

While working in the interactive Python environment, you can use the built-in **`help()`** function to drop into a help utility that offers useful information about Python's keywords and operators, installed modules, and general topics. When paging through a long help description, press **`q`** to return to the `help>` prompt. To exit the utility and return to the Python `>>>` prompt, type **`quit`**. You can also get help on specific objects at the Python prompt without entering the help utility by typing **`help(`***`obj`***`)`**, where *`obj`* is the program object you want more help with.

There are several ways you can end an interactive session. The most common are:

- Enter the end-of-file keystroke for your OS (Ctrl-Z on Windows, Ctrl-D on Unix-like systems).
- Execute either of the built-in functions `quit` or `exit`, using the form `quit()` or `exit()`. (Omitting the trailing `()` will display a message

like “Use quit() or Ctrl-D (i.e., EOF) to exit,” but will still leave you in the interpreter.)

- Execute the statement **`raise`** `SystemExit`, or call `sys.exit()` (we cover `SystemExit` and **`raise`** in **Chapter 6**, and the `sys` module in **Chapter 8**).

USE THE PYTHON INTERACTIVE INTERPRETER TO EXPERIMENT

Trying out Python statements in the interactive interpreter is a quick way to experiment with Python and immediately see the results. For example, here is a simple use of the built-in `enumerate` function:

```
>>> print(list(enumerate("abc")))
```

```
[(0, 'a'), (1, 'b'), (2, 'c')]
```

The interactive interpreter is a good introductory platform to learn core Python syntax and features. (Experienced Python developers often open a Python interpreter to quickly check out an infrequently used command or function.)

Line-editing and history facilities depend in part on how Python was built: if the `readline` module was included, all features of the GNU `readline` library are available. Windows has a simple but usable history facility for interactive text mode programs like **`python`**.

In addition to the built-in Python interactive environment, and those offered as part of richer development environments covered in the next section, you can freely download other powerful interactive environments. The most popular one is ***IPython***, covered in **“IPython”**, which offers a dazzling wealth of features. A simpler, lighter weight, but still quite handy alternative read-line interpreter is ***bpython***.

Python Development Environments

The Python interpreter's built-in interactive mode is the simplest development environment for Python. It is primitive, but it's lightweight, has a small footprint, and starts fast. Together with a good text editor (as discussed in **"Free Text Editors with Python Support"**) and line-editing and history facilities, the interactive interpreter (or, alternatively, the much more powerful IPython/Jupyter command-line interpreter) is a usable development environment. However, there are several other development environments you can use.

IDLE

Python's **Integrated Development and Learning Environment (IDLE)** comes with standard Python distributions on most platforms. IDLE is a cross-platform, 100% pure Python application based on the Tkinter GUI. It offers a Python shell similar to the interactive Python interpreter, but richer. It also includes a text editor optimized to edit Python source code, an integrated interactive debugger, and several specialized browsers/viewers.

For more functionality in IDLE, install **IdleX**, a substantial collection of free third-party extensions.

To install and use IDLE on macOS, follow the specific **instructions** on the Python website.

Other Python IDEs

IDLE is mature, stable, easy, fairly rich, and extensible. There are, however, many other IDEs: cross-platform or platform specific, free or commercial (including commercial IDEs with free offerings, especially if you're developing open source software), standalone or add-ons to other IDEs.

Some of these IDEs sport features such as static analysis, GUI builders, debuggers, and so on. Python's IDE **wiki page** lists over 30, and points to many other URLs with reviews and comparisons. If you're an IDE collector, happy hunting!

We can't do justice to even a tiny subset of all the available IDEs. The free third-party plug-in **PyDev** for the popular cross-platform, cross-language modular IDE **Eclipse** has excellent Python support. Steve is a longtime user of **Wing** by Archaeopteryx, the most venerable Python-specific IDE. Paul's IDE of choice, and perhaps the single most popular third-party Python IDE today, is **PyCharm** by JetBrains. **Thonny** is a popular beginner's IDE, lightweight but full featured and easily installed on the Raspberry Pi (or just about any other popular platform). And not to be overlooked is Microsoft's **Visual Studio Code**, an excellent, very popular cross-platform IDE with support (via plug-ins) for a number of languages, including Python. If you use Visual Studio, check out **PTVS**, an open source plug-in that's particularly good at allowing mixed-language debugging in Python and C as and when needed.

Free Text Editors with Python Support

You can edit Python source code with any text editor, even simple ones such as Notepad on Windows or *ed* on Linux. Many powerful free editors support Python with extra features such as syntax-based colorization and automatic indentation. Cross-platform editors let you work in uniform ways on different platforms. Good text editors also let you run, from within the editor, tools of your choice on the source code you're editing. An up-to-date list of editors for Python can be found on the **PythonEditors wiki**, which lists dozens of them.

The very best for sheer editing power may be classic **Emacs** (see the Python **wiki page** for Python-specific add-ons). Emacs is not easy to learn, nor is it lightweight.[3] Alex's personal favorite[4] is another classic: **Vim**, Bram Moolenaar's improved version of the traditional Unix editor *vi*. It's arguably not *quite* as powerful as Emacs, but still well worth considering —it's fast, lightweight, Python programmable, and runs everywhere in both text mode and GUI versions. For excellent Vim coverage, see

Learning the vi and Vim Editors, 8th edition, by Arnold Robbins and Elbert Hannah (O'Reilly); see the Python **wiki page** for Python-specific tips and add-ons. Steve and Anna use Vim too, and where it's available, Steve also uses the commercial editor **Sublime Text**, with good syntax coloring and enough integration to run your programs from inside the editor. For quick editing and executing of short Python scripts (and as a fast and lightweight general text editor, even for multimegabyte text files), **SciTE** is Paul's go-to editor.

Tools for Checking Python Programs

The Python compiler checks program syntax sufficiently to be able to run the program, or to report a syntax error. If you want more thorough checks of your Python code, you can download and install one or more third-party tools for the purpose. **`pyflakes`** is a very quick, lightweight checker: it's not thorough, but it doesn't import the modules it's checking, which makes using it fast and safe. At the other end of the spectrum, **`pylint`** is very powerful and highly configurable; it's not lightweight, but repays that by being able to check many style details in highly customizable ways based on editable configuration files.[5] **`flake8`** bundles `pyflakes` with other formatters and custom plug-ins, and can handle large codebases by spreading work across multiple processes. **`black`** and its variant **`blue`** are intentionally less configurable; this makes them popular with widely dispersed project teams and open source projects in order to enforce a common Python style. To make sure you don't forget to run them, you can incorporate one or more of these checkers/formatters into your workflow using the **`pre-commit` package**.

For more thorough checking of Python code for proper type usages, use tools like **`mypy`**; see **Chapter 5** for more on this topic.

Running Python Programs

Whatever tools you use to produce your Python application, you can see your application as a set of Python source files, which are normal text files that typically have the extension *.py*. A *script* is a file that you can

run directly. A *module* is a file that you can import (as covered in **Chapter 7**) to provide some functionality to other files or interactive sessions. A Python file can be *both* a module (providing functionality when imported) *and* a script (OK to run directly). A useful and widespread convention is that Python files that are primarily intended to be imported as modules, when run directly, should execute some self-test operations, as covered in **"Testing"**.

The Python interpreter automatically compiles Python source files as needed. Python saves the compiled bytecode in a subdirectory called *__pycache__* within the directory with the module's source, with a version-specific extension annotated to denote the optimization level.

To avoid saving compiled bytecode to disk, you can run Python with the option `-B`, which can be handy when you import modules from a read-only disk. Also, Python does not save the compiled bytecode form of a script when you run the script directly; instead, Python recompiles the script each time you run it. Python saves bytecode files only for modules you import. It automatically rebuilds each module's bytecode file whenever necessary—for example, when you edit the module's source. Eventually, for deployment, you may package Python modules using tools covered in **Chapter 24** (available **online**).

You can run Python code with the Python interpreter or an IDE.[6] Normally, you start execution by running a top-level script. To run a script, give its path as an argument to **`python`**, as covered in **"The python Program"**. Depending on your operating system, you can invoke **`python`** directly from a shell script or command file. On Unix-like systems, you can make a Python script directly executable by setting the file's permission bits `x` and `r`, and beginning the script with a *shebang* line, a line such as:

```
#!/usr/bin/env python
```

or some other line starting with `#!` followed by a path to the `python` interpreter program, in which case you can optionally add a single word of op-

tions—for example:

```
#!/usr/bin/python -OB
```

On Windows, you can use the same style `#!` line, in accordance with **PEP 397**, to specify a particular version of Python, so your scripts can be cross-platform between Unix-like and Windows systems. You can also run Python scripts with the usual Windows mechanisms, such as double-clicking their icons. When you run a Python script by double-clicking the script's icon, Windows automatically closes the text-mode console associated with the script as soon as the script terminates. If you want the console to linger (to allow the user to read the script's output on the screen), ensure the script doesn't terminate too soon. For example, use, as the script's last statement:

```
input('Press Enter to terminate')
```

This is not necessary when you run the script from a command prompt.

On Windows, you can also use the extension *.pyw* and interpreter program *pythonw.exe* instead of *.py* and *python.exe*. The *w* variants run Python without a text-mode console, and thus without standard input and output. This is good for scripts that rely on GUIs or run invisibly in the background. Use them only when a program is fully debugged, to keep standard output and error available for information, warnings, and error messages during development.

Applications coded in other languages may embed Python, controlling the execution of Python for their own purposes. We examine this briefly in "Embedding Python" in **Chapter 25** (available **online**).

Running Python in the Browser

There are also options for running Python code within a browser session, executed in either the browser process or some separate server-based component. PyScript exemplifies the former approach, and Jupyter the latter.

PyScript

A recent development in the Python-in-a-browser endeavor is the release of **PyScript** by Anaconda. PyScript is built on top of Pyodide,[7] which uses WebAssembly to bring up a full Python engine in the browser. PyScript introduces custom HTML tags so that you can write Python code without having to know or use JavaScript. Using these tags, you can create a static HTML file containing Python code that will run in a remote browser, with no additional installed software required.

A simple PyScript "Hello, World!" HTML file might look like this:

```
<html>
<head>
    <link rel='stylesheet'
 href='https://pyscript.net/releases/2022.06.1/pyscript.css' />
    <script defer
 src='https://pyscript.net/releases/2022.06.1/pyscript.js'></script>
</head>
<body>
<py-script>
import time
print('Hello, World!')
print(f'The current local time is {time.asctime()}')
print(f'The current UTC time is {time.asctime(time.gmtime())}')
</py-script>
</body>
</html>
```

You can save this code snippet as a static HTML file and successfully run it in a client browser, even if Python isn't installed on your computer.

CHANGES ARE COMING TO PYSCRIPT

PyScript is still in early development at the time of publication, so the specific tags and APIs shown here are likely to change as the package undergoes further development.

For more complete and up-to-date information, see the **PyScript website**.

Jupyter

The extensions to the interactive interpreter in IPython (covered in **"IPython"**) were further extended by the **Jupyter project**, best known for the Jupyter Notebook, which offers Python developers a **"literate programming"** tool. A notebook server, typically accessed via a website, saves and loads each notebook, creating a Python kernel process to execute its Python commands interactively.

Notebooks are a rich environment. Each one is a sequence of cells whose contents may either be code or rich text formatted with the Markdown language extended with LaTeX, allowing complex mathematics to be included. Code cells can produce rich outputs too, including most popular image formats as well as scripted HTML. Special integrations adapt the `matplotlib` library to the web, and there are an increasing number of mechanisms for interaction with notebook code.

Further integrations allow notebooks to appear in other ways. For example, with the right extension, you can easily format a Jupyter notebook as a **reveal.js** slideshow for presentations in which the code cells can be interactively executed. **Jupyter Book** allows you to collect notebooks together as chapters and publish the collection as a book. GitHub allows browsing (but not executing) of uploaded notebooks (a special renderer provides correct formatting of the notebook).

There are many examples of Jupyter notebooks available on the internet. For a good demonstration of its features, take a look at the **Executable Books website**; notebooks underpin its publishing format.

1 This may involve using quotes if the pathname contains spaces—again, this depends on your operating system.

2 This may affect code that parses docstrings for meaningful purposes; we suggest you avoid writing such code.

3 A great place to start is ***Learning GNU Emacs*, 3rd edition** (O'Reilly).

4 Not only as "an editor," but also as Alex's favorite "as close to an IDE as Alex will go" tool!

5 `pylint` also includes the useful `pyreverse` utility to autogenerate **UML** class and package diagrams directly from your Python code.

6 Or online: Paul, for example, maintains a **list** of online Python interpreters.

7 A great example of the synergy open source gets by projects "standing on the shoulders of giants" as an ordinary, everyday thing!