Contents

CHAPTER 1

Getting Started with Python

The Analytical Engine weaves algebraical patterns,
just as the Jacquard loom weaves flowers and leaves.
— Ada, Countess of Lovelace, 1815–1853

1.1 ALGORITHMS AND ALGORITHMIC THINKING

The goal of this tutorial is to get you started in computational science using the computer language Python. Python is open-source software. You can download, install, and use it anywhere. Many good introductions exist, and more are written every year. *This* one is distinguished mainly by the fact that it focuses on skills useful for solving problems in physical modeling.

Modeling a physical system can be a complicated task. Let's take a look at how we can use the powerful processors inside your computer to help.

1.1.1 Algorithmic thinking

Suppose that you need to instruct a friend how to back your car out of your driveway. Your friend has never driven a car, but it's an emergency, and your only communication channel is a phone conversation before the operation begins.

You need to break the required task down into small, explicit steps that your friend understands and can execute in sequence. For example, you might provide your friend with the following set of instructions:

```
Put the key in the ignition.
Turn the key until the car starts, then let go.
Push the button on the shift lever and move it to "Reverse."
...
```

Unfortunately, for many cars this "code" won't work, even if your friend understands each instruction: It contains a **bug**. Before step 3, many cars require that the driver

```
Press down the left pedal.
```

Also, the shifter may be marked `R` instead of `Reverse`. It is difficult at first to get used to the high degree of precision required when composing instructions like these.

Because you are giving the instructions in advance (your friend has no mobile phone), it's also wise to allow for contingencies:

```
If a crunching sound is heard, press down on the left pedal ...
```

Breaking the steps of a long operation down into small, explicit substeps and anticipating contingencies are the beginning of *algorithmic thinking*.

If your friend has had a lot of experience watching people drive cars, then the instructions above may suffice. But a friend from Mars—or a robot—would need much more detail. For example, the first two steps may need to be expanded to something like

```
Grab the wide end of the key.
Insert the pointed end of the key into the slot on the lower right side
        of the steering column.
Rotate the key about its long axis in the clockwise direction
        (when viewed from the wide end toward the pointed end).
...
```

These two sets of instructions illustrate the difference between low-level and high-level languages for communicating with a computer. A *low-level* computer program is similar to the second set of explicit instructions, written in a language that a machine can understand.[1] A *high-level* system understands many common tasks, and therefore can be programmed in a more condensed style, like the first set of instructions above. Python is a high-level language. It includes commands for common operations in mathematical calculations, processing text, and manipulating files. In addition, Python can access many *standard libraries*, which are collections of programs that perform advanced functions such as data visualization and image processing.

Python also comes with a *command line interpreter*—a program that executes Python commands as you type them. Thus, with Python, you can save instructions in a file and run them later, or you can type commands and execute them immediately. In contrast, many other programming languages used in scientific computing, like C++ or FORTRAN, require you to *compile* your programs before you can *execute* them. A separate program called a compiler translates your code into a low-level language. You then run the resulting compiled program to execute (carry out) your algorithm. With Python, it is comparatively easy to quickly write, run, and debug programs. (It still takes patience and practice, though.)

A command line interpreter combined with standard libraries and programs you write yourself provides a convenient and powerful scientific computing platform.

1.1.2 States

You have probably studied multistep mathematical proofs, perhaps long ago in a geometry class. The goal is to verify a proposition through a chain of steps that use given information and a formal system. Thus, each statement's truth, although not evident in isolation, is supposed to be straightforward in light of the preceding statements. The reader's "state" (list of propositions known to be true) changes while reading through the proof. At the end, that list includes the desired result.

An **algorithm** has a different goal. It is a chain of instructions, each of which describes a simple operation, that accomplishes a complex task. The chain may involve a lot of repetition, so you won't want to supervise the execution of every step. Instead, you specify all the steps in advance, then stand back while your electronic assistant performs them rapidly. There may also be contingencies that cannot be known in advance. (`If a crunching sound is heard, ...`)

In an algorithm, the *computer* has a state that is constantly being modified. For example, it has many memory cells, whose contents may change during the course of an operation. Your goal might be to

[1] Machine code and assembly language are low-level programming languages.

arrange for one or more of these cells to contain the result of some complex calculation once the algorithm has finished running. You may also want a particular graphical image to appear.

1.1.3 What does `a = a + 1` mean?

To get a computer to execute your algorithm, you must first express it in a programming language. The commands used in computer programming can be confusing at first, especially when they contradict standard mathematical usage. For example, many programming languages (including Python) accept statements such as these:

```
a = 100
a = a + 1
```

In mathematics, this makes no sense. The second line is an assertion that is always false; equivalently, it is an equation with no solution. To Python, however, "=" is not a test of equality, but an instruction to be executed. These lines have roughly the following meaning:[2]

1. Assign the name `a` (a **variable**) to an integer object with the value 100.
2. Extract the value of the object named `a`. Calculate the sum of that value and 1. Assign the name `a` to the result, and *discard* whatever was previously stored under the name `a`.

In other words, the equals sign instructs Python to change its *state*. In contrast, mathematical notation uses the equals sign to create a proposition, which may be true or false. Note, too, that Python treats the left and right sides of the command `x=y` differently, whereas in math the equals sign is symmetric. For example, Python will give an error message if you say something like `b+1=a`; the left side of an assignment must be a name that can be assigned to the result of evaluating the right side.

We do often wish to check whether a variable has a particular value. To avoid ambiguity between assignment and testing for equality, Python uses a double equals sign for the latter:

```
a = 1
a == 0
b = (a == 1)
```

This code again creates a variable `a` and assigns it to a numerical value. Then it compares this numerical value with 0. Finally, it creates a second variable `b`, and assigns it a logical value (**`True`** or **`False`**) after performing another comparison. That value can be used in contingent code, as we'll see later.

Do not use = (assignment) when == (test for equality) is required.

This is a common mistake for beginning programmers. You can get mysterious results if you make this error, because both = and == are legitimate Python syntax. In any particular situation, however, only one of them is what you want.

1.1.4 Symbolic versus numerical

In math, it's perfectly reasonable to start a derivation with "Let $b = a^2 - a$," even if the reader doesn't yet know the value of a. This statement defines b in terms of a, whatever the value of a may be.

[2] $\boxed{T_2}$ Appendix F gives more precise information about the handling of assignment statements.

If you launch Python and immediately give the equivalent statement, `b=a**2-a`, the result is an error message.[3] Every time you hit `<Return/Enter>`, Python tries to compute values for every assignment statement. If the variable `a` has not been assigned a value yet, evaluation fails, and Python complains. Other computer math packages can accept such input, keep track of the symbolic relationship, and evaluate it later, but basic Python does not.[4]

In math, it's also understood that a definition like "Let $b = a^2 - a$" will persist unchanged throughout the discussion. If we say, "In the case $a = 1, \ldots$" then the reader knows that b equals zero; if later we say, "In the case $a = 2, \ldots$" then we need not reiterate the definition of b for the reader to know that this symbol now represents the value $2^2 - 2 = 2$.

In contrast, a numerical system like Python *forgets* any relation between `b` and `a` after executing the assignment `b=a**2-a`. All that it remembers is the *value* now assigned to `b`. If we later change the value of `a`, the value of `b` will *not* change.[5]

Changing symbolic relationships in the middle of a proof is generally not a good idea. However, in Python, if we say `b=a**2-a`, nothing stops us from later saying `b=2**a`. The second assignment updates Python's state by discarding the value calculated in the first assignment statement and replacing it with the newly computed value.

1.2 LAUNCH PYTHON

Rather than reading about what happens when you type some command, try out the commands for yourself. Appendix A describes how to install and launch Python. From now on, you should have Python running as you read: Try every snippet of code and observe what Python does in response. For example, this tutorial won't show you much graphics or output. You must generate these yourself as you work through the examples.

Reading this tutorial won't teach you Python. You can ***teach yourself*** *Python by working through all the examples and exercises here, and then using what you've learned on your own problems.*

Set yourself little challenges and test them out. ("What would happen if ...?" "How could I accomplish...?") Python is not some expensive piece of lab apparatus that could break or explode if you type something wrong! Just try things. This strategy is not only more fun than passively accumulating facts—it is also far more effective.

Before you start typing, we would like to to explain a few conventions we use in this book. The most important is this:

Python code consists entirely of plain text.

All fonts, typefaces, and coloring in the code samples of this tutorial were added for readability. These are not things you need to worry about while entering code. Similarly, the line numbers shown on the left of code samples are there to allow us to refer to particular lines. Don't type them. Spyder will assign and

[3] The notation `**` denotes exponentiation. See Section 1.4.2.

[4] The SymPy library makes symbolic calculations possible in Python. See Section 10.3.2.

[5] In math, the statement $b = a^2 - a$ essentially defines b as a *function* of a. We can certainly do that in Python by defining a function that returns the value of $a^2 - a$ and assigning that function the name `b` (see Section 6.1), but this is *not* what "=" does.

show line numbers when you work in the Editor, and Python will use them to tell you where it thinks you have made errors. They are not part of the code. Note also that most blank spaces are optional, except when used for indentation. We use extra blank spaces to improve readability, but these are not required.

This tutorial uses the following color and font scheme when displaying code:

- Built-in functions and reserved words are displayed in boldface green type: **`print`**`("Hello, world!")`. You do not need to import these functions.
- Python errors and runtime exceptions are displayed in boldface red type: **`SyntaxError`**.
- Functions and other objects from NumPy and PyPlot are displayed in boldface black type: **`np.sqrt`**`(2)`, or **`plt.plot`**`(x,y)`. We will assume that you import NumPy and PyPlot at the beginning of each session and program you write.[6]
- Functions imported from other libraries are displayed in blue boldface type: **`from scipy.special import factorial`**.
- Strings are displayed in red type: **`print`**`("Hello, world!")`.
- Comments are displayed in oblique blue type: *`# This is a comment.`*
- Keywords in function arguments are displayed in oblique black type: **`np.loadtxt`**`('data.csv', `*`delimiter`*`=',')`. Keywords are not arbitrary; they must be spelled exactly as shown.
- Buttons you can click with a mouse are displayed in small capitals within a rectangle: RUN ▶. Some buttons in Spyder have icons rather than text, but hovering the mouse pointer over the button will display the text shown in this tutorial.
- Keystrokes are displayed within angled brackets: `<Return>` or `<Ctrl-C>`.
- Most other text is displayed in normal type.

Regarding keystrokes, our notation may not look exactly like what you see on your keyboard, so we summarize our conventions in the table below. All keys that appear in a single unit should be pressed together. For example, `<Ctrl-C>` means press and hold the "control" key on your keyboard, and while holding it press the "c" key (you may also see this abbreviated as `^C`). Our conventions follow the macOS keyboard layout. If you are using Windows or Linux, substitute `<Ctrl>` for `<Cmd>`. We'll abbreviate `<Return/Enter>` as simply `<Return>`.

Key	Example	Function
enter or return	`<Return>`	end line or run command
control	`<Ctrl-C>`	interrupt current Python command
command	`<Cmd-V>`	paste from the clipboard
option or alt	`<Alt-Shift-R>`	restart Spyder

Now that you know what to type (plain text) and how to type it, all you need is Python! A complete Python programming environment has many components. See Table 1.1 for a brief description of the ones that we'll be discussing. Be aware that we use "Python" loosely in this guide. In addition to the language itself, Python may refer to a *Python interpreter*, which is a computer application that accepts commands and performs the steps described in a program. Python may also refer to the language together with common libraries.

[6] See Section 1.3 (page 11).

Python	A computer programming language. A way to describe algorithms to a computer.
IPython	A Python *interpreter:* A computer application that provides a convenient, interactive mode for executing Python commands and programs.
Spyder	An ***i**ntegrated **d**evelopment **e**nvironment* (IDE): A computer application that includes IPython, a tool to inspect variables, a text editor for writing and debugging programs, and more.
Jupyter	A notebook-style interface for Python.
NumPy	A standard library that provides numerical arrays and mathematical functions.
PyPlot	A standard library that provides visualization tools.
SciPy	A standard library that provides scientific computing tools.
Anaconda	A *distribution*: A single download that includes all of the above and provides access to many additional libraries for special purposes. It also includes a *package manager* that helps you to keep everything up to date.

Table 1.1: Elements of the Python environment described in this tutorial.

Most of the code that follows will run with any Python distribution. However, since we cannot provide instructions for every available version of Python and every integrated development environment (IDE), we have chosen the following particular setup:

- The Anaconda distribution of Python 3, available at `anaconda.com`.
  Many scientists instead use an earlier version of Python (such as version 2.7). Appendix E discusses the minor changes needed to adapt the codes in this tutorial for earlier versions.
- The Spyder IDE, which comes with Anaconda or can be obtained at `www.spyder-ide.org`. Any programming task can be accomplished with a different IDE—or with no IDE at all. Other IDEs are available, such as IDLE, which comes with every distribution of Python. Browser-based Jupyter notebooks and JupyterLab are another option.[7]

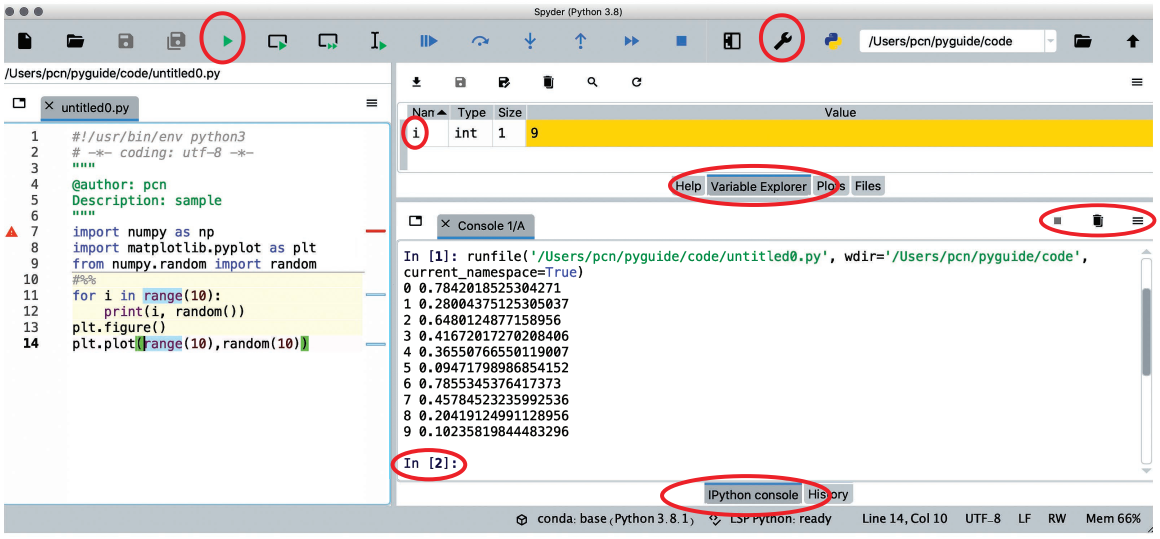
The choice of distribution is a matter of personal preference. We chose Anaconda because it is simple to install, update, and maintain, and it is free. You may find a different distribution is better suited to your needs. For example, you can install Python from source (`python.org`) and manage your packages with pip, but this tutorial will assume you are using Anaconda and the conda package manager.

1.2.1 IPython console

To keep our discussion focused on Python rather than on details of various platforms, we will assume that you are using Spyder as you work through this guide. This is not required! If you prefer to start with a simpler interface, you can open the Qt Console app from the Anaconda Navigator and start typing commands. If you prefer a notebook interface, you can follow along in a Jupyter Notebook. (See Appendix C.) If you prefer working from the command line, you can start IPython from a terminal. (See Appendix B.) At some point, though, you will need an IPython interpreter and a text editor. Spyder includes both of these, plus some other useful features, in an interface that will be familiar to users of MATLAB. There are many ways to use Python, and you can use any of them as you work through this tutorial. If you are new to Python, Spyder is a good choice.

Open Spyder now. Upon launch, Spyder opens a window that includes several *panes*. See Figure 1.1. There is an Editor pane on the left for editing program files (*scripts*). There are two panes on the right.

[7] If you prefer the notebook interface, see Appendix C to get started. Many code samples are available in notebook format via this book's blog.



```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@author: pcn
Description: sample
"""
import numpy as np
import matplotlib.pyplot as plt
from numpy.random import random
#%%
for i in range(10):
    print(i, random())
plt.figure()
plt.plot(range(10),random(10))
```

Name	Type	Size	Value
i	int	1	9

```
In [1]: runfile('/Users/pcn/pyguide/code/untitled0.py', wdir='/Users/pcn/pyguide/code',
current_namespace=True)
0 0.7842018525304271
1 0.28004375125305037
2 0.6480124877158956
3 0.41672017270208406
4 0.36550766550119007
5 0.09471798986854152
6 0.7855345376417373
7 0.45784523235992536
8 0.20419124991128956
9 0.10235819844483296

In [2]:
```

Figure 1.1: The Spyder display. Red circles have been added to emphasize (*from top to bottom*) the RUN ▶ button, Preferences (wrench icon), a variable in the Variable Explorer, the tab that brings the Variable Explorer to the front in its pane, the STOP ■, RESET 🗑, and OPTIONS ≡ buttons, the IPython command prompt, and the IPython console tab. Two scripts are open in the Editor; `untitled0.py` has been brought to the front by clicking its tab at the top of the Editor pane. Two warnings appear at far left.

The top-right pane may contain Help, Variable Explorer, Plots, and File Explorer tabs. If necessary, click on the Variable Explorer's tab to bring it to the front. The bottom-right pane should include a tab called "IPython console"; if necessary, click it now.[8] It provides the command line interpreter that allows you to execute Python commands interactively as you type them.

If your window layout gets disorganized, do not worry. It is easy to adjust. The standard format for Spyder is a single window, divided into the three panes just described. Each pane can have multiple tabs. If you have unwanted windows, close them individually by clicking on their CLOSE ⊗ buttons. You can also use the menu `View>Panes` to select panes you want to be visible and deactivate those you do not want. `View>Window layouts>Spyder Default Layout` will restore the standard layout.

Click in the IPython console. Now, things you type will show up after the *command prompt*. By default, this will be something like

```
In[1]:
```

Try typing simple commands like "`2+2`" and hitting `<Return>` after each line. Python responds immediately after each `<Return>`, attempting to perform whatever command you entered.[9]

Click on the Variable Explorer tab. Each time you enter a command and hit `<Return>`, the contents

[8] If no IPython console tab is present, you can open one from the menu at the top of the screen: `Consoles>New console`.

[9] This tutorial uses the word "command" to mean any Python statement that can be executed by an interpreter. Assignments like `a=1`, function calls like `plt.plot(x,y)`, and special instructions like `%reset` are commands.

of this pane will reflect any changes in Python's state: Initially empty, it will display a list of your variables and a summary of their values.[10] When a variable contains many values (for example, an array), you can double-click its entry in this list to open a spreadsheet that contains all the values of the array. You can copy from this spreadsheet and paste into other applications.

At any time, you can reset Python's state by quitting and relaunching it, or by executing the command

```
%reset
```

Because you are about to delete almost everything that has been created in this session, you will be asked to confirm this irreversible operation.[11] Press `<y>` then `<Return>` to proceed. (Commands that begin with a `%` symbol are **magic commands**: commands specific to the IPython interpreter. They may not work in a more basic Python interpreter, or in scripts that you write. To learn more about these, type `%magic` at the IPython command prompt.)

Example: Use the `%reset` command, then try the following commands at the prompt. Type each line exactly as shown, then press `<Return>`. Explain everything you see happen:

```
q
q == 2
q = 2
q
q == 2
q == 3
```

Solution: Python complains about the first two lines: Initially, the symbol `q` is not associated with any object. It has no value, and so expressions involving it cannot be evaluated. Altering Python's state in the third line above changes this situation, so the last three lines do not generate errors.

Example: Now clear Python's state again. Try the following at the prompt, and explain everything that happens. (It may be useful to refer to Section 1.1.4.)

```
a = 1
a
b = a**2 - a
b
a = 2
print(a)
print(b)
b = a**2 - a
a, b
print(a, b)
```

Solution: The results from the first four lines should be clear: We assign values to the variables `a` and `b`. In the fifth line, we change the value of `a`, but because Python remembers only the *value* of `b` and not its relation to `a`, `b`'s value is unchanged until we update it explicitly in the eighth line.

[10] Some variables will not appear. You can control which variables are excluded through the OPTIONS ≡ menu, in the upper-right corner of the Variable Explorer pane.

[11] If IPython does not seem to respond to `%reset`, try scrolling the IPython console up manually to see the confirm query.

When entering code at the command prompt, you may run into a confusing situation where Python seems unresponsive and displays "`...:`" instead of executing commands.

> *If a command contains an unmatched* `(`, `[`, *or* `{`, *then Python continues reading more lines, searching for the corresponding* `)`, `]`, *or* `}`.

Look for an unmatched bracket. If you find one, type the closing bracket and press `<Return>`. If you cannot figure out how to match up your brackets, or if there is some other problem, you can force execution with `<Shift + Return>` or abort the command by pressing `<Esc>`.[12]

The examples above illustrate an important point: An assignment statement does not display the value that it assigns to a variable. To see the value assigned to a variable in an IPython session, use the `print()` command or type the variable name on a line by itself.[13]

The last two lines of the example above illustrate how to see the values of multiple objects at once. Notice that the output is not exactly the same.

You can end a command by starting a new line. Or, if you wish, you can end a command with a semicolon (`;`) and then add another command on the same line.

It is also possible to make multiple assignments with a single = command. This is an alternative to using semicolons. Both of the following lines assign the same values to their respective variables:

```
a = 1; b = 2; c = 3
x, y, z = 1, 2, 3
```

Either side of the second command may be enclosed in parentheses without affecting the result.

The preceding paragraph demonstrates ways to save space and reduce typing with Python. Sometimes this is convenient, but it's best not to make too much use of this ability. You should instead try to make the *meaning* of your code as clear as possible. Human readability is worth a few extra seconds of typing or a few extra lines in a program.

In some situations, you may wish to use a very long command that doesn't fit on one line. For such cases, you can end a line with a backslash (\). Python will then continue reading the next line as part of the same command. Try this:

```
q = 1 + \
2
print(q)
```

A single command can even stretch over multiple lines:

```
xv\
a\
l\
= 1 + \
2
```

This will create a variable `xval` and assign it the value 3.

To write clear code, use backslashes and semicolons sparingly.

[12] `<Esc>` cancels the current command in Spyder. In another IDE or interpreter, you may need to use `<Ctrl-C>` to interrupt and `<Alt+Return>` to force execution.

[13] In scripts that you write, Python will evaluate an expression *without* showing anything on the screen; if you want output, you must give an explicit `print()` command. Scripts will be discussed in Section 3.3.

1.2.2 Error messages

You should have encountered some error messages by now. When Python detects an error, it tells you where it encountered the error, provides a fragment of the code surrounding the statement that caused the problem, and tells you which general kind of error it detected among the many types it recognizes. For example, Python responds with a `NameError` whenever you try to evaluate an undefined variable. (Recall the Example on page 8.) See Appendix D for a description of common Python errors and some hints for interpreting the resulting messages.

Donald Knuth, a well-known computer scientist, once wrote, "Error messages can be terrifying when you aren't prepared for them; but they can be fun when you have the right attitude. Just remember that you really haven't hurt the computer's feelings, and that nobody will hold the errors against you." We encourage you to adopt this attitude.

- Don't be afraid to make errors. It's very hard to break anything.
- Read the error messages. They tell you what kind of error you made—not just that you made an error.
- Inspect the code that produces an error. You can learn from your mistakes.

This approach to errors will make you a better coder right away, and it will help you "debug" more complicated programs later.[14]

1.2.3 Sources of help

The definitive documentation on Python is available online at `www.python.org/doc`. However, in many cases you'll find the answers you need more quickly by other means, such as asking a friend, searching the web, or visiting `stackoverflow.com`.

Suppose that you wish to evaluate the square root of 2. You type `2**0.5` and hit `<Return>`. That does the job, but Python is displaying 16 digits after the decimal point, and you only want 3. You think there's probably a function called `round` in Python, but you are not sure how to use it or how it works. You can get help directly from Python by typing `help(round)` at the command prompt. You'll see that this is indeed the function you were looking for:

```
round(2**0.5, 3)
```

gives the desired result.

In Spyder, there are additional ways to get help. Type `round` at the command prompt, but do not hit `<Return>`. Instead hit `<Cmd-I>` or `<Ctrl-I>` (for "**I**nformation"). The information that was displayed in the IPython console when you issued the `help` command now shows up in the Help tab, and in a format that is easier to navigate and read, especially for long entries. You can also use the Help tab without entering anything at the command prompt: Try entering `pow` in the "Object" field at the top of the pane. The Help tab provides information about an alternative to `**` for raising a number to a power.

In IPython, you can also follow or precede the name of any Python object, including function and variable names, by a question mark to obtain help: `round?` or `?round` provides almost the same information as `help(round)` and is easier to type.

When you type `help(...)`, Python will print out the information it has about the expression in parentheses if it recognizes the name. Unfortunately, Python is not as friendly if you don't know the name of the command you need. Perhaps you think there ought to be a way to take the square root of a number

[14] See Section 3.3.3 (page 43).

without using the power notation. After all, it is a pretty basic operation. Type `help(sqrt)` to see what happens when Python does not recognize the name you request.

To find out what commands are currently available to you, you can use Python's `dir()` command. This is short for "**dir**ectory," and it returns a list of all the modules, functions, and variable names that have been created or imported during the current session (or since the last `%reset` command). Ask Python for help on `dir` to learn more. Nothing in the output of `dir()` looks like a square root, but there is an item called `__builtins__`. This is the collection of all the functions and other objects that Python recognizes when it first starts up. It is Python's "last resort" when hunting for a function or variable.[15] To see the list of built-in functions, type

```
dir(__builtins__)
```

There is no `sqrt` function or anything like it. In fact, *none* of the standard mathematical functions, such as `sin`, `cos`, or `exp` show up!

Python cannot help you any further at this point. You now have to turn to outside resources. Good options include books about Python, search engines, friends who know more about Python than you do, and so on.

In the beginning, a lot of your coding time will be spent using a search engine to get help.

The `sqrt` function we seek belongs to a library. Later we will discuss how to access libraries of useful functions that are not automatically available with Python.

Your Turn 1A

Before proceeding, try a web search for
`how to take square roots in python`

1.2.4 Good practice: Keep a log

As you work through this tutorial, you will hit many small roadblocks—and some large ones. How do you evaluate a modified Bessel function? What do you do if you want a subscript in a graph axis label? The list is endless. Every time you resolve such a puzzle (or a friend helps you), *make a note of how you did it* in a notebook or in a dedicated file somewhere on your computer. Later, looking through that log will be much easier than scanning through all the code you wrote months ago (and less irritating than asking your friend over and over).

1.3 PYTHON MODULES

We discovered that Python does not have a built-in `sqrt` function. Even your calculator has that! What good is Python? Think for a moment about how, exactly, your calculator knows how to find square roots. At some point in the past, someone came up with an algorithm for computing the square root of a number and stored it in the permanent memory of your calculator. Someone had to create a *program* to calculate square roots.

[15] Appendix F explains how Python searches for variables and other objects.

Python is a programming language. A Python interpreter understands a basic set of commands that can be combined to perform complex tasks. Python also has a large community of developers who have created entire libraries of useful functions. To gain access to these, however, you need to **import** them into your working environment.

Use `import` to access functions that do not come standard with Python.

1.3.1 `import`

At the command prompt, type

```
import numpy
```

and hit `<Return>`. You now have access to many useful functions. You have imported the NumPy module, a collection of tools for numerical calculation using Python: "**Num**erical **Py**thon." (Do not capitalize its name in your code.)

To see what has been gained, type `dir(numpy)`. You will find nearly 600 new options at your disposal, and one of them is the `sqrt` function you originally sought. You can search for the function within NumPy by using the command `numpy.lookfor('sqrt')` (This will often return more than you need, but the first few lines can be quite helpful.) Now that you have imported NumPy, try

```
sqrt(2)
```

What's going on? You just imported a square root function, but Python tells you that `sqrt` is not defined! Try instead

```
numpy.sqrt(2)
```

The `sqrt` function you want "belongs" to the `numpy` module you imported. Even after importing, you still have to tell Python where to find it before you can use it.

After you have imported a module, call its functions by giving the module name, a period, and then the name of the desired function.

1.3.2 `from ... import`

There is another way to import functions. For example, you may wish access to all of the functions in NumPy without having to type the "`numpy.`" prefix before them. Try this:

```
from numpy import *
sqrt(2)
```

This is convenient, but it can lead to trouble when you want to use two different modules simultaneously. There is a module called `math` that also has a `sqrt` function. If you import all of the functions from `math` and `numpy`, which one gets called when you type `sqrt(2)`? (This is important when you are working with arrays of numbers.) To keep things straight, it is usually best to avoid the "`from module import *`" command. Instead, import a module and explicitly call `numpy.sqrt` or `math.sqrt` as appropriate. However, there is a middle ground. You can give a module any *nickname* you want. Try this:

```
import numpy as np
np.sqrt(2)
```

Now we can save typing and avoid confusion when functions from different modules have the same name.

There may be times when you only want a specific function, not a whole library of functions. You can ask for specific functions by name:

```
from numpy import sqrt, exp
sqrt(2)
exp(3)
```

We now have just two functions from the NumPy module, which can be accessed without the "`numpy.`" prefix. Notice the similarity with the "`from numpy import *`" command. The asterisk is a "wildcard" that tells the import command to grab everything.

One more useful variant of importing allows you to give the function you import a custom nickname:

```
from numpy.random import random as rand
rand()
```

We now have a **rand**om number generator with the convenient nickname `rand`.

This example also illustrates a module within a module: `numpy` contains the module `numpy.random`, which in turn contains the function `numpy.random.random`. When we typed `import numpy`, we imported many such subsidiary modules. Instead, we can import just one function by using `from` and providing a precise specification of the function we want, where to find it, and what to call it.

1.3.3 NumPy and PyPlot

The two modules we will use most often are called NumPy and PyPlot. NumPy provides the numerical tools we need to generate and analyze data, and PyPlot provides the tools we need to visualize data. PyPlot is a subset of the much larger Matplotlib library. From now on, we will assume that you have issued the following commands:

```
import numpy as np
import matplotlib.pyplot as plt
```

This can also be accomplished with the single command

```
import numpy as np, matplotlib.pyplot as plt
```

You should execute these commands at the start of every session. You should also add these lines at the beginning of any scripts that you write. You will also need to reimport both modules each time you use the `%reset` command.

Give the `%reset` command, then try importing these modules now. Explore some of the functions available from NumPy and PyPlot. You can get information about any of them by using `help()` or any of the procedures described in Section 1.2.3. You will probably find the NumPy help files considerably more informative than those for the built-in Python functions. They often include examples that you can try at the command prompt.

Now that we have these collections of tools at our disposal, let's see what we can do with them.

1.4 PYTHON EXPRESSIONS

The Python language has a **syntax**—a set of rules for constructing expressions and statements. In this section, we will look at some simple expressions to get an idea of how to communicate with Python. The basic building blocks of expressions are literals, variable names, operators, and functions.

1.4.1 Numbers

You can enter explicit numerical values (numeric **literals**) in various ways:

- `123` and `1.23` mean what you might expect. When entering a large number, however, don't separate groups of digits by commas. (Don't type `1,000,000` if you mean a million.)
- `2.3e5` is convenient shorthand for $2.3 \cdot 10^5$.
- `2+3j` represents the complex number $2 + 3\sqrt{-1}$. (Engineers may find the name j for $\sqrt{-1}$ familiar; mathematicians and physicists will have to adjust to Python's convention.)

Python stores numbers internally in several different formats. However, it will usually convert from one type to another when necessary. Beginners generally don't need to think about this. Just be aware that Python sometimes requires an integer. Even if a value has no fractional part, Python may not interpret it as an integer (for example, `a=1.0`). If you need to force a value to be an integer (for example, when indicating an entry in a list), you can use the functions **`int`** or **`round`**.

1.4.2 Arithmetic operations and predefined functions

Python includes basic arithmetic operators, for example, `+`, `-`, `*` (multiplication), `/` (division), and `**` (exponentiation).

Python uses two asterisks, `**`, to denote raising a number to a power.

For example, `a**2` means "`a` squared." (The notation `a^2` is used by some other math software but means something quite different to Python.)

Unlike standard mathematics notation, you must include multiplication signs. Try typing

```
(2)(3)
a = 2; a(3)
3a
3 a
```

Each command produces an error message. None, however, generates a message like, "`You forgot a '*'!`" Python used its evaluation rules, and these expressions didn't make sense. Python doesn't know what you were trying to express, so it can't tell you exactly what is wrong. *Study these error messages*; you'll probably see them again. See Appendix D for a description of these and other common errors.

Arithmetic operations have the usual precedence (ordering).

You can use parentheses to override operator precedence.

Unlike math textbooks, Python recognizes only parentheses (round brackets) for ordering operations. Square and curly brackets are reserved for other purposes. We have already seen that parentheses can also have another meaning (enclosing the arguments of a function). Yet another meaning will appear later: specifying a tuple. Python uses context to figure out which meaning to use.

For example, if you want to use the number $\frac{1}{2\pi}$, you might type `1/2*np.pi`. (Basic Python does not know the value of π, but NumPy does.) Try it. What goes wrong, and why? You can fix the expression by inserting parentheses. Later we'll meet other kinds of operators such as comparisons and logical operations. They, too, have a precedence ordering, which you may not wish to memorize. Instead, use parentheses liberally to specify precisely what you mean.

To get used to Python arithmetic operations, figure out what famous math problem these lines solve, and check that Python got it right:

```
a, b, c = 1, -1, -2
(-b + np.sqrt(b**2 - 4*a*c))/(2*a)
```

Recall that **`np.sqrt`** is the name of a *function* that Python does not recognize when it launches, but that becomes available once we import the NumPy module. When Python encounters the expression in the second line, it does the following:

1. Evaluate the **argument** of the **`np.sqrt`** function—that is, everything inside the pair of parentheses that follows the function name—by substituting values for variables and evaluating arithmetic operations. (The argument may itself contain functions.)
2. Interrupt evaluation of the expression and execute a piece of code named **`np.sqrt`**, handing that code the result found in step **1**.
3. Substitute the value returned by **`np.sqrt`** into the expression.
4. Finish evaluating the expression as usual.

How do you know what functions are available for you? See Section 1.2.3 above: Type `dir(np)` and `dir(__builtins__)` at the IPython console prompt.

A few symbols in Python and NumPy are predefined. These do not require any arguments or parentheses. Try **`np.pi`** (the constant π), **`np.e`** (the base of natural logarithms e), and `1j` (the constant $\sqrt{-1}$). NumPy also provides the standard trig functions, but be alert when using them:

The trig functions **`np.sin`**, **`np.cos`**, *and* **`np.tan`** *all treat their arguments as angles expressed in radians.*

1.4.3 Good practice: Variable names

Note that Python offers you no protection against accidentally changing the value of a symbol: If you say `np.pi=22/7`, then until you change it or reset Python, **`np.pi`** will have that value. It is even possible to create a variable whose name supplants a built-in function, for example, `round=3`.[16] This illustrates another good reason for using the "`import numpy as np`" command instead of the "`from numpy import *`" command: You are quite unlikely to use the "**`np.`**" prefix and name your *own* variables **`np.pi`** or **`np.e`**. Those variables retain their standard values no matter how you define `pi` and `e`.

When your code gets long, you may inadvertently reuse variable names. If you assign a variable with a generic name like `x` in the beginning, you may later choose the same name for some completely different

[16] This can be undone by deleting your version of `round`: Type `del(round)`. Python will revert to its built-in definition.

purpose. Later still, you will want the original `x`, having forgotten about the new one. Python will have overwritten the value you wanted, and puzzling behavior will ensue. You have a **name collision**.

It's good practice to use longer, more meaningful names for variables. They take longer to type, but they help avoid name collisions and make your code easier to read. Perhaps the first variable you were planning to call `x` could instead be called `index`, because it indexes a list. Perhaps the second variable you were planning to call `x` could logically be called `total`. Later, when you ask for `index`, there will be no problem.

Keep in mind, however, that "meaningful" in this context implies "meaningful to a human reader." Python itself pays no attention to the meaning of your variable names; for example, naming a variable `filename` will not tell Python how to use that variable.

Variable names are case sensitive, and most predefined names are lowercase. Thus, you can avoid some name collisions by including capital letters in variable or function names you define.

Blank spaces and periods are not allowed in variable names. Some coders use capitalization in the middle of variable names ("camel case") to denote word boundaries—for example, `whichItem`. Others use the underscore ("snake case") instead, as in `which_item`. Variable names may also contain digits (`myCount2`), but they must start with a letter.[17]

Some variable names are forbidden. Python won't let you name variables **`if`**, **`for`**, **`lambda`**, or a handful of other **reserved words**. You can find them with a web search for `python reserved words`.

1.4.4 More about functions

You may be accustomed to thinking of a function, for example, square root, as a machine that takes one number as input (its argument) and returns another number (its result) as output. Some Python functions do have this character, but Python has a much broader notion of function. Here are some illustrations. (Some involve functions that we have not seen yet.)

- A function may take a single argument, multiple arguments separated by commas, or none at all.
- A function may allow a *variable* number of arguments, and behave differently depending on how many you supply. For example, we will see functions that allow you to specify options by using *keyword arguments*. Each function's help text will describe the allowed ways of using it.
- A function may also *return* more than one value. The number of values returned can even vary depending on the arguments you supply. You can capture the returned values by using a special kind of assignment statement.[18]
- A function may change your computer's state in ways other than by returning a result. For example, **`plt.savefig`** saves a plot to a file on your computer's hard drive. Other possible side effects include writing text into the IPython console: `print('hello')`.

If you use a function name without parentheses, you are referring to the function instead of evaluating it. In mathematics, f is a function; $f(2)$ is the value of the function when its argument is 2. Type **`np.sqrt`** with no parentheses at the IPython command line to see how Python handles function names.

When evaluating a function, always include parentheses—even if there are no arguments.

[17] Strictly speaking, a name may also begin with the underscore character, but normally such names are reserved for Python's internal use.

[18] Section 6.1.4 (page 79) discusses the values returned by functions in more detail. $\boxed{T_2}$ More precisely, a Python function always returns a single object. However, that object may be a tuple that contains several items.

If a function accepts two or more arguments, how does it know which is which? In mathematical notation, the *order* of arguments conveys this information. For example, if we define $f(x, y) = x\,\mathrm{e}^{-y}$, then later $f(2, 6)$ means $2 \cdot \mathrm{e}^{-6}$: The first given value (2) gets substituted for the first named variable in the definition (x), and so on. This **positional argument** scheme is also the standard one used by Python. But when a function accepts many arguments, relying on order can be annoying and prone to error. For this reason, Python has an alternative approach called **keyword arguments**. For example,

```
f(y=6, x=2)
```

instructs Python to execute a function named `f`, initializing a variable named `y` with the value 6 and another named `x` with the value 2. You need not adhere to any particular order in giving keyword arguments. (However, keyword arguments must follow all positional arguments and you must use their correct names, which you can learn from the function's documentation.) Many functions will let you omit specifying values for some or all of their keyword arguments; if you omit them, the function supplies default values. Keyword arguments will be discussed further in Section 6.1.3.

You now know enough Python to do simple calculations. Try the examples from this chapter and play around on your own. In the next chapter, we will explore how to write simple programs in Python.

Index

Bold references indicate the main or defining instance of a key term.