

Contents

Let's Go

xiii

1 Getting Started with Python

1

- 1.1 Algorithms and algorithmic thinking 1
 - 1.1.1 Algorithmic thinking 1
 - 1.1.2 States 2
 - 1.1.3 What does `a = a + 1` mean? 3
 - 1.1.4 Symbolic versus numerical 3
- 1.2 Launch Python 4
 - 1.2.1 IPython console 6
 - 1.2.2 Error messages 10
 - 1.2.3 Sources of help 10
 - 1.2.4 Good practice: Keep a log 11
- 1.3 Python modules 11
 - 1.3.1 `import` 12
 - 1.3.2 `from ... import` 12
 - 1.3.3 NumPy and PyPlot 13
- 1.4 Python expressions 14
 - 1.4.1 Numbers 14
 - 1.4.2 Arithmetic operations and predefined functions 14
 - 1.4.3 Good practice: Variable names 15
 - 1.4.4 More about functions 16

2 Organizing Data

19

- 2.1 Objects and their methods 19
- 2.2 Lists, tuples, and arrays 21
 - 2.2.1 Creating a list or tuple 21
 - 2.2.2 NumPy arrays 21
 - 2.2.3 Filling an array with values 23
 - 2.2.4 Concatenation of arrays 24
 - 2.2.5 Accessing array elements 25
 - 2.2.6 Arrays and assignments 26
 - 2.2.7 Slicing 27
 - 2.2.8 Flattening an array 28

vii

- 2.2.9 Reshaping an array 28
- 2.2.10 **T₂** Lists and arrays as indices 29
- 2.3 Strings 29
 - 2.3.1 Raw strings 31
 - 2.3.2 Formatting strings with the `format()` method 31
 - 2.3.3 **T₂** Formatting strings with `%` 32

3 | Structure and Control

35

- 3.1 Loops 35
 - 3.1.1 `for` loops 35
 - 3.1.2 `while` loops 37
 - 3.1.3 Very long loops 37
 - 3.1.4 Infinite loops 37
- 3.2 Array operations 38
 - 3.2.1 Vectorizing math 38
 - 3.2.2 Matrix math 40
 - 3.2.3 Reducing an array 41
- 3.3 Scripts 42
 - 3.3.1 The Editor 42
 - 3.3.2 **T₂** Other editors 42
 - 3.3.3 First steps to debugging 43
 - 3.3.4 Good practice: Commenting 45
 - 3.3.5 Good practice: Using named parameters 47
 - 3.3.6 Good practice: Units 48
- 3.4 Contingent behavior: Branching 49
 - 3.4.1 The `if` statement 50
 - 3.4.2 Testing equality of floats 51
- 3.5 Nesting 52

4 | Data In, Results Out

53

- 4.1 Importing data 53
 - 4.1.1 Obtaining data 54
 - 4.1.2 Bringing data into Python 54
- 4.2 Exporting data 57
 - 4.2.1 Scripts 57
 - 4.2.2 Data files 58
- 4.3 Visualizing data 60
 - 4.3.1 The `plot` command and its relatives 60
 - 4.3.2 Log axes 63
 - 4.3.3 Manipulate and embellish 63
 - 4.3.4 Replacing curves 65
 - 4.3.5 **T₂** More about figures and their axes 65
 - 4.3.6 **T₂** Error bars 66

- 4.3.7 3D graphs 66
- 4.3.8 Multiple plots 67
- 4.3.9 Subplots 68
- 4.3.10 Saving figures 69
- 4.3.11 T_2 Using figures in other applications 70

5 First Computer Lab

71

- 5.1 HIV example 71
 - 5.1.1 Explore the model 71
 - 5.1.2 Fit experimental data 72
- 5.2 Bacterial example 73
 - 5.2.1 Explore the model 73
 - 5.2.2 Fit experimental data 73

6 Random Number Generation and Numerical Methods

75

- 6.1 Writing your own functions 75
 - 6.1.1 Defining functions in Python 76
 - 6.1.2 Updating functions 78
 - 6.1.3 Arguments, keywords, and defaults 78
 - 6.1.4 Return values 79
 - 6.1.5 Functional programming 80
- 6.2 Random numbers and simulation 81
 - 6.2.1 Simulating coin flips 82
 - 6.2.2 Generating trajectories 82
- 6.3 Histograms and bar graphs 83
 - 6.3.1 Creating histograms 83
 - 6.3.2 Finer control 85
- 6.4 Contour plots, surface plots, and heat maps 86
 - 6.4.1 Generating a grid of points 86
 - 6.4.2 Contour plots 86
 - 6.4.3 Surface plots 87
 - 6.4.4 Heat maps 88
- 6.5 Numerical solution of nonlinear equations 89
 - 6.5.1 General real functions 89
 - 6.5.2 Complex roots of polynomials 90
- 6.6 Solving systems of linear equations 91
- 6.7 Numerical integration 92
 - 6.7.1 Integrating a predefined function 92
 - 6.7.2 Integrating your own function 93
 - 6.7.3 Oscillatory integrands 94
 - 6.7.4 T_2 Parameter dependence 94
- 6.8 Numerical solution of differential equations 95
 - 6.8.1 Reformulating the problem 95

x

- 6.8.2 Solving an ODE 96
- 6.8.3 $\boxed{T_2}$ Parameter dependence 97
- 6.8.4 Other ODE solvers 98
- 6.9 Vector fields and streamlines 100
 - 6.9.1 Vector fields 100
 - 6.9.2 Streamlines 101

7 | Second Computer Lab 103

- 7.1 Generating and plotting trajectories 103
- 7.2 Plotting the displacement distribution 104
- 7.3 Rare events 105
 - 7.3.1 The Poisson distribution 105
 - 7.3.2 Waiting times 106

8 | Images and Animation 109

- 8.1 Image processing 109
 - 8.1.1 Images as NumPy arrays 109
 - 8.1.2 Saving and displaying images 110
 - 8.1.3 Manipulating images 110
- 8.2 Displaying data as an image 111
- 8.3 Animation 113
 - 8.3.1 Creating animations 113
 - 8.3.2 Saving animations 114
 - HTML movies 115
 - $\boxed{T_2}$ Using an encoder 117
 - 8.3.3 Conclusion 117

9 | Third Computer Lab 119

- 9.1 Convolution 119
 - 9.1.1 Python tools for image processing 120
 - 9.1.2 Averaging 121
 - 9.1.3 Smoothing with a Gaussian 121
- 9.2 Denoising an image 122
- 9.3 Emphasizing features 122
- 9.4 $\boxed{T_2}$ Image files and arrays 123

10 | Advanced Techniques 125

- 10.1 Dictionaries and generators 125
 - 10.1.1 Dictionaries 126
 - 10.1.2 Special function arguments 128
 - 10.1.3 List comprehensions and generators 129
- 10.2 Tools for data science 133
 - 10.2.1 Series and data frames with pandas 133

10.2.2	Machine learning with scikit-learn	135
10.2.3	Next steps	138
10.3	Symbolic computing	138
10.3.1	Wolfram Alpha	139
10.3.2	The SymPy library	141
10.3.3	Other alternatives	144
10.3.4	First passage revisited	144
10.4	Writing your own classes	148
10.4.1	A random walk class	148
10.4.2	When to use classes	155

Get Going	157
-----------	-----

A | Installing Python 159


A.1	Install Python and Spyder	159
A.1.1	Graphical installation	160
A.1.2	Command line installation	161
A.2	Setting up Spyder	162
A.2.1	Working directory	162
A.2.2	Interactive graphics	163
A.2.3	Script template	163
A.2.4	Restart	164
A.3	Keeping up to date	164
A.4	Installing FFmpeg	164
A.5	Installing ImageMagick	164

B | Command Line Tools 166

B.1	The command line	166
B.1.1	Navigating your file system	167
B.1.2	Creating, renaming, moving, and removing files	169
B.1.3	Creating and removing directories	169
B.1.4	Python and Conda	170
B.2	Text editors	171
B.3	Version control	172
B.3.1	How Git works	172
B.3.2	Installing and using Git	174
B.3.3	Tracking changes and synchronizing repositories	177
B.3.4	Summary of useful workflows	179
B.3.5	Troubleshooting	181
B.4	Conclusion	182

C | Jupyter Notebooks 183

C.1	Getting started	183
-----	-----------------	-----

C.1.1	Launch Jupyter Notebooks	183
C.1.2	Open a notebook	184
C.1.3	Multiple notebooks	184
C.1.4	Quitting Jupyter	185
C.1.5	 Setting the default directory	185
C.2	Cells	186
C.2.1	Code cells	186
C.2.2	Graphics	187
C.2.3	Markdown cells	187
C.2.4	Edit mode and command mode	187
C.3	Sharing	188
C.4	More details	188
C.5	Pros and cons	188

D | Errors and Error Messages 190

D.1	Python errors in general	190
D.2	Some common errors	191

E | Python 2 versus Python 3 194

E.1	Division	194
E.2	Print command	195
E.3	User input	195
E.4	More assistance	196

F | Under the Hood 197

F.1	Assignment statements	197
F.2	Memory management	199
F.3	Functions	199
F.4	Scope	200
F.4.1	Name collisions	202
F.4.2	Variables passed as arguments	203
F.5	Summary	203

G | Answers to “Your Turn” Questions 205

Acknowledgments	213
-----------------	-----

Recommended Reading	215
---------------------	-----

Index	217
-------	-----

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:

```
1 Put the key in the ignition.  
2 Turn the key until the car starts, then let go.  
3 Push the button on the shift lever and move it to "Reverse."  
4 ...
```

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 ...
```

2 Chapter 1 Getting Started with Python

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:

```
1 a = 100
2 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:

```
1 a = 1
2 a == 0
3 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 [2] Appendix F gives more precise information about the handling of assignment statements.

4 Chapter 1 Getting Started with Python

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 $a = 1, \dots$ ” then the reader knows that b equals zero; if later we say, “In the case $a = 2, \dots$ ” 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 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	<code><Return></code>	end line or run command
control	<code><Ctrl-C></code>	interrupt current Python command
command	<code><Cmd-V></code>	paste from the clipboard
option or alt	<code><Alt-Shift-R></code>	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).

6 Chapter 1 Getting Started with Python

Python	A computer programming language. A way to describe algorithms to a computer.
IPython	A Python <i>interpreter</i> : A computer application that provides a convenient, interactive mode for executing Python commands and programs.
Spyder	An <i>integrated development environment</i> (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 <i>distribution</i> : A single download that includes all of the above and provides access to many additional libraries for special purposes. It also includes a <i>package manager</i> 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.

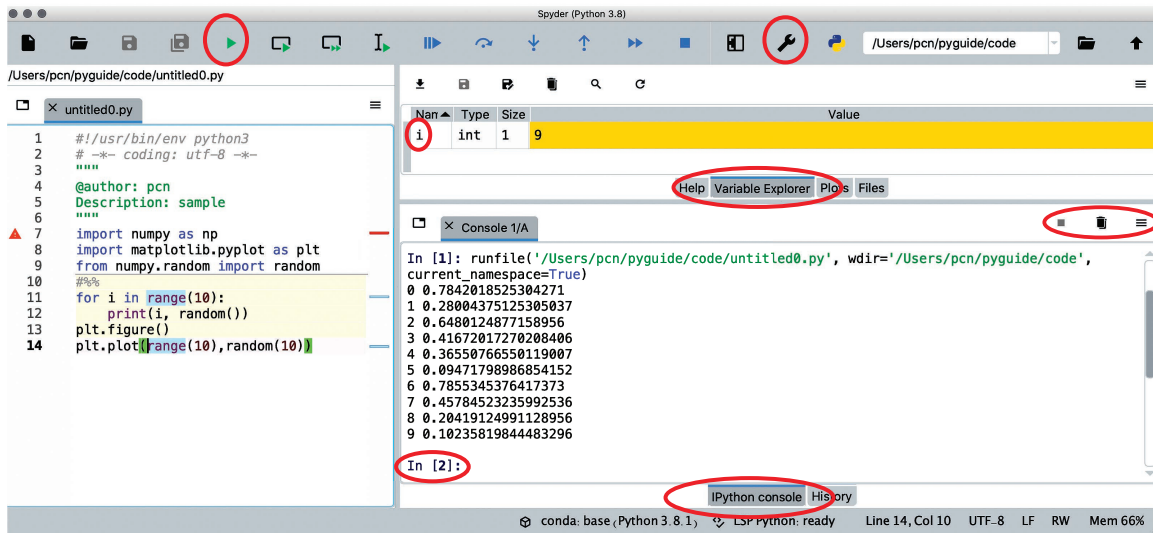


Figure 1.1: The Spyder display. Red circles have been added to emphasize (from top to bottom) the **RUN** button, Preferences (wrench icon), a variable in the Variable Explorer, the tab that brings the Variable Explorer to the front in its pane, the **STOP**, **RESET**, and **OPTIONS** buttons, the IPython 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>Panels** 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.

8 Chapter 1 Getting Started with Python

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\
1\
= 1 + \
2
```

This will create a variable `xva1` 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.

10 Chapter 1 Getting Started with Python

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 “Information”). 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 “directory,” 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.

12 Chapter 1 Getting Started with Python

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: “**Numerical Python**.” (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 **random** 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.

14 Chapter 1 Getting Started with Python

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.

16 Chapter 1 Getting Started with Python

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. ^[T2] 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 $f(x, y) = x e^{-y}$, then later $f(2, 6)$ means $2 \cdot e^{-6}$: 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.

^ (not used for exponentiation), 14
 ', *see* quote
 \, *see* backslash
 \\(enter backslash in a string), *see* backslash
 :, *see* colon
 =, *see* equals sign
 ==, *see* equals sign
 >=, 50
 <=, 50
 #, *see* commenting
 ##, *see* commenting
 -, *see* hyphen
 != (not equal), 50
 (), *see* parentheses
 {}, *see* brackets, curly
 [], *see* brackets, square
 %, *see* percent sign
 . . (command line), 168
 +, *see* plus sign
 +=, 46
 " " ", *see* commenting
 ;, *see* semicolon
 // (integer division), *see* arithmetic operators
 *, *see* star
 *=, 46
 **
 exponentiation, 14
 function argument, 129
 ~ (Boolean operation), 40
 ~ (command line), 168
 _ , *see* underscore
 CLOSE , 7
 OPTIONS , 7, 38
 RESET , 7
 RUN , 5, 7, 42, 43, 47, 78
 STOP , 7, 37, 38

A

a, 155
 abs, 29, 51, 76, 79, 190–193, 201, 202
 add_subplot, 152
 algorithm, 2
 align, 84, 85

all, 51
 alpha, 62, 71, 72
 Anaconda, 6, 141, 159–161, 164
 Navigator, 160
 Prompt, 166–168, 174
 and, 40, 50
 and (Boolean operator), 50, *see also* star
 angles, 15
 animation, 113–117
 frame, 113, 114
 animation, 113
 any, 51
 append, 149
 arange, 23, 24, 27–29
 arctan, 193
 args, 95, 98, 128, 129, 150
 argument, of a function, 15, 16, 17, 76, 79, *see also* keyword arguments
 double starred, 129
 starred, 129
 arithmetic operators, 14, *see also* plus, star
 integer division (//), 194
 remainder (%), 33, 37
 array, 23, 25, 28, 40, 41, 56, 80, 92, 123, 134, 149, 153, 210
 array, NumPy, 6, 8, 12, 20–29
 concatenation, 24–25
 creation from list, 23
 data attributes and methods, 22
 flattening, 28
 indexing, 25–26, *see also* index
 multidimensional, 22, 27, 39
 reduction, 38
 reshaping, 28–29
 shape, 22
 slicing, *see* slicing
 transpose, 101
 view, 28, 198
 assert, 44, 61, 193, 195
 AssertionError, 44, 193
 astype, 124
 attribute, 63

data, *see* data of an object
 method, *see* methods
 AttributeError, 192
 autoindent, 171
 axes
 current, *see* current axes
 equal scaling, 62, 103
 log-log, 63, 104
 object, *see* object
 semilog, 63, 104
 axes, 62, 69, 113, 116
 Axes3D, 66, 67, 85–88, 101, 114, 152, 208
 axis, 62, 100–103, 110, 112, 149, 158, 207
 azimuth, 67
 B
 back end, graphics, 60, 61, 63, 67, 163
 Qt, *see* Qt
 backslash, 30
 double (to enter backslash in a string), 30, 31, 64
 escape character in strings, 30, 59
 in raw strings, 55
 line continuation, 9
 bar, 84, 85
 bar3d, 85, 86
 base, 85
 Bernoulli trial, 139
 beta function, 139
 betainc, 139
 bin, 83
 edges, 208
 width, 208
 binding, 19
 bins, 83, 85, 136, 137, 150, 152, 154, 155
 bit depth, 109
 Bitbucket, 172, 173, 175, 180
 bitmap (image file format), 70, 109
 bool data type, 52, 209
 Boolean
 expression, 50

218 Index

operator, 50
 value, **29**, 50, 82
 boundary, 121
 boundary value, 143, 146
 Wolfram Alpha, 140
 brackets
 curly, 32, 36
 dictionary specification, 127
 round, *see* parentheses
 square, *see also* list
 comprehension
 indexing, *see* index
 list specification, 21
 branches, **176**
 branching, **49**
 break, 49, 50, 158
 broadcasting, 40
 bug, **1**
 theory, 44
`__builtins__`, 11, 15, 201

C

calculus (SymPy), 142
 camel case, 16
 canvas, 69
 Cauchy distribution, 139
 ccount, 87
 cd (command line), 167
 cell
 Jupyter, **183**
 Spyder Editor, **47**
 subplot, 68
 changeset, 173–174
 chisquare, 154
 choice, 151, 153
 circle, *see* graphs
 cla, 65
 clabel, 87
 class, **148**
 class, 149, 151, 153, 154
 close, 56, 59, 61, 68, 113, 116
 cmap, 87, 88, 110, 158
`coef_`, 136, 211
 coefficient of determination, 137
 coin flips, 82
 colon, 36, 76
 class definition, 149
 for loop, 35
 function definition, 76
 if, *elif*, *else*, 50
 in string formatting specification, 32
 key-value pair, 127
 plot line specifier, 62
 slicing, 27
 while loop, 37
 color, 62, 67

color, 64, 84, 101
 colorbar, 88
 colorbar, 88, 111
 colormaps, 121
 colors, 87
 comma
 as delimiter, 56, 58
 in list or tuple specification, 19, 21
 separated values file, *see* `.csv` file
 to separate arguments, 16
 command, 7
 command line, 166–170, *see also*
 specific commands
 command line interpreter, *see*
 IPython console
 command prompt, *see* IPython
 console
 commenting, 45–47
 comments, 55
 complex number, *see* number
 concatenation, *see* string *and* plus
 Conda, 170
 console, *see* IPython console
 constructor, **149**
 contour, 86–88
`contourf`, 101
 control structures, **35**
 convolution, 119–120, 139
 convolve, 120, 121, 123, 124
 convolve2d, 121
 copy, 81, 198
 copying lists and arrays, 198
 cos
 credible interval, **139**
 crunching sound, 2
`cstride`, 87
`.csv` files, 53, 135
`<Ctrl-C>`, 9
 cumsum, 83, 113, 126, 149, 207, 208
 current axes, 61, 63
 current figure, 61
 curve fitting, 72

D

dashed line, *see* graphs
`.dat` files, 53
 data
 fields, 63, 197
 in a class, 149
 of an object, **20**, 20–22, 61–64, 66, 110
 structures, 19
 DataFrame, 133–135, 209, 210
 DataFrame object (pandas), 134–135
 debugging, 43–45
 def, 76, 79, 81, 89–91, 93–96, 98, 99,

114, 115, 126, 139, 149, 151, 153–155, 200–202, 207–209
`default_rng`, 82, 113, 126, 131, 149, 207
Define once, and reuse often, 48, 55, 61, 75, 155
 del, 15, 199
 delimiter, 5, 55, 56, 58
 density, 102
 derivative (SymPy), 142
 describe, 134, 135
 det, 92
 df, 154
 dict, 127, 129, 130, 210
 dictionary, **58**, 126–129
 to create DataFrame, 134
 diff, 106
 differential equation, ordinary, *see*
 equations, solving
 dir, 11, 12, 15, 19, 20, 89, 92, 201
 directory, 42, **54**, 78, 116, 160, 161, 192
 working, **54**, 54, 55, 57, 58, 70, 72, 73, 78, 109, 121, 122, 162
 global, 78
 distribution, 6
 divide, 191
 division
 integer (`/`), 194
 Python 2 vs Python 3, 194
 division, 194
 docstring, **47**, **76**, 78
Don't duplicate, 48, 75
 dot, *see* graphs
 dot, 40, 41, 80, 92, 153, 205
 dot product, 40
 dotted line, *see* graphs
 double-starred expression, **129**
 draw, 67
 dropna, 136, 137
 dtype, 110, 158

E

`%edit`, 55, 171
 Editor, 5–7, 36, 42–43, 49, 54, 55, 78, 163
 eig, 92
`eigh`, 92
 elev, 67
 elif, 35, 49–51
 else, 35, 49–51, 79, 126, 192
 encapsulation, **156**
 encoder, video, 115, **117**
 enumerate, 132, 133, 153–155
 enumeration, **132**
`.eps` files, 69, 70, 109
 Equality (SymPy), 143

equals sign
 assignment, 3
 double (relation), 3, 50
 equations, solving
 linear, 91–92
 nonlinear, 89–91
 ordinary differential (ODE),
 95–97, 101–102
 SymPy, 143
 Wolfram Alpha, 140
 polynomial, 90
 error, *see also* exceptions
 messages, 10, 190–193
 runtime, 43
 syntax, 43
 error bars, *see* graphs
 errorbar, 66
 escape character, **30**
 except, 191
 exceptions, 5, 43, 76, 190–193, 201
 handling, 191
 exp, 13
 expm, 92
 exponential, 154
 exponentiation, 4
 exporting, *see* saving
 eye, 22, 151

F
 factorial, 39
 factorial, 5, 39, 106, 140, 205
 False, 3, 20, 29, 37, 40, 44, 50–52,
 82, 83, 126, 135
 family, 64
 fargs, 114
 features, 136
 FFmpeg, 117, 164
 Fick’s law, 146
 field
 data, 192, 193, 197
 lines (electric and magnetic), 101
 figsize, 112, 116
 figure, 66, 68, 69, 87, 88, 97, 103,
 113, 116, 120, 121, 132, 137,
 152, 206–208, 211
 figure window, 60, 61, 68, 103, 110
 current, 65
 saving, 69–70
 File Explorer, 7
 file output, 58–59
 text, 59
 fill, 81
 fillvalue, 121
 filter, 119
 Gaussian, 121, 123
 Laplace, 123
 Laplace of Gaussian, 123

square, 121
 fit, 136–138, 211
 flatten, 28
 flattening, *see* array, NumPy
 float data type, *see* number
 float, 19, 30, 49, 56, 203
 fmt, 66
 folder, *see* directory
 font, 64
 fontsize, 64, 87, 206
 for, 16, 35–39, 44, 49, 50
 for loop, *see* loop *and* list
 comprehension
 fork, **175**
 format, 31, 32, 35–37, 44, 46, 49,
 59, 61, 65, 116, 126, 127,
 132–134, 144, 150, 197, 198,
 202, 210, 211
 formatting strings, 32–33
 frame, video, *see* animation
 frames, 114
 fsolve, 89–91
 full_output, 90
 FuncAnimation, 113, 114
 function
 argument passing, 200
 nickname, 13
 returned value, 76
 scope, *see* scope
 updating, 78
 user defined, 75–81
 functional programming, 80–81
 fussy hygiene, 49
 __future__, 194–196

G
 gamma, 140
 garbage collection, 199
 gca, 63–65, 68, 206
 gcf, 68, 69, 110
 geeks, 122
 generator, **131**
 get_lines, 64, 65, 116
 get_supported_filetypes,
 69
 get_xticks, 63, 64
 get_yticks, 64
 __getitem__, 26
 .gif files, 70
 GIMP, 119
 Git, 172–182
 add, 176, 177
 branch, 176
 checkout, 177, 178, 181
 clone, 175
 commit, 177
 config, 175

init, 175
 log, 181
 merge, 178
 pull, 178
 push, 177, 178
 reset, 181
 revert, 182
 status, 176, 177
 GitHub, 172–178, 180
 gradient, 100, 101
 graphs, 84
 2D, 60
 3D, 66–67
 viewpoint, 67
 axis labels, 64
 colorbars, 88
 contour, 86–87
 error bars, 66
 font choice, *see* font
 heat map, 88
 histogram, *see* histogram
 legend, 31
 line color, *see* color
 line styles, 62, 64
 log axes, 63
 options, 61–65
 scatter plot, 62, 104
 surface, 87–88
 in SymPy, 144
 title, 31, 64
 updating, 114
 vector field, 100–101

H
 hard coding, 47
 hash sign, *see* commenting
 head, 134, 135
 heat map, 86
 Heisenbug, 19
 help, 10, 11, 13, 23
 Help tab, 7, 10
 hidden files, 166
 high-level language, 2
 hist, 68, 69, 83–85, 104, 106, 134,
 135, 150, 152, 154, 155, 208,
 210
 histogram, **83**
 higher-dimensional, 85
 with specified bin edges, 85
 histogram, 84, 136, 137, 208
 histogram_bin_edges, 84
 hstack, 25
 html_movie.py, 115
 hyphen
 in file name, 60
 plot line specifier, 62

220 Index

- I**
- i (imaginary number), *see* `j`
 - id, 198
 - IDE, *see* integrated development environment
 - if, 16, 35, 37, 49–52, 76, 79, 91, 126, 158, 192
 - if statement, **50**
 - imag, 20, 91
 - image
 - color, 109
 - display, subtlety with coordinates, 111–112
 - export, 110
 - import, 109
 - ImageMagick, 117
 - immutable object, *see* object
 - import, 5, 12, 13, 15
 - ImportError, 192
 - importing
 - code, **12**
 - data, 53–57
 - pandas, 135
 - importlib, 78, 152
 - impulse response, **120**
 - imread, 109, 121, 124
 - imsave, 109, 110
 - imshow, 109–112, 120, 122, 123, 158
 - indentation, 35–36, 47
 - index, 26
 - array, list, tuple, 25
 - Boolean list as, 29
 - integer list as, 29
 - negative, 26, 27
 - string, 25
 - index, 135
 - IndexError, 193
 - indexing, 86
 - inf, 44, 51, 94, 191, 209
 - infinite loop, **37**
 - inheritance, **151**, 151, 152, 155
 - `__init__`, 149
 - initial conditions, *see* boundary value
 - Inkscape, 70
 - input statement, Python 2 vs Python 3, 195
 - input, 49, 50, 194–196
 - instance of a class, 136, **150**, 150, 156
 - int data type, *see* number
 - int, 14, 19, 30
 - integrate, 92, 93, 95, 97–99, 208, 209
 - integrated development environment, 6, 159, 161, 191
 - integration, 92–94
 - analytic
 - SymPy, 142
 - Wolfram Alpha, 139–140
 - integration direction, 102
 - intercept, **136**
 - interpolation, 158
 - interpreter, 5, 6
 - interrupt, 9
 - inv, 92
 - ioff, 60
 - ion, 60
 - IPython, 6, 8, 10, 36
 - console, 2, 6–10, 12, 13, 15, 16
 - is, 198
 - is_integer, 20
 - isclose, 52, 91
 - item-by-item evaluation, 39–41, 82
 - items, 127
 - iterable, 80, **132**
- J**
- j (imaginary number), 15
 - j (special syntax for complex literals), 14
 - .jpg or .jpeg files, 69, 70, 109
 - Jupyter, 6, 159, 162, 167, 168, 170, 183–189
- K**
- Keras, 138
 - kernel, **186**
 - key (dictionary), **127**
 - key-value pair, **126**
 - KeyboardInterrupt, 37, 38
 - KeyError, 128
 - keys, 127, 133, 153–155
 - keyword arguments, 5, **17**, 55, 62, 64, 78–79, 83, 87, 88, 90, 94, 98
 - default values, 78
 - kind, 134, 135, 210
 - kwargs, 128, 129
- L**
- label, 64, 99, 129, 133, 137, 138, 206, 211
 - lambda, 16
 - \LaTeX , 31, 141, 147, 174, 187, 188, 215
 - lattice
 - triangular or honeycomb, 152
 - legend, **64**
 - legend, 64, 65, 99, 133, 137, 138, 206, 210, 211
 - len, 61, 81, 193, 200
 - limit, 94
 - limits (SymPy), 142
 - linalg, 91, 92
 - line numbers, 5
 - line object, *see* object
 - LinearRegression, 136, 138, 210
 - linestyle, 62, 64
 - linewidth, 64, 97, 102, 114, 129, 206
 - linewidths, 87
 - linspace, 24, 58, 60
 - list, 19, 21, 25, 40, 52, 80
 - comprehension, 130–131
 - indexing, 25–26
 - slicing, *see* slicing
 - list data type, *see* list
 - literal, **19**, 20
 - float, 19
 - list, 19, 21
 - numeric, **14**, 19
 - string, 19, **30**, 31
 - tuple, 21
 - load, 58
 - loadmat, 57
 - loadtxt, 5, 55–58, 121, 133
 - log, 44, 51, 137, 138, 210, 211
 - logical indexing, **29**
 - loglog, 63, 206
 - logspace, 63, 85
 - lookfor, 12
 - loop
 - for, 35–36
 - nested, **52**
 - while, 35, 37
 - low-level language, 2
 - ls (command line), 167
 - ls, 62
 - lw, 113, 114, 116
- M**
- machine learning, **136**
 - magic commands, **8**, 54, 55, 187
 - Maple, 144
 - Markdown, 174, 183–189
 - marker, 62, 72
 - for errorbar, 66
 - markerfacecolor, 62
 - markersize, 62, 114
 - math, 12, 39, 44
 - Mathematica*, 144
 - MATLAB, 6, 36, 53, 57, 99, 215
 - Matplotlib, 13, 42, 60
 - matrix, 22, 25
 - operations, 22, 40, 92
 - max, 21, 41, 84, 123, 137, 154, 208, 210
 - mean, 22, 41, 104, 110, 134, 210
 - mean-square displacement, **104**
 - meshgrid, 86, 100, 101, 111, 112, 123, 208

method, 99
 method of images, 147
 methods, **20**, 19–21, 63, 197
 mew, 211
 mfc, 211
 min, 41, 137, 208
 Miniconda, 159
 minlength, 102
 mkdir (command line), 167
 mode, 120, 123
 modularity, 203
 module, 12, **77**
 nickname, 12, 43
 user defined, 78
 modulo operator, *see* arithmetic
 operators, remainder
 movies, *see* animation
 mpl_toolkits, 66, 87, 152, 208
 mplot3d, 66, 87, 152, 208
 ms, 113, 114
 mutable object, *see* objects
 mv (command line), 167

N

\n (new line), 59
 name, **19**
 collision, **16**, 202, 203
 NameError, 10, 43, 193, 201
 namespace, **200**, 203
 built-in, 201
 enclosing, 201
 global, 201
 local, 200, 201
 nan, 44, 51, 126, 134, 135
 ndenumerate, 133
 ndi (nickname), *see* ndimage
 ndimage, 124
 ndindex, 133
 nesting, 52
 next, 131
 nickname, *see* module and function
 None, 52, 76, 79
 nonzero, 106, 126
 normal, 154
 not, 40, 50
 not (Boolean operator), 50, *see also* ~
 np (nickname), *see* NumPy
 .npy file, 58
 .npz file, 58
 num, 85
 number
 complex, 14
 floating-point (float), **19**, 203
 conversion from, to string, 31
 conversion to, from string, 30
 integer (int), **19**
 conversion from, to string, 31

 conversion to, from string, 30
 unsigned integer (uint8), 110,
 124, 209
 numerical error, 90, 92
 NumPy, 5, 6, 12, 13, 15, 16, *see also*
 individual function names
 array, *see* array

O

object, 19–21, 63
 Axes, 63, 65
 DataFrame, 134–135
 figure, 65
 immutable, **20**, 21
 line, 64, 65
 list, *see* list
 mutable, **20**, 200
 Series, 133–134
 tuple, 21
 object, 132
 object-oriented programming, **148**
 ODE, *see* equations
 odeint, 95–99
 ones, 22, 23, 25, 28, 120, 154, 200
 open, 55, 56, 59
 optimize, 89
 or, 40, 50
 or (Boolean operator), 50, *see also*
 plus sign
 origin, 111, 112
 overflow, 106
 overwriting an array, **81**

P

package, **159**
 package manager, 6, 117
 pandas, 133–135, 209, 210
 pandas, 53, 133, 134, 210
 panes in Spyder window, 6
 parameter
 actual, 77
 as used in this book, 46
 formal, 76
 in numerical integral, 94
 in ODE, 97
 parentheses (round brackets)
 function argument, *see* argument
 of function
 to override operator precedence,
 14
 tuple specification, *see* tuple
 pareto, 155
 pass by address, 200
 pass by reference, 200
 pass by value, 200
 path, **55**, 70, 78, **168**

pcolormesh, 88, 111, 112
 pd (nickname), *see* pandas
 .pdf files, 69, 70
 percent sign
 in header template, 163
 in string formatting specification,
 32
 magic commands, 8
 remainder, *see* arithmetic
 operators
 pi, 15, 32, 66, 67, 80, 93, 128, 132,
 153, 209
 pip, 6
 pivot, 100
 pixel, 109
 plot, *see also* graphs
 window, *see* figure window
 plot, 5, 7, 60–69, 72, 78, 93, 97, 99,
 103, 104, 113, 114, 116, 128,
 129, 133–135, 137, 138, 143,
 150, 152, 153, 155, 206–208,
 210, 211
 plot_surface, 87, 88, 121, 122,
 208
 plt (nickname), *see* PyPlot
 plus sign
 Boolean operation, 40
 concatenation, 30, 40
 .png files, 69, 70
 Poisson distribution, **105**, 106
 Poisson process, **106**
 poly1d, 91
 pop, 20
 positional argument scheme, **17**
 pow, 10, 59
 power, 186
 power series (SymPy), 143
 predict, 137, 138, 211
 Preferences menu, 54, 57, 162, 163
 print statement, Python 2 vs
 Python 3, 195
 print, 5, 8, 9
 probability density function, 92, 134,
 139, 145
 prod, 41
 projection, 152
 prompt, *see* IPython console
 property of an object, *see* data of an
 object
 pull, *see* Git
 push, *see* Git
 %pwd, 55
 pwd (command line), 167
 pyflakes, 43
 pylint, 43
 PyPlot, 5–7, 13, 16, *see also*
 individual function names

Q

Qt, 60
quad, 92–95, 97, 98, 140, 208, 209
quiver, 100, 101
quiver3d, 101
quote
 double, 30
 left, single (grave accent), 30
 right, single (apostrophe), 30
 triple, *see* commenting

R

R^2 value, 137
raise, 190
random, 13, 68, 69, 82, 83, 111, 113,
 123, 126, 131–133, 149, 154,
 207
random walk, 82–83, 103, 104, 113
range, 24, 36, 37
raster (image file format), 70, 109
ravel, 28, 198
raw string, **31**
rcount, 87
rcParams, 70
read_csv, 135
reducing an array, *see* array
reload, 78, 150, 152, 153, 155
repository, **172**, 173–178, 181
request, 55, 56
reserved words, 5, **16**
%reset, 8, 11, 13, 15, 42
reshape, 28, 29, 41, 136, 137, 198,
 210
return, 76, 77
reverse, 20
RGB color scheme, 109
RGBA color scheme, 109
rm (command line), 167
rmdir (command line), 167
root directory, **168**
roots
 of equation, *see* solving equations
 of a polynomial, 89
roots, 90, 91
round, 10, 14, 15
rstride, 87
Rubber Duck Debugging, 44
%run, 42, 54, 78, 171
Runge–Kutta method, 99
RuntimeWarning, 44, 191

S

Sage, 144
save, 58, 117
savefig, 16, 69, 116
savemat, 57

savetxt, 58
savez, 58
saving
 data and code, 57–60
 pandas, 135
 figures, 69–70
scalar quantity, 22
scale, 100, 101
scatter, 104
scatter plot, *see* graphs
scikit-learn, *see* sklearn
SciPy, 5, 6, 39
 .io, 53, 57
 .ndimage, 124
 .special, 5, 39, 106, 139, 140,
 205
scope of variables, 200–203
score, 137, 138, 211
scripts, 6, 9, 42–47
seaborn, 138
semicolon, 9
semilogx, 63
semilogy, 63, 206
Series, 133–135
Series object (pandas), 133–134
set_cmap, 87, 88, 101, 110
set_data, 61, 65, 114, 116
set_data_3d, 114
set_facecolor, 110
set_label, 64
set_title, 64, 112, 133, 152–155,
 206
set_xlabel, 63, 64, 206
set_xticklabels, 64
set_ylabel, 64, 206
set_yticklabels, 64
set_zlabel, 67
__setitem__, 26
setp, 64
shading, 88
shape, 22, 40, 61, 110, 149
sharex, 103, 133, 207
sharey, 103, 133, 207, 210
shot noise, 106
show, 187
side effect, 16, **81**, 114, 203
signal, 120, 121, 124
sim (nickname), *see*
 scipy.ndimage
sin, 15
singularity of a function, 90
size, 22, 27, 64, 67, 93, 126, 149,
 151, 153–155, 158, 208
__sizeof__, 131
skiprows, 55
sklearn, 133, 135–138, 210
slash, double (integer division), *see*

 arithmetic operators
slicing, 27–29, 198
snake case, 16
snapshot (Git), 173
solid line, *see* graphs
solve_ivp, 98, 99
sort, 81
split, 56
Spyder, 4, 6, 7, 9, 10, 20, 36, 38, 42,
 43, 47, 54, 55, 57, 69, 159–164,
 170
sqrt, 5
sqrtm, 92
staging, 177
star
 Boolean operation, 40
 command line, *see* wildcard
 function argument, *see* starred
 expression
starred expression, **129**
start_points, 102
std, 22, 41
StopIteration, 131
str data type, *see* string
str, 31, 32, 43
streamlines, 101–102
streamplot, 101, 102
stress fibers, 122
stride, 27
 surface plot, 87
string, 19, 25, 29–33, 80, 193
 accessing individual characters,
 25
 conversion from, to numeric, 30
 conversion to, from numeric, 31
 join (concatenate), 30
 literal, **30**
 raw, **31**, 55
 variable, 29
subplot, 68, 69, 103
subplots, 69, 103, 112, 121, 133,
 150, 152–155, 207
sum, 22, 41, 82, 149, 154
sum, analytic
 SymPy, 142
 Wolfram Alpha, 140
super, 151, 153, 154
suptitle, 68, 69, 112
surface plots, *see* graphs
svd, 92
.svg files, 70, 109
swapcase, 20
symbolic mathematics, 138, 141
SymPy, 4, 141–144
syntax, **14**
SyntaxError, 5, 190–192, 195

T

T, 101, 153
\t (tab), 59
t_eval, 99
tail, 135
tan, 15
targets, 136
TensorFlow, 138
text editor, 171–172
tick marks, 63, 64, 66, 110, 114
.tif or .tiff image files, 69, 70, 109, 121
tight_layout, 68
tilde, *see* ~
title, 64, 69, 206
to_csv, 135
training data, 136
transpose, *see* array
transpose, 112, 149, 151, 158
True, 3, 20
try, 191
.tsv files, 53
tuple, 16, **21**, 22, 25, 52, 77, 80, 83
 slicing, *see* slicing
.txt files, 53
type, 29, 30, 131
TypeError, 30, 76, 190, 193

U

uint8 (data type), *see* number
underscore, 16, 60, 84, 136, 150
 as temporary variable, **80**
units, 48–49
units, 100

unpacking, **80**
urllib, 55–57
urlopen, 55–57

V

value
 default (keyword), 78
 dictionary, **127**
 of an object, 3, 4, 8, 9, 15–17, *see*
 also data of an object, 20,
 199–204
 in a graph title, 31
 returned by a function, 16, 77,
 79–80
values, **127**
variable, **3**, **3**, **19**
 names, 16
Variable Explorer, 7, 20, 22, 43, 56,
 72, 110, 201
vector, 25
 column, 22, 23
 field, *see* graphs
 graphics, 70, 109
 row, 22
vectorizing math, 29, 38–40
 Boolean, 40
version control, 172–182
view of an array, *see* array
view_init, 67
viewpoint, **67**
viral load, **71**
vmax, 123
vmin, 123
vstack, 25, 41

W

waiting times, 106–107
weight, 64
while, 35, 37, 46, 47, 49–52, 76, 158,
 195
while loop, *see* loop
whitespace, 46, 47, 53
width, 84, 85
wildcard, 13, 116, **168**, 169, 179
window
 figure (or plot), *see* figure window
 Spyder, 6
Wolfram Alpha, 139–140
write, 59

X

xerr, 66
Xfig, 70
xlabel, 64, 65
xlim, 62, 103, 113, 116
xmax, 62
xmin, 62

Y

yerr, 66
ylabel, 64, 65
ylim, 62, 113, 116

Z

ZeroDivisionError, 190, 191
zeros, *see* equations
zeros, 21, 22, 25, 26