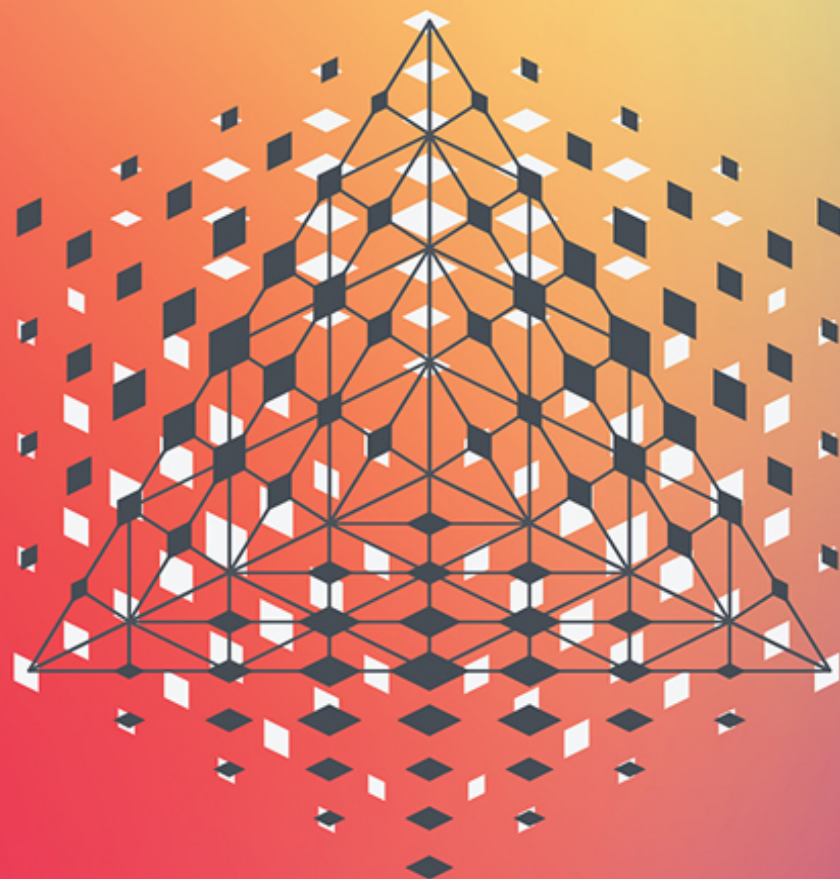


ADDISON WESLEY DATA & ANALYTICS SERIES



FOUNDATIONAL PYTHON FOR DATA SCIENCE



KENNEDY BEHRMAN

1

Introduction to Notebooks

All animals are equal, but some animals are more equal than others.

George Orwell

In This Chapter

- [Running Python statements](#)
- [Introduction to Jupyter notebooks](#)
- [Introduction to Google Colab hosted notebooks](#)
- [Text and code cells](#)
- [Uploading files to the Colab environment](#)
- [Using a system alias to run shell commands](#)
- [Magic functions](#)

This chapter introduces Google Colab's Jupyter notebook environment, which is a great way for a beginner to get started in scientific Python development. This chapter begins by looking at traditional ways of running Python code.

Running Python Statements


Historically, Python was invoked either in an interactive Python shell or by supplying text files to the interpreter. If you have Python installed on your

system, you can open the Python built-in interactive shell by typing `python` at the command line:

[Click here to view code image](#)

python

```
Python 3.9.1 (default, Mar  7 2021, 09:53:19)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more inform
```



Note

For code in this book, we use **bold text** for user input (the code you would type), and non-bold text for any output that results.

You can then type Python statements and run them by pressing Enter:

```
print("Hello")
Hello
```

As shown here, you see the result of each statement displayed directly after the statement's line.

When Python commands are stored in a text file with the extension `.py`, you can run them on the command line by typing **python** followed by the filename. If you have a file named `hello.py`, for example, and it contains the statement `print("Hello")`, you can invoke this file on the command line as follows and see its output displayed on the next line:

```
python hello.py
Hello
```

For traditional Python software projects, the interactive shell was adequate as a place to figure out syntax or do simple experiments. The file-based code was where the real development took place and where software was written. These files could be distributed to whatever environment needed to run the code. For scientific computing, neither of these solutions was ideal. Scientists wanted to have interactive engagement with data while still being

able to persist and share in a document-based format. Notebook-based development emerged to fill the gap.

Jupyter Notebooks

The IPython project is a more feature-rich version of the Python interactive shell. The Jupyter project sprang from the IPython project. Jupyter notebooks combine the interactive nature of the Python shell with the persistence of a document-based format. A notebook is an executable document that combines executable code with formatted text. A notebook is composed of *cells*, which contain code or text. When a code cell is executed, any output is displayed directly below the cell. Any state changes performed by a code cell are shared by any cells executed subsequently. This means you can build up your code cell by cell, without having to rerun the whole document when you make a change. This is especially useful when you are exploring and experimenting with data.

Jupyter notebooks have been widely adopted for data science work. You can run these notebooks locally from your machine or from hosted services such as those provided by AWS, Kaggle, Databricks, or Google.

Google Colab

Colab (short for Colaboratory) is Google's hosted notebook service. Using Colab is a great way to get started with Python, as you don't need to install anything or deal with library dependencies or environment management. This book uses Colab notebooks for all of its examples. To use Colab, you must be signed in to a Google account and go to <https://colab.research.google.com> (see [Figure 1.1](#)). From here you can create new notebooks or open existing notebooks. The existing notebooks can include examples supplied by Google, notebooks you have previously created, or notebooks you have copied to your Google Drive.

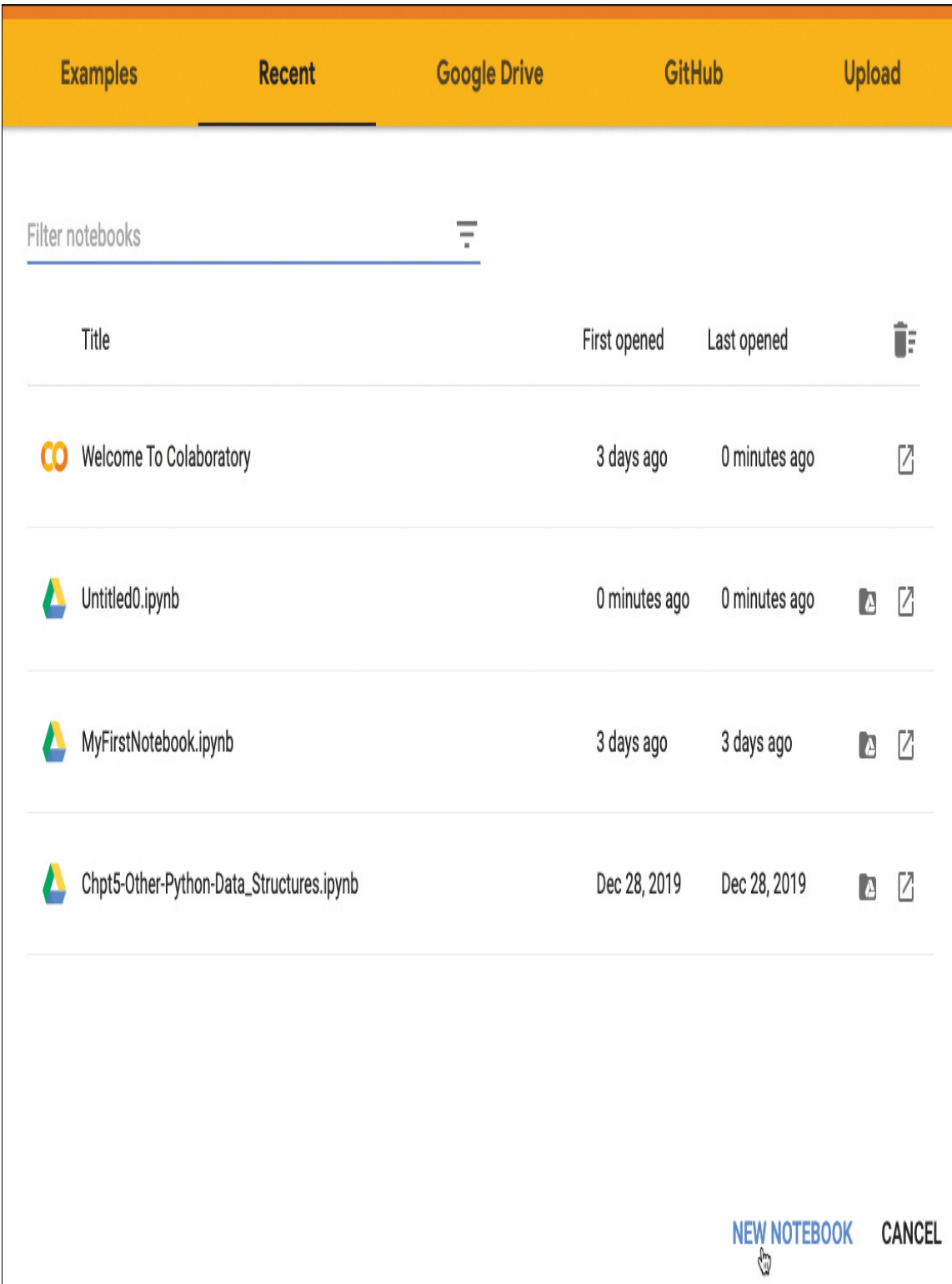


Figure 1.1 The Initial Google Colab Dialog

When you choose to create a new notebook, it opens in a new browser tab. The first notebook you create has the default title Untitled0.ipynb. To change its name, double-click on the title and type a new name (see [Figure 1.2](#)).

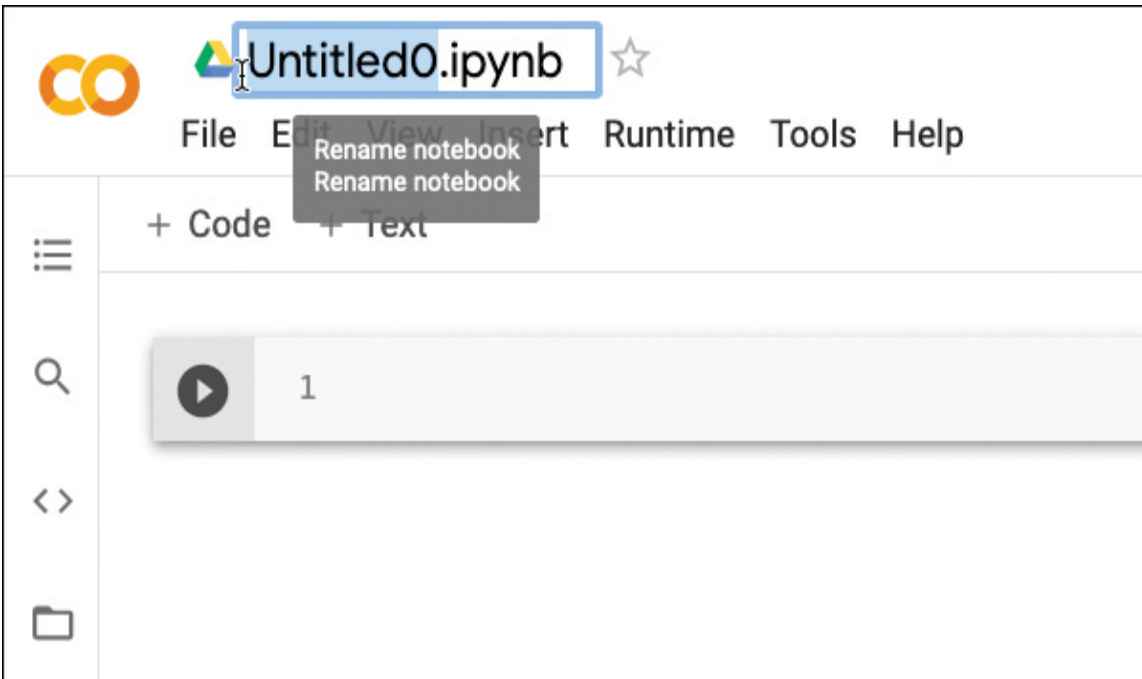


Figure 1.2 Renaming a Notebook in Google Colab

Colab automatically saves your notebooks to your Google Drive, which you can access by going to [Drive.Google.com](https://drive.google.com). The default location is a directory named Colab Notebooks (see [Figure 1.3](#)).





Name	↑
	books
	Colab Notebooks
	Digital Editions
	Music

Figure 1.3 The Colab Notebooks Folder at Google Drive

Colab Text Cells

A new Google Colab notebook has a single code cell. A cell can be one of two types: text or code. You can add new cells by using the + Code and + Text buttons in the upper left of the notebook interface.

Text cells are formatted using a language called Markdown. (For more information on Markdown, see https://colab.research.google.com/notebooks/markdown_guide.ipynb.) To edit a cell, you double-click it, and the Markdown appears to the right, with a preview of its output to the left (see [Figure 1.4](#)).

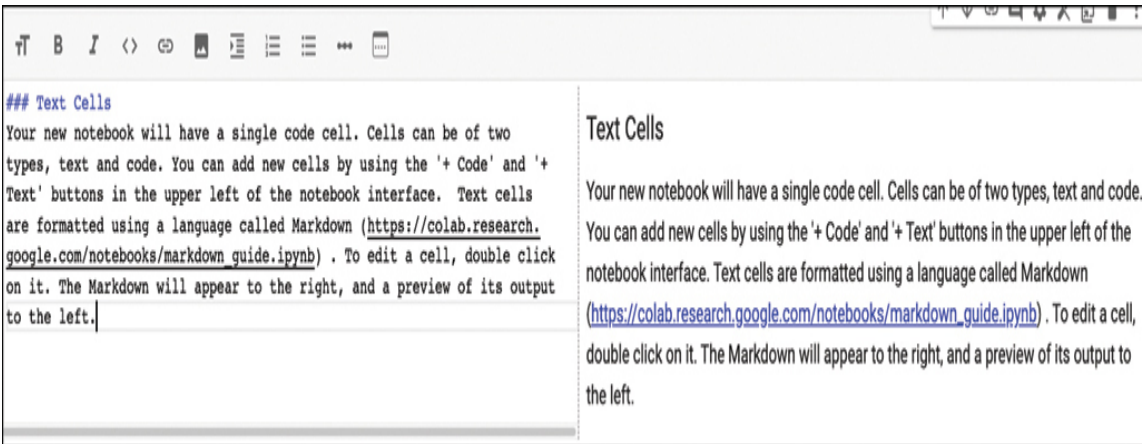


Figure 1.4 Editing Text Cells in a Google Colab Notebook

As shown in Figure 1.5, you can modify text in a notebook to be bold, italic, struck through, and monospaced.

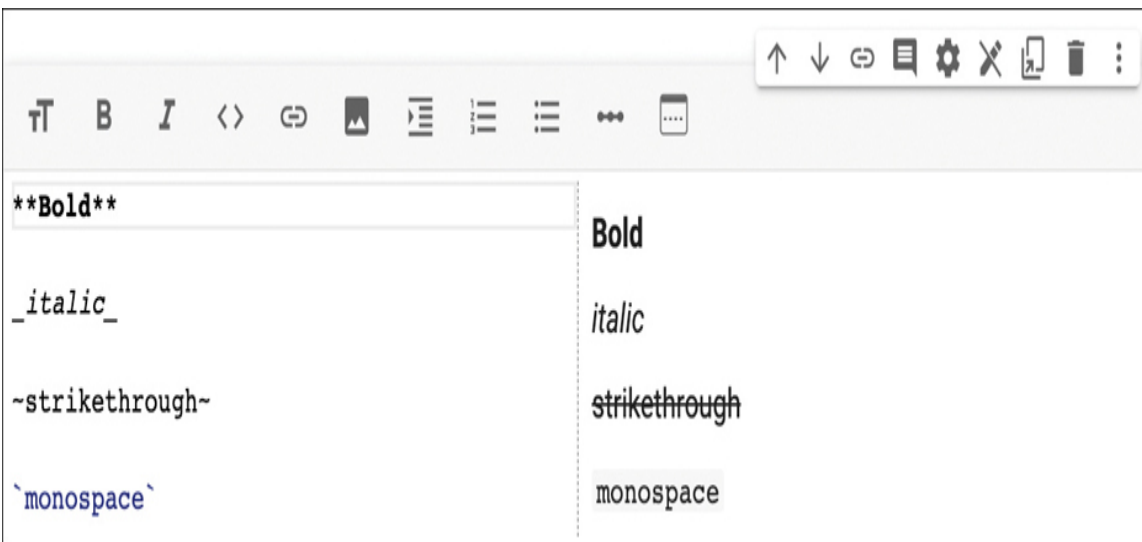


Figure 1.5 Formatting Text in a Google Colab Notebook

As shown in Figure 1.6, you can create a numbered list by prefacing items with numbers, and you can create a bulleted list by prefacing items with stars.

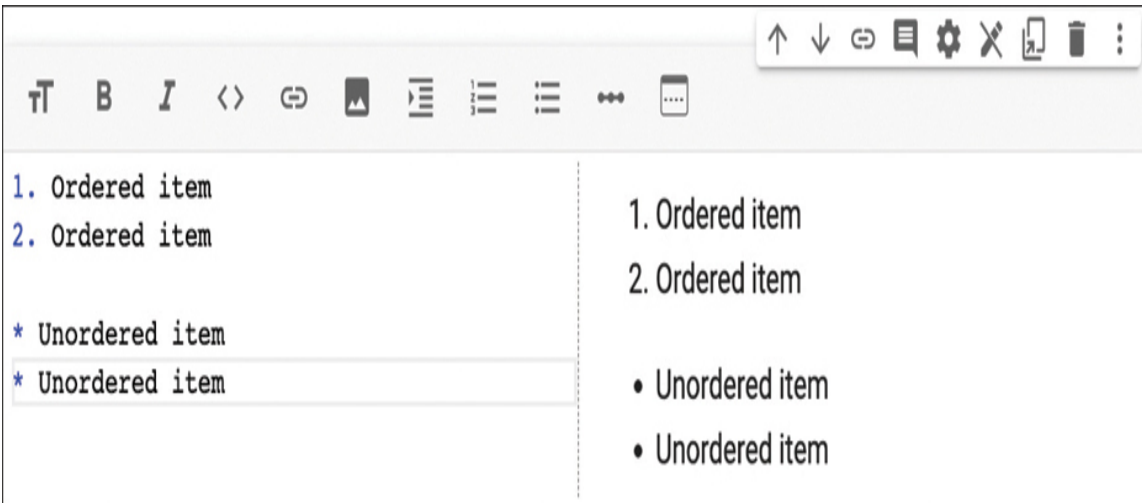


Figure 1.6 Creating Lists in a Google Colab Notebook

As shown in [Figure 1.7](#), you can create headings by preceding text with hash signs. A single hash sign creates a top-level heading, two hashes creates a first level heading, and so forth.

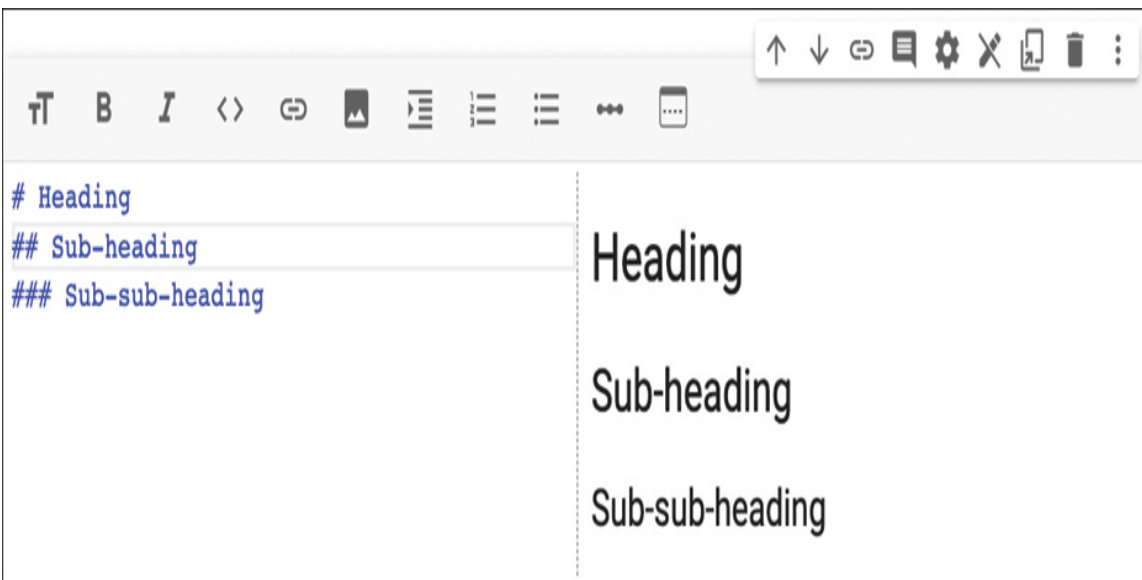


Figure 1.7 Creating Headings in a Google Colab Notebook

A heading that is at the top of a cell determines the cell’s hierarchy in the document. You can view this hierarchy by opening the table of contents,

which you do by clicking the Menu button at the top left of the notebook interface, as shown in [Figure 1.8](#).



Figure 1.8 The Table of Contents in a Google Colab Notebook

You can use the table of contents to navigate the document by clicking on the displayed headings. A heading cell that has child cells has a triangle next to the heading text. You can click this triangle to hide or view the child cells (see [Figure 1.9](#)).

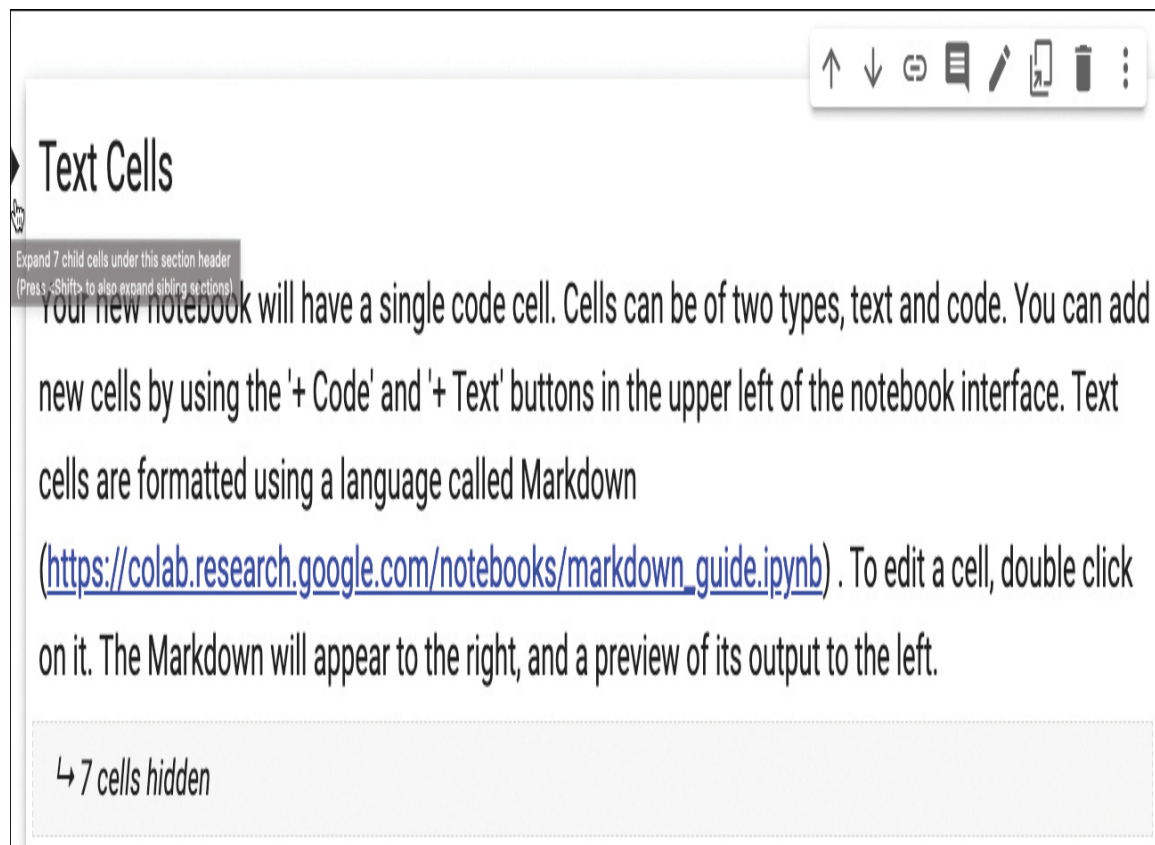


Figure 1.9 Hiding Cells in a Google Colab Notebook

LaTeX

The LaTeX language (see <https://www.latex-project.org/about/>), which is designed for preparing technical documents, excels at presenting mathematical text. LaTeX uses a code-based approach that is designed to allow you to concentrate on content rather than layout. You can insert LaTeX code into Colab notebook text cells by surrounding it with dollar signs. **Figure 1.10** shows an example from the LaTeX documentation embedded in a Colab notebook text cell.

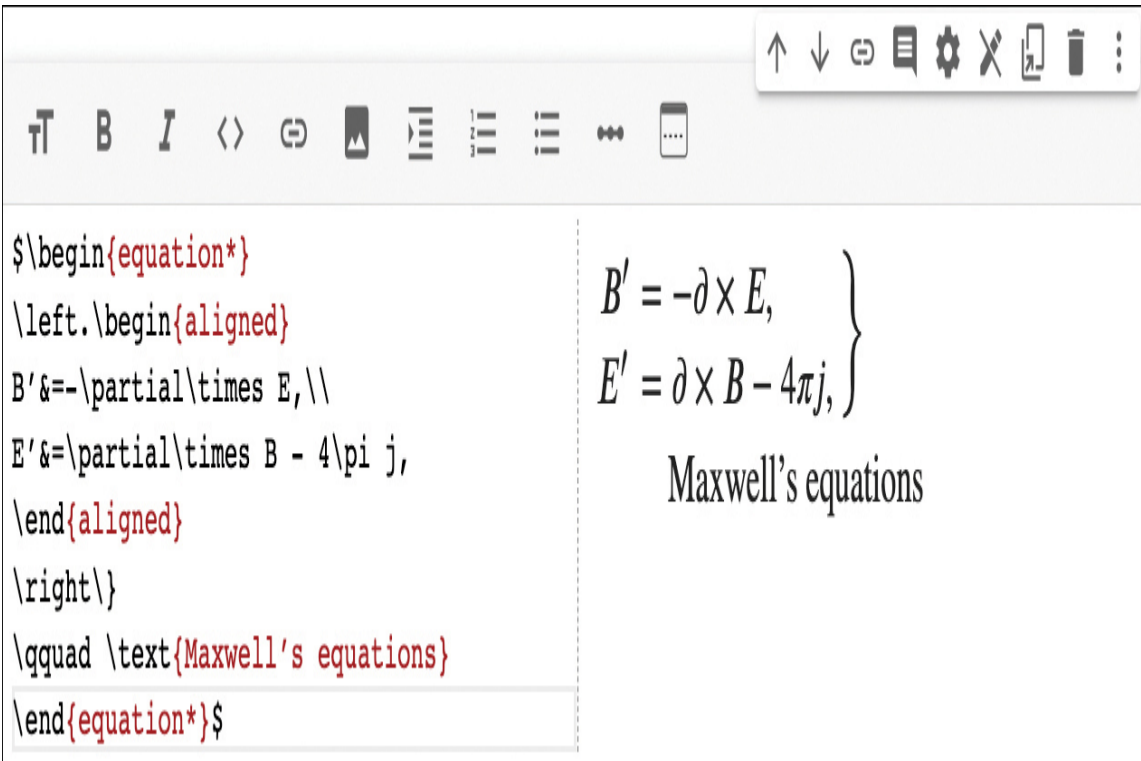


Figure 1.10 LaTeX Embedded in a Google Colab Notebook

Colab Code Cells

In Google Colab notebooks, you use code cells to write and execute Python code. To execute a Python statement, you type it into a code cell and either click the Play button at the left of the cell or press Shift+Enter. Pressing Shift+Enter takes you to the next cell or creates a new cell if there are none following. Any output from the code you execute is displayed below the cell, as in this example:

```
print("Hello")
hello
```

Subsequent chapters of this book use only code cells for Colab notebooks.

Colab Files

To see the files and folders available in Colab, click the Files button on the left of the interface (see [Figure 1.11](#)). By default, you have access to the `sample_data` folder supplied by Google.

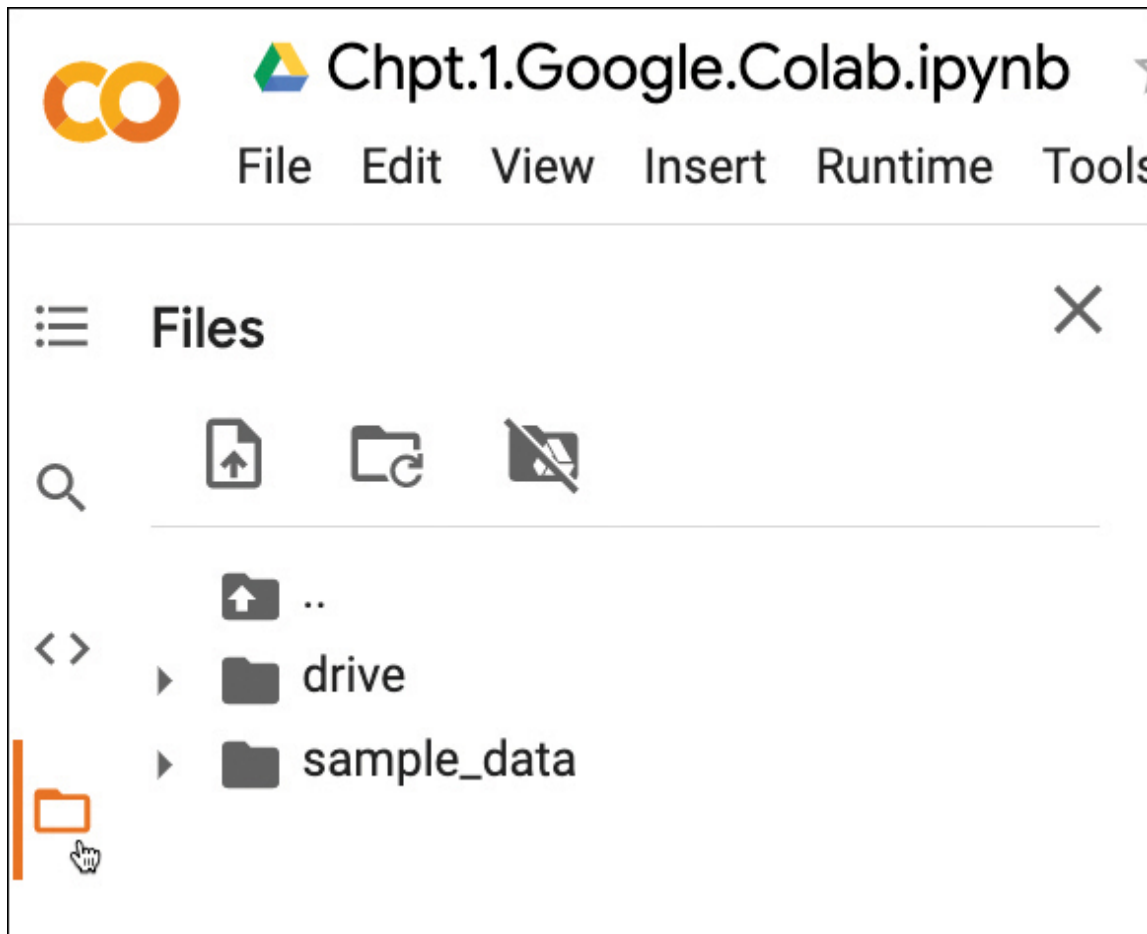


Figure 1.11 Viewing Files in Google Colab

You can also click the Upload button to upload files to the session (see [Figure 1.12](#)).

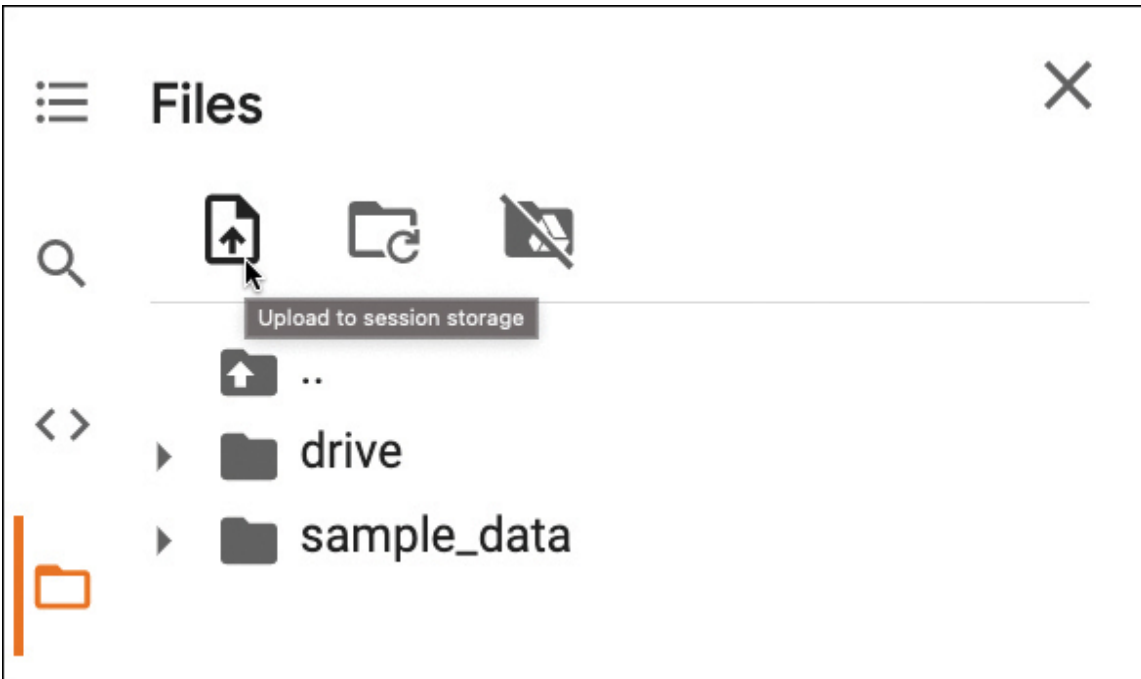


Figure 1.12 Uploading Files in Google Colab

Files that you upload are available only in the current session of your document. If you come back to the same document later, you need to upload them again. All files available in Colab have the path root `/content/`, so if you upload a file named `heights.over.time.csv`, its path is `/content/heights.over.time.csv`.

You can mount your Google Drive by clicking the Mount Drive button (see [Figure 1.13](#)). The contents of you drive have the root path `/content/drive`.

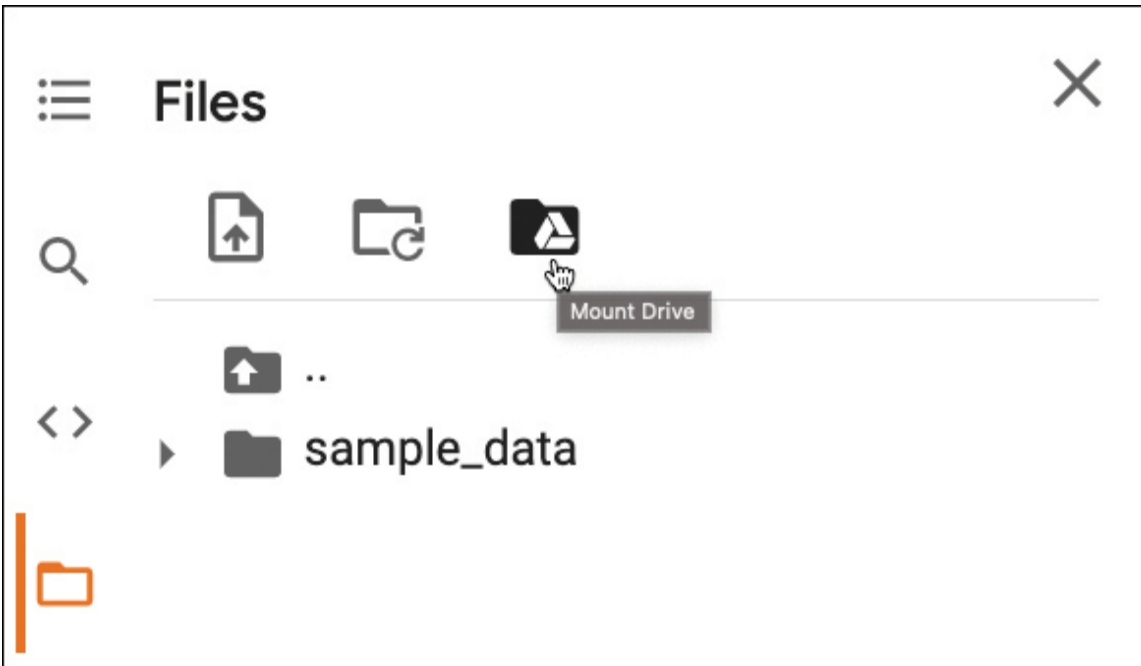


Figure 1.13 Mounting Your Google Drive

Managing Colab Documents

By default, notebooks are saved to your Google Drive. In the File menu you can see other options for saving notebooks. You can save them to GitHub, either as gists or as tracked files. You can also download them either in Jupyter notebook format (with the `.ipynb` extension) or as Python files (with the `.py` extension). You can also share notebooks by clicking the Share button in the upper right of the notebook interface.

Colab Code Snippets

The Code Snippets section of the left navigation section of Colab lets you search and select code snippets (see [Figure 1.14](#)). You can insert selected snippets by clicking the Insert button. Using code snippets is a great way to see examples of what can be done in Colab, including making interactive forms, downloading data, and using various visualization options.

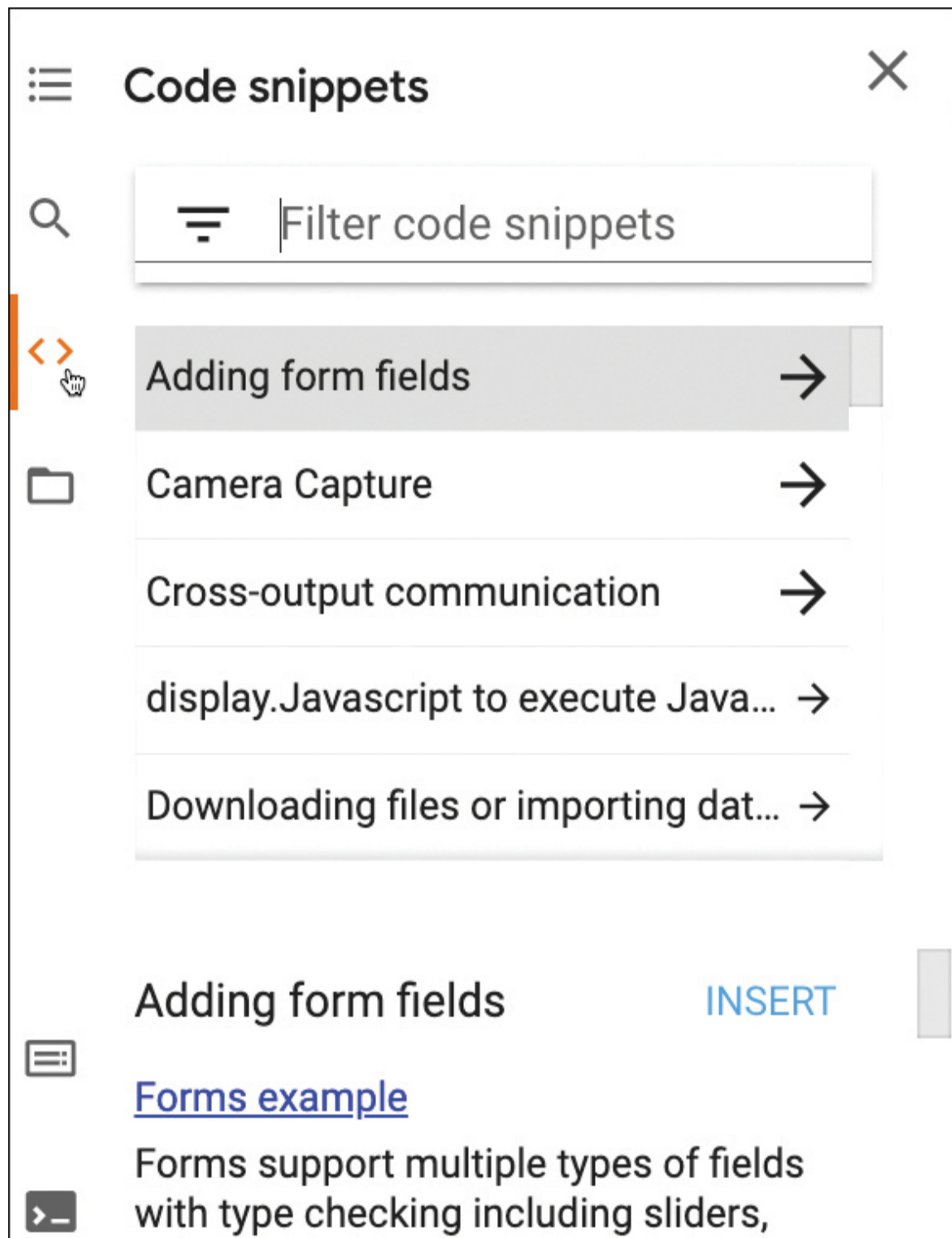


Figure 1.14 Using Code Snippets in Google Colab

Existing Collections

You can use Google Colab notebooks to explain and demonstrate techniques, concepts, and workflows. Data science work is shared in many collections of notebooks available around the web. Kaggle (see <https://www.kaggle.com/code>) has plenty of shared notebooks, as does the Google Seedbank (see <https://research.google.com/seedbank/>).

System Aliases

You can run a shell command from within a Colab notebook code cell by prepending the command with an exclamation point. For example, the following example prints the working directory:

```
!pwd  
/content
```

You can capture any output from a shell command in a Python variable, as shown here, and use it in subsequent code:

```
var = !ls sample_data  
print(var)
```

Note

Don't worry about variables yet. You will learn about them in [Chapter 2](#), “Fundamentals of Python.”

Magic Functions

Magic functions are functions that change the way a code cell is run. For example, you can time a Python statement by using the magic function `%timeit()` as shown here:

[Click here to view code image](#)

```
import time
%timeit(time.sleep(1))
```

As another example, you can have a cell run HTML code by using the magic function `%%html`:

[Click here to view code image](#)

```
%%html
<marquee style='width: 30%; color: blue;'><b>Whee!</b></marquee>
```



Note

You can find more information about magic functions in Cell Magics example notebooks that is part of the Jupyter documentation at <https://nbviewer.jupyter.org/github/ipython/ipython/blob/1.x/examples/notebooks/Cell%20Magics.ipynb>.

Summary

Jupyter notebooks are documents that combine formatted text with executable code. They have become a very popular format for scientific work, and many examples are available around the web. Google Colab offers hosted notebooks and includes many popular libraries used in data science. A notebook is made up of text cells, which are formatted in Markdown, and code cells, which can execute Python code. The following chapters present many examples of Colab notebooks.

Questions

1. What kind of notebooks are hosted in Google Colab?
2. What cell types are available in Google Colab?
3. How do you mount your Google Drive in Colab?
4. What language runs in Google Colab code cells?

2

Fundamentals of Python

All models are wrong, but some are useful.

George E.P. Box

In This Chapter

- [Python built-in types](#)
- [Introduction to statements](#)
- [Expression statements](#)
- [Assert statements](#)
- [Assignment statements and variables](#)
- [Import statements](#)
- [Printing](#)
- [Basic math operations](#)
- [Dot notation](#)

This chapter looks at some of the building blocks you can use to create a Python program. It introduces the basic built-in data types, such as integers and strings. It also introduces various simple statements you can use to direct your computer's actions. This chapter covers statements that assign values to variables and statements to ensure that code evaluates as expected. It also discusses how to import modules to extend the functionality available to you in your code. By the end of this chapter, you will have

enough knowledge to write a program that performs simple math operations on stored values.

Basic Types in Python

Biologists find it useful to organize living things into a hierarchy from domain and kingdom down to genus and species. The lower down this hierarchy you go, the more alike the life-forms that share a group. A similar hierarchy exists in data science.

A *parser* is a program that takes your code as input and translates it into instructions to a computer. The Python parser breaks up your code into tokens, which have particular meaning defined for the Python language. It is useful to group these tokens based on shared behaviors and attributes, much as biologists do with beings in nature. These groups in Python are called *collections* and *types*. There are types that are built into the language itself, and there are types that are defined by developers outside the language core. At a high level, the Python documentation (see <https://docs.python.org/3/library/stdtypes.xhtml>) defines the principal built-in types as numerics, sequences (see [Chapter 3](#), “Sequences”), mappings (see [Chapter 4](#), “Other Data Structures”), classes (see [Chapter 14](#), “Object-Oriented Programming”), instances (see [Chapter 14](#)), and exceptions. At a low level, the most basic built-in types are as follows:

- **Numerics:** Booleans, integers, floating point numbers, and imaginary numbers
- **Sequences:** Strings and binary strings

At the simplest, integers (or ints) are represented in code as ordinary digits. Floating point numbers, referred to as floats, are represented as a group of digits including a dot separator. You can use the `type` function to see the type of an integer and a float:

```
type(13)
int
```

```
type(4.1)
float
```

If you want a number to be a float, you must ensure that it has a dot and a number to the right, even if that number is zero:

```
type(1.0)  
float
```

Booleans are represented by the two constants, `True` and `False`, both of which evaluate to the type `bool`, which, behind the scenes, is a specialized form of `int`:

```
type(True)  
bool
```

```
type(False)  
bool
```

A string is characters surrounded by quotation marks. You can use strings to represent a variety of text, with many different uses. The following is one example:

```
type("Hello")  
str
```

Note

You will learn much more about strings and binary strings in [Chapter 4](#).

A special type, `NoneType`, has only one value, `None`. It is used to represent something that has no value:

```
type(None)  
NoneType
```

High-Level Versus Low-Level Languages

Writing software is, at its essence, just giving a computer instructions. The trick is to translate actions from a human-understandable form to instructions a computer can understand. Programming languages today

range from being very close to how a computer understands logic to being much closer to human language. The languages closer to the computer's instructions are referred to as *low-level languages*. Machine code and assembly language are examples of low-level languages. With these languages, you have the ultimate control over exactly what your computer's processor does, but writing code with them is tedious and time-consuming.

Higher-level languages abstract groups of instructions together into larger chunks of functionality. Different languages across this spectrum have unique strengths. For example, the language C, which is on the lower end of high-level languages, enables you to directly manage a program's use of memory and write the highly optimized software required for embedded systems. Python, in contrast, is on the upper end of high-level languages. It does not allow you to directly say how much memory to use to save your data or free that memory when you're done. Python's syntax is much closer to logic as defined in human language and is generally easier to understand and write than low-level languages. Translating actions from human language to Python is generally a fast and intuitive process.

Statements

A Python program is constructed of statements. Each statement can be thought of as an action that the computer should perform. If you think of a software program as being akin to a recipe from a cookbook, a statement is a single instruction, such as “beat the eggs yolks until they turn white” or “bake for 15 minutes.”

At the simplest, a Python statement is a single line of code with the end of the line signifying the end of the statement. A simple statement could, for example, call a single function, as in this expression statement:

```
print("hello")
```

A statement could also be more complicated, such as this statement, which evaluates conditions and assigns a variable based on that evaluation:

[Click here to view code image](#)


```
x,y = 5,6
bar = x**2 if (x < y) and (y or z) else x//2
```

Python allows for both simple and complex statements. Simple Python statements include expression, assert, assignment, pass, delete, return, yield, raise, break, continue, import, future, global, and nonlocal statements. This chapter covers some of these simple statements, and later chapters cover most of the rest of them. [Chapter 5](#), “Execution Control,” and [Chapter 6](#), “Functions,” cover complex statements.

Multiple Statements

While using a single statement is enough to define a program, most useful programs consist of multiple statements. The results of one statement can be used by the statements that follow, building functionality by combining actions. For example, you can use the following statement to assign a variable the result of an integer division, use that result to calculate a value for another variable, and then use both variables in a third statement as inputs to a print statement:

[Click here to view code image](#)

```
x = 23//3
y = x**2
print(f"x is {x}, y is {y}")
x is 7, y is 49
```

Expression Statements

A Python expression is a piece of code that evaluates to a value (or to None). This value could be, among other things, a mathematical expression or a call to a function or method. An expression statement is simply a statement that just has an expression but does not capture its output for further use. Expression statements are generally useful only in interactive environments, such as an IPython shell. In such an environment, the result of an expression is displayed to the user after it is run. This means that if you are in a shell and you want to know what a function returns or what 12344 divided by 12 is, you can see the output without coding a means to display it. You can also use an expression statement to see the value of a variable

(as shown in the following example) or just to echo the display value of any type. Here are some simple expression statements and the output of each one:

[Click here to view code image](#)

```
23 * 42
966
```

```
"Hello"
'Hello'
```

```
import os
os.getcwd()
'/content'
```

You will see a number of expression statements used in this book to demonstrate Python functionality. In each case, you will see the expression first, with its result on the following line.

Assert Statements

An assert statement takes an expression as an argument and ensures that the result evaluates to `True`. Expressions that return `False`, `None`, zero, empty containers, and empty strings evaluate to `False`; all other values evaluate to `True`. (Containers are discussed in [Chapter 3](#), “Sequences,” and [Chapter 4](#), “Other Data Structures.”) An assert statement throws an error if the expression evaluates to `False`, as shown in this example:

[Click here to view code image](#)

```
assert(False)
-----
AssertionError                                Traceback (most recent call last)
<ipython-input-5-8808c4021c9c> in <module>()
----> 1 assert(False)
```

Otherwise, the assert statement calls the expression and continues on to the next statement, as shown in this example:

```
assert(True)
```

You can use assert statements when debugging to ensure that some condition you assume to be true is indeed the case. These statements do have an impact on performance, though, so if you are using them generously when you develop, you might want to disable them when running your code in a production environment. If you are running your code from the command line, you can add the `-o`, optimize, flag to disable them:

```
python -o my_script.py
```

Assignment Statements

A variable is a name that points to some piece of data. It is important to understand that, in an assignment statement, the variable points to the data and is not the data itself. The same variable can be pointed to different items—even items that are of different types. In addition, you can change the data at which a variable points without changing the variable. As in the earlier examples in this chapter, a variable is assigned a value using the assignment operator (a single equals sign). The variable name appears to the left of the operator, and the value appears to the right. The following examples shows how to assign the value 12 to the variable `x` and the text `'Hello'` to the variable `y`:

```
x = 12
y = 'Hello'
```

Once the variables are assigned values, you can use the variable names in place of the values. So, you can perform math by using the `x` variable or use the `y` variable to construct a larger piece of text, as shown in this example:

[Click here to view code image](#)

```
answer = x - 3
print(f"{y} Jeff, the answer is {answer}")
Hello Jeff, the answer is 9
```

You can see that the values for `x` and `y` are used where the variables have been inserted. You can assign multiple values to multiple variables in a single statement by separating the variable names and values with commas:

```
x, y, z = 1, 'a', 3.0
```

Here `x` is assigned the value 1, `y` the value `'a'`, and `z` the value 3.0.

It is a best practice to give your variables meaningful names that help explain their use. Using `x` for a value on the x-axis of a graph is fine, for example, but using `x` to hold the value for a client's first name is confusing; `first_name` would be a much clearer variable name for a client's first name.

Pass Statements

Pass statements are placeholders. They perform no action themselves, but when there is code that requires a statement to be syntactically correct, a pass statement can be used. A pass statement consists of the keyword `pass` and nothing else. Pass statements are generally used for stubbing out functions and classes when laying out code design (that is, putting in the names without functionality). You'll learn more about functions in [Chapter 6](#), "Functions," and classes in [Chapter 14](#).

Delete Statements

A delete statement deletes something from the running program. It consists of the `del` keyword followed by the item to be deleted, in parentheses. Once the item is deleted, it cannot be referenced again unless it is redefined. The following example shows a value being assigned to a variable and then deleted:

[Click here to view code image](#)

```
polly = 'parrot'
del(polly)
print(polly)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-6-c0525896ade9> in <module>()
      1 polly = 'parrot'
      2 del(polly)
----> 3 print(polly)
```

```
NameError: name 'polly' is not defined
```

When you try to access the variable using a `print` function in this example, an error is raised.

Note

Python has its own garbage collection system, and, generally you don't need to delete objects to free up memory, but there may be times when you want to remove them anyway.

Return Statements

A return statement defines the return value of a function. You will see how to write functions, including using return statements, in [Chapter 6](#).

Yield Statements

Yield statements are used in writing generator functions, which provide a powerful way to optimize for performance and memory usage. We cover generators in [Chapter 13](#), “Functional Programming.”

Raise Statements

Some of the examples so far in this chapter have demonstrated code that causes errors. Such errors that occur during the running of a program (as opposed to errors in syntax that prevent a program from running at all) are called *exceptions*. Exceptions interrupt the normal execution of a program, and unless they are handled, cause the program to exit. Raise statements are used both to re-invoke an exception that has been caught and to raise either a built-in exception or an exception that you have designed specifically for your program. Python has many built-in exceptions, covering many different use cases (see

<https://docs.python.org/3/library/exceptions.xhtml#builtin-exceptions>). If you want to invoke one of these built-in exceptions, you can use a raise statement, which consists of the `raise` keyword followed by the exception. For example, `NotImplementedError` is an error used in class hierarchies to indicate that a child class should implement a method (see [Chapter 14](#)). The following example raises this error with a `raise` statement:

[Click here to view code image](#)

```
raise NotImplementedError
```

```
-----  
NotImplementedError      Traceback (most recent call last)  
<ipython-input-1-91639a24e592> in <module>()  
----> 1 raise NotImplementedError
```

Break Statements

You use a break statement to end a loop before its normal looping condition is met. Looping and break statements are covered in [Chapter 5](#).

Continue Statements

You use a continue statement to skip a single iteration of a loop. These statements are also covered in [Chapter 5](#).

Import Statements

One of the most powerful features of writing software is the ability to reuse pieces of code in different contexts. Python code can be saved in files (with the .py extension); if these files are designed for reuse, they are referred to as *modules*. When you run Python, whether in an interactive session or as a standalone program, some features are available as core language features, which means you can use them directly, without additional setup. When you install Python, these core features are installed, and so is the Python Standard Library. This library is a series of modules that you can bring into your Python session to extend functionality. To have access to one of these modules in your code, you use an import statement, which consists of the keyword `import` and the name of the module to import. The following example shows how to import the `os` module, which is used to interact with the operating system:

```
import os
```

Once `os` is imported, you can use the module's functionality as if it were built in. The `os` module has a `listdir` function that lists the contents of the current directory:

[Click here to view code image](#)

```
os.listdir()  
['.config', 'sample_data']
```

When modules or groups of modules are prepared for wider distribution, they are referred to as *packages*. One of the appealing aspects of Python, especially for data science, is the large ecosystem of third-party packages. These packages can be local to you or your organization, but the majority of public packages are hosted in the Python Package Index, pypi.org. To use one of these packages, you must install it first, generally by using `pip`, the standard package manager for Python. For example, to install the famously useful Pandas library for your local use, you run the following at the command line:

```
pip install pandas
```

Then you import it into your code:

```
import pandas
```

You can also give a module an alias during import. For example, it is a common convention to import Pandas as `pd`:

```
import pandas as pd
```

You can then reference the module by using the alias rather than the module name, as shown in this example:

[Click here to view code image](#)

```
pd.read_excel('/some_excel_file.xls')
```

You can also import specific parts of a module by using the `from` keyword with `import`:

[Click here to view code image](#)

```
import os from path  
path  
<module 'posixpath' from '/usr/lib/python3.6/posixpath.py'>
```


This example imports the submodule `path` from the module `os`. You can now use `path` in your program as if it were defined by your own code.

Future Statements

Future statements allow you to use certain modules that are part of a future release. This book does not cover them as they are rarely used in Data Science.

Global Statements

Scope in a program refers to the environment that shares definitions of names and values. Earlier you saw that when you define a variable in an assignment statement, that variable retains its name and value for future statements. These statements are said to share *scope*. When you start writing functions (in [Chapter 6](#)) and classes (in [Chapter 14](#)), you will encounter scopes that are not shared. Using a global statement is a way to share variables across scopes. (You will learn more about global statements in [Chapter 13](#).)

Nonlocal Statements

Using nonlocal statements is another way of sharing variables across scope. Whereas a global variable is shared across a whole module, a nonlocal statement encloses the current scope. Nonlocal statements are valuable only with multiple nested scopes, and you should not need them outside of very specialized situations, so this book does not cover them.

Print Statements

When you are working in an interactive environment such as the Python shell, IPython, or, by extension, a Colab notebook, you can use expression statements to see the value of any Python expression. (An *expression* is piece of code that evaluates to a value.) In some cases, you may need to output text in other ways, such as when you run a program at the command line or in a cloud function. The most basic way to display output in such situations is to use a print statement. By default, the `print` function outputs

text to the standard-out stream. You can pass any of the built-in types or most other objects as arguments to be printed. Consider these examples:

```
print(1)
1
```

```
print('a')
a
```

You can also pass multiple arguments, and they are printed on the same line:

```
print(1, 'b')
1 b
```

You can use an optional argument to define the separator used between items when multiple arguments are provided:

```
print(1, 'b', sep=' ->')
1->b
```

You can even print the `print` function itself:

[Click here to view code image](#)

```
print(print)
<built-in function print>
```

Performing Basic Math Operations

You can use Python as a calculator. Basic math operations are built into the core functionality. You can do math in an interactive shell or use the results of calculations in a program. The following are examples of addition, subtraction, multiplication, division, and exponentiation in Python:

```
2 + 3
5
```

```
5 - 6
-1
```

```
3*4  
12
```

```
9/3  
3.0
```

```
2**3  
8
```

Notice that division returns a floating-point number, even if integers are used. If you want to limit the result of division to integers, you can use a doubled forward slash, as shown in this example:

```
5//2  
2
```

Another handy operator is modulo, which returns the remainder of a division. To perform the modulo operation, you use the percent sign:

```
5%2  
1
```

Modulo is useful in determining whether one number is a factor of another (in which case the result is zero). This example uses the `is` keyword to test if the result of the modulo is zero:

```
14 % 7 is 0  
True
```

We will look at more math operations in Part II, “Data Science Libraries.”

Using Classes and Objects with Dot Notation

In [Chapter 14](#) you will learn about defining your own classes and objects. For now, you can think of an object as a bundling of functionality with data. The majority of things in Python have attributes or methods attached to them. To access an object’s attributes or methods (that is, functions attached to an object), you use dot syntax. To access an attribute, simply use a dot after the object’s name, followed by the attribute name.

The following example shows how to access the numerator attribute of an integer:

```
a_number = 2  
a_number.numerator
```

You access object methods in a similar way, but with parentheses following. The following example uses the `to_bytes()` method of the same integer:

[Click here to view code image](#)

```
a_number.to_bytes(8, 'little')  
b'\x02\x00\x00\x00\x00\x00\x00\x00'
```

Summary

Programming languages provide a means of translating human instructions to computer instructions. Python uses different types of statements to give a computer instructions, with each statement describing an action. You can combine statements together to create software. The data on which actions are taken is represented in Python by a variety of types, including both built-in types and types defined by developers and third parties. These types have their own characteristics, attributes, and, in many cases, methods that can be accessed using the dot syntax.

Questions

1. With Python, what is the output of `type(12)`?
2. When using Python, what is the effect of using `assert(True)` on the statements that follow it?
3. How would you use Python to invoke the exception `LastParamError`?
4. How would you use Python to print the string "Hello"?
5. How do you use Python to raise 2 to the power of 3?

3

Sequences

Errors using inadequate data are much less than those using no data at all.

Charles Babbage

In This Chapter

- [Shared sequence operations](#)
- [Lists and tuples](#)
- [Strings and string methods](#)
- [Ranges](#)

In [Chapter 2](#), “Fundamentals of Python,” you learned about collections of types. This chapter introduces the group of built-in types called *sequences*. A sequence is an ordered, finite collection. You might think of a sequence as a shelf in a library, where each book on the shelf has a location and can be accessed easily if you know its place. The books are ordered, with each book (except those at the ends) having books before and after it. You can add books to the shelf, and you can remove them, and it is possible for the shelf to be empty. The built-in types that comprise a sequence are lists, tuples, strings, binary strings, and ranges. This chapter covers the shared characteristics and specifics of these types.

Shared Operations

The sequences family shares quite a bit of functionality. Specifically, there are ways of using sequences that are applicable to most of the group members. There are operations that relate to sequences having a finite length, for accessing the items in a sequence, and for creating a new sequence based a sequence's content.

Testing Membership

You can test whether an item is a member of a sequence by using the `in` operation. This operation returns `True` if the sequence contains an item that evaluates as equal to the item in question, and it returns `False` otherwise. The following are examples of using `in` with different sequence types:

[Click here to view code image](#)

```
'first' in ['first', 'second', 'third']  
True
```

```
23 in (23,)  
True
```

```
'b' in 'cat'  
False
```

```
b'a' in b'ieojjza'  
True
```

You can use the keyword `not` in conjunction with `in` to check whether something is absent from a sequence:

```
'b' not in 'cat'  
True
```

The two places you are most likely to use `in` and `not in` are in an interactive session to explore data and as part of an `if` statement (see [Chapter 5](#), “Execution Control”).

Indexing

Because a sequence is an ordered series of items, you can access an item in a sequence by using its position, or *index*. Indexes start at zero and go up to one less than the number of items. In an eight-item sequence, for example, the first item has an index of zero, and the last item an index of seven.

To access an item by using its index, you use square brackets around the index number. The following example defines a string and accesses its first and last substrings using their index numbers:

[Click here to view code image](#)

```
name = "Ignatius"  
name[0]  
'I'
```

```
name[4]  
't'
```

You can also index counting back from the end of a sequence by using negative index numbers:

```
name[-1]  
's'
```

```
name[-2]  
'u'
```

Slicing

You can use indexes to create new sequences that represent subsequences of the original. In square brackets, supply the beginning and ending index numbers of the subsequence separated by a colon, and a new sequence is returned:

[Click here to view code image](#)

```
name = "Ignatius"  
name[2:5]  
'nat'
```


The subsequence that is returned contains items starting from the first index and up to, but not including, the ending index. If you leave out the beginning index, the subsequence starts at the beginning of the parent sequence; if you leave out the end index, the subsequence goes to the end of the sequence:

```
name[:5]  
'Ignat '
```

```
name[4:]  
'tius'
```

You can use negative index numbers to create slices counting from the end of a sequence. This example shows how to grab the last three letters of a string:

```
name[-3:]  
'ius'
```

If you want a slice to skip items, you can provide a third argument that indicates what to count by. So, if you have a list sequence of integers, as shown earlier, you can create a slice just by using the starting and ending index numbers:

[Click here to view code image](#)

```
scores = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,  
scores[3:15]  
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

But you can also indicate the step to take, such as counting by threes:

```
scores[3:15:3]  
[3, 6, 9, 12]
```

To count backward, you use a negative step:

```
scores[18:0:-4]  
[18, 14, 10, 6, 2]
```

Interrogation

You can perform shared operations on sequences to glean information about them. Because a sequence is finite, it has a length, which you can find by using the `len` function:

```
name = "Ignatius"
len(name)
8
```

You can use the `min` and `max` functions to find the minimum and maximum items, respectively:

[Click here to view code image](#)

```
scores = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
min(scores)
0
max(name)
'u'
```

These methods assume that the contents of a sequence can be compared in a way that implies an ordering. For sequence types that allow for mixed item types, an error occurs if the contents cannot be compared:

[Click here to view code image](#)

```
max(['Free', 2, 'b'])
-----
TypeError                                Traceback (most recent c
<ipython-input-15-d8babe38f9d9> in <module>()
----> 1 max(['Free', 2, 'b'])
TypeError: '>' not supported between instances of 'int' and 'str'
```

You can find out how many times an item appears in a sequence by using the `count` method:

```
name.count('a')
1
```

You can get the index of an item in a sequence by using the `index` method:

```
name.index('s')  
7
```

You can use the result of the `index` method to create a slice up to an item, such as a letter in a string:

[Click here to view code image](#)

```
name[:name.index('u')]  
'Ignati'
```

Math Operations

You can perform addition and multiplication with sequences of the same type. When you do, you conduct these operations on the sequence, not on its contents. So, for example, adding the list `[1]` to the list `[2]` will produce the list `[1,2]`, not `[3]`. Here is an example of using the plus (+) operator to create a new string from three separate strings:

[Click here to view code image](#)

```
"prefix" + "-" + "postfix"  
'prefix-postfix'
```

The multiplication (*) operator works by performing multiple additions on the whole sequence, not on its contents:

```
[0,2] * 4  
[0, 2, 0, 2, 0, 2, 0, 2]
```

This is a useful way of setting up a sequence with default values. For example, say that you want to track scores for a set number of participants in a list. You can initialize that list so that it has an initial score for each participant by using multiplication:

[Click here to view code image](#)

```
num_participants = 10  
scores = [0] * num_participants  
scores  
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Lists and Tuples

Lists and tuples are sequences that can hold objects of any type. Their contents can be of mixed types, so you can have strings, integers, instances, floats, and anything else in the same list. The items in lists and tuples are separated by commas. The items in a list are enclosed in square brackets, and the items in a tuple are enclosed in parentheses. The main difference between lists and tuples is that lists are mutable, and tuples are immutable. This means that you can change the contents of a list, but once a tuple is created, it cannot be changed. If you want to change the contents of a tuple, you need to make a new one based on the content of the current one. Because of the mutability difference, lists have more functionality than tuples—and they also use more memory.

Creating Lists and Tuples

You create a list by using the list constructor, `list()`, or by just using the square bracket syntax. To create a list with initial values, for example, simply supply the values in brackets:

[Click here to view code image](#)

```
some_list = [1,2,3]
some_list
[1, 2, 3]
```

You can create tuples by using the tuple constructor, `tuple()`, or using parentheses. If you want to create a tuple with a single item, you must follow that item with a comma, or Python will interpret the parentheses not as indicating a tuple but as indicating a logical grouping. You can also create a tuple without parentheses by just putting a comma after an item.

[Listing 3.1](#) provides examples of tuple creation.

Listing 3.1 Creating Tuples

[Click here to view code image](#)

```
tup = (1,2)
tup
```

```
(1, 2)
```

```
tup = (1, )  
tup  
(1, )
```

```
tup = 1, 2,  
tup  
(1, 2)
```

Warning

A common but subtle bug occurs when you leave a trailing comma behind an argument to a function. It turns the argument into a tuple containing the original argument. So the second argument to the function `my_function(1, 2,)` will be `(2,)` and not `2`.

You can also use the list or tuple constructors with a sequence as an argument. The following example uses a string and creates a list of the items the string contains:

[Click here to view code image](#)

```
name = "Ignatius"  
letters = list(name)  
letters  
['I', 'g', 'n', 'a', 't', 'i', 'u', 's']
```

Adding and Removing List Items

You can add items to a list and remove items from a list. To conceptualize how it works, think of a list as a stack of books. The most efficient way to add items to a list is to use the `append` method, which adds an item to the end of the list, much as you could easily add a book to the top of a stack. To add an item to a different position in the list, you can use the `insert` method, with the index number where you wish to position the new item as an argument. This is less efficient than using the `append` method as the other

items in the list may need to move to make room for the new item; however, this is typically an issue only in very large lists. [Listing 3.2](#) shows examples of appending and inserting.

Listing 3.2 Appending and Inserting List Items

[Click here to view code image](#)

```
flavours = ['Chocolate', 'Vanilla']  
flavours  
['Chocolate', 'Vanilla']  
  
flavours.append('SuperFudgeNutPretzelTwist')  
flavours  
['Chocolate', 'Vanilla', 'SuperFudgeNutPretzelTwist']  
  
flavours.insert(0, "sourMash")  
flavours  
['sourMash', 'Chocolate', 'Vanilla', 'SuperFudgeNutPretzelTwist']
```

To remove an item from a list, you use the pop method. With no argument, this method removes the last item. By using an optional index argument, you can specify a specific item. In either case, the item is removed from the list and returned.

The following example pops the last item off the list and then pops off the item at index 0. You can see that both items are returned when they are popped and that they are then gone from the list:

[Click here to view code image](#)

```
flavours.pop()  
'SuperFudgeNutPretzelTwist'  
  
flavours.pop(0)  
'sourMash'  
  
flavours  
['Chocolate', 'Vanilla']
```

To add the contents of one list to another, you use the extend method:

[Click here to view code image](#)

```
deserts = ['Cookies', 'Water Melon']
desserts
['Cookies', 'Water Melon']

desserts.extend(flavours)
desserts
['Cookies', 'Water Melon', 'Chocolate', 'Vanilla']
```

This method modifies the first list so that it now has the contents of the second list appended to its contents.

Nested List Initialization

There is a tricky bug that bites beginning Python developers. It involves combining list mutability with the nature of multiplying sequences. If you want to initialize a list containing four sublists, you might try multiplying a single list in a list like this:

[Click here to view code image](#)

```
lists = [[]] * 4
lists
[[], [], [], []]
```

This appears to have worked, until you modify one of the sublists:

[Click here to view code image](#)

```
lists[-1].append(4)
lists
[[4], [4], [4], [4]]
```

All of the sublists are modified! This is because the multiplication only initializes one list and references it four times. The references look independent until you try modifying one. The solution to this is to use a list comprehension (discussed further in [Chapter 13](#), “Functional Programming”):

[Click here to view code image](#)

```
lists = [[] for _ in range(4)]
lists[-1].append(4)
```

```
lists
[[], [], [], [4]]
```

Unpacking

You can assign values to multiple variables from a list or tuple in one line:

```
a, b, c = (1,3,4)
a
1

b
3

c
4
```

Or, if you want to assign multiple values to one variable while assigning single ones to the others, you can use a * next to the variable that will take multiple values. Then that variable will absorb all the items not assigned to other variables:

[Click here to view code image](#)

```
*first, middle, last = ['horse', 'carrot', 'swan', 'burrito', 'fly']
first
['horse', 'carrot', 'swan']

last
'fly'

middle
'burrito'
```

Sorting Lists

For lists you can use built-in sort and reverse methods that can change the order of the contents. Much like the sequence min and max functions, these

methods work only if the contents are comparable, as shown in these examples:

[Click here to view code image](#)

```
name = "Ignatius"
letters = list(name)
letters
['I', 'g', 'n', 'a', 't', 'i', 'u', 's']

letters.sort()
letters
['I', 'a', 'g', 'i', 'n', 's', 't', 'u']

letters.reverse()
letters
['u', 't', 's', 'n', 'i', 'g', 'a', 'I']
```

Strings

A string is a sequence of characters. In Python, strings are Unicode by default, and any Unicode character can be part of a string. Strings are represented as characters surrounded by quotation marks. Single or double quotations both work, and strings made with them are equal:

[Click here to view code image](#)

```
'Here is a string'
'Here is a string'

"Here is a string" == 'Here is a string'
True
```

If you want to include quotation marks around a word or words within a string, you need to use one type of quotation marks—single or double—to enclose that word or words and use the other type of quotation marks to enclose the whole string. The following example shows the word *is* enclosed in double quotation marks and the whole string enclosed in single quotation marks:

[Click here to view code image](#)

```
'Here "is" a string'
'Here "is" a string'
```

You enclose multiple-line strings in three sets of double quotation marks as shown in the following example:

[Click here to view code image](#)

```
a_very_large_phrase = """
Wikipedia is hosted by the Wikimedia Foundation,
a non-profit organization that also hosts a range of other projects
"""
```



With Python strings you can use special characters, each preceded by a backslash. The special characters include `\t` for tab, `\r` for carriage return, and `\n` for newline. These characters are interpreted with special meaning during printing. While these characters are generally useful, they can be inconvenient if you are representing a Windows path:

[Click here to view code image](#)

```
windows_path = "c:\\row\\the\\boat\\now"
print(windows_path)
```

```
ow heoat
      ow
```

For such situations, you can use Python's raw string type, which interprets all characters literally. You signify the raw string type by prefixing the string with an `r`:

[Click here to view code image](#)

```
windows_path = r"c:\\row\\the\\boat\\now"
print(windows_path)
c:\\row\\the\\boat\\now
```

As demonstrated in [Listing 3.3](#), there are a number of string helper functions that enable you to deal with different capitalizations.

Listing 3.3 String Helper Functions

[Click here to view code image](#)

```
captain = "Patrick Tayluer"
captain
'Patrick Tayluer'

captain.capitalize()
'Patrick tayluer'

captain.lower()
'patrick tayluer'

captain.upper()
'PATRICK TAYLUER'

captain.swapcase()
'pATRICK tAYLUER'

captain = 'patrick tayluer'
captain.title()
'Patrick Tayluer'
```

Python 3.6 introduced format strings, or f-strings. You can insert values into f-strings at runtime by using replacement fields, which are delimited by curly braces. You can insert any expression, including variables, into the replacement field. An f-string is prefixed with either an `F` or an `f`, as shown in this example:

[Click here to view code image](#)

```
strings_count = 5
frets_count = 24
f"Noam Pikelny's banjo has {strings_count} strings and {frets_cc
'Noam Pikelny's banjo has 5 strings and 24 frets'
```

This example shows how to insert a mathematic expression into the replacement field:

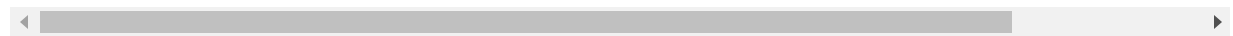
[Click here to view code image](#)

```
a = 12
b = 32
f"{a} times {b} equals {a*b}"
'12 times 32 equals 384'
```

This example shows how to insert items from a list into the replacement field:

[Click here to view code image](#)

```
players = ["Tony Trischka", "Bill Evans", "Alan Munde"]
f"Performances will be held by {players[1]}, {players[0]}, and {"
'Performances will be held by Bill Evans, Tony Trischka, and Al
```



Ranges

Using range objects is an efficient way to represent a series of numbers, ordered by value. They are largely used for specifying the number of times a loop should run. [Chapter 5](#) introduces loops. Range objects can take start (optional), end, and step (optional) arguments. Much as with slicing, the start is included in the range, and the end is not. Also as with slicing, you can use negative steps to count down. Ranges calculate numbers as you request them, and so they don't need to store more memory for large ranges. [Listing 3.4](#) demonstrates how to create ranges with and without the optional arguments. This listing makes lists from the ranges so that you can see the full contents that the range would supply.

Listing 3.4 Creating Ranges

[Click here to view code image](#)

```
range(10)
range(0, 10)

list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]

list(range(0,10,2))
[0, 2, 4, 6, 8]
```

```
list(range(10, 0, -2))  
[10, 8, 6, 4, 2]
```

Summary

This chapter covers the import group of types known as sequences. A sequence is an ordered, finite collection of items. Lists and tuples can contain mixed types. Lists can be modified after creation, but tuples cannot. Strings are sequences of text. Range objects are used to describe ranges of numbers. Lists, strings, and ranges are among the most commonly used types in Python.

Questions

1. How would you test whether `a` is in the list `my_list`?
2. How would you find out how many times `b` appears in a string named `my_string`?
3. How would you add `a` to the end of the list `my_list`?
4. Are the strings `'superior'` and `"superior"` equal?
5. How would you make a range going from 3 to 13?

4

Other Data Structures

Statistical thinking will one day be as necessary for efficient citizenship as the ability to read and write.

Samuel S. Wilks

In This Chapter

- [Creating dictionaries](#)
- [Accessing and updating dictionary contents](#)
- [Creating sets](#)
- [Set operations](#)

The order-based representation of data is powerful, but other data representations are also possible. Dictionaries and sets are data structures that do not rely on the order of the data. Both are powerful models that are integral to the Python toolbox.

Dictionaries

Imagine that you are doing a study to determine if there is a correlation between student height and grade point average (GPA). You need a data structure to represent the data for an individual student, including the person's name, height, and GPA. You could store the information in a list or tuple. You would have to keep track of which index represented which piece of data, though. A better representation would be to label the data so that you wouldn't need to track the translation from index to attribute. You

can use dictionaries to store data as key/value pairs. Every item, or value, in a dictionary is accessed using a key. This lookup is very efficient and is much faster than searching a long sequence.

With a key/value pair, the key and the value are separated with a colon. You can present multiple key/value pairs, separated by commas and enclosed in curly brackets. So, a dictionary for the student record might look like this:

[Click here to view code image](#)

```
{ 'name': 'Betty', 'height': 62, 'gpa': 3.6 }
```

The keys for this dictionary are the strings 'name', 'height', and 'gpa'. Each key points to a piece of data: 'name' points to the string 'Betty', 'height' points to the integer 62, and 'gpa' points to the floating point number 3.6. The values can be of any type, though there are some restrictions on the key type, as discussed later in the chapter.

Creating Dictionaries

You can create dictionaries with or without initial data. You can create an empty dictionary by using the `dict()` constructor method or by simply using curly braces:

[Click here to view code image](#)

```
dictionary = dict()  
dictionary  
{}
```

```
dictionary = {}  
dictionary  
{}
```

The first example creates an empty dictionary by using the `dict()` constructor method and assigns that dictionary to a variable named `dictionary`. The second example creates an empty dictionary by using curly braces and also assigns to the same variable. Each of these examples produces an empty dictionary, represented by empty curly braces.

You can also create dictionaries initialized with data. One option for doing this is to pass in the keys and values as named parameters as in this example:


[Click here to view code image](#)

```
subject_1 = dict(name='Paula', height=64, gpa=3.8, ranking=1)
```

An alternative is to pass in the key/value pairs to the constructor as a list or tuple of lists or tuples, with each sublist being a key/value pair:

[Click here to view code image](#)

```
subject_2 = dict(['name', 'Paula'], ['height', 64], ['gpa', 3.8], ['
```



A third option is to create a dictionary by using curly braces, with the keys and values paired using colons and separated with commas:

[Click here to view code image](#)

```
subject_3 = {'name': 'Paula', 'height': 64, 'gpa': 3.8, 'ranking': 1
```



These three methods all create dictionaries that evaluate the same way, as long as the same keys and values are used:

[Click here to view code image](#)

```
subject_1 == subject_2 == subject_3
True
```

Accessing, Adding, and Updating by Using Keys

Dictionary keys provide a means to access and change data. You generally access data by the relevant key in square brackets, in much the way you access indexes in sequences:

[Click here to view code image](#)

```
student_record = {'name': 'Paula', 'height': 64, 'gpa': 3.8}
student_record['name']
```



```
'Paula'
```

```
student_record['height']  
64
```

```
student_record['gpa']  
3.8
```

If you want to add a new key/value pair to an existing dictionary, you can assign the value to the slot by using the same syntax:

[Click here to view code image](#)

```
student_record['applied'] = '2019-10-31'  
student_record  
{'name': 'Paula',  
  'height': 64,  
  'gpa': 3.8,  
  'applied': '2019-10-31'}
```

The new key/value pair is now contained in the original dictionary.

If you want to update the value for an existing key, you can also use the square bracket syntax:

[Click here to view code image](#)

```
student_record['gpa'] = 3.0  
student_record['gpa']  
3.0
```

A handy way to increment numeric data is by using the += operator, which is a shortcut for updating a value by adding to it:

[Click here to view code image](#)

```
student_record['gpa'] += 1.0  
student_record['gpa']  
4.0
```

Removing Items from Dictionaries

Sometimes you need to remove data, such as when a dictionary includes personally identifiable information (PII). Say that your data includes a student's ID, but this ID is irrelevant to a particular study. In order to preserve the privacy of the student, you could update the value for the ID to None:

[Click here to view code image](#)

```
student_record = {'advisor': 'Pickerson',
                  'first': 'Julia',
                  'gpa': 4.0,
                  'last': 'Brown',
                  'major': 'Data Science',
                  'minor': 'Math'}
student_record['id'] = None
student_record
{'advisor': 'Pickerson',
 'first': 'Julia',
 'gpa': 4.0,
 'id': None,
 'last': 'Brown',
 'major': 'Data Science',
 'minor': 'Math'}
```

This would prevent anyone from using the ID.

Another option would be to remove the key/value pair altogether by using the `del()` function. This function takes the dictionary with the key in square brackets as an argument and removes the appropriate key/value pair:

[Click here to view code image](#)

```
del(student_record['id'])
student_record
{'advisor': 'Pickerson',
 'first': 'Julia',
 'gpa': 4.0,
 'last': 'Brown',
 'major': 'Data Science',
 'minor': 'Math'}
```

Note

Of course, to really protect the subject's identity, you would want to remove the person's name as well as any other PII.

Dictionary Views

Dictionary views are objects that offer insights into a dictionary. There are three views: `dict_keys`, `dict_values`, and `dict_items`. Each view type lets you look at the dictionary from a different perspective.

Dictionaries have a `keys()` method, which returns a `dict_keys` object. This object gives you access to the current keys of the dictionary:

[Click here to view code image](#)

```
keys = subject_1.keys()  
keys  
dict_keys(['name', 'height', 'gpa', 'ranking'])
```

The `values()` method returns a `dict_values` object, which gives you access to the values stored in the dictionary:

[Click here to view code image](#)

```
values = subject_1.values()  
values  
dict_values(['Paula', 64, 4.0, 1])
```

The `items()` method returns a `dict_items` object, which represents the key/value pairs in a dictionary:

[Click here to view code image](#)

```
items = subject_1.items()  
items  
dict_items([('name', 'Paula'), ('height', 64), ('gpa', 4.0), ('r
```

You can test membership in any of these views by using the `in` operator. This example shows how to check whether the key 'ranking' is used in this dictionary:

```
'ranking' in keys  
True
```

This example shows how to check whether the integer 1 is one of the values in the dictionary:

```
1 in values  
True
```

This example shows how to check whether the key/value pair mapping 'ranking' is 1:

```
('ranking',1) in items  
True
```

Starting in Python 3.8, dictionary views are dynamic. This means that if you change a dictionary after acquiring a view, the view reflects the new changes. For example, say that you want to delete a key/value pair from the dictionary whose views are accessed above, as shown here:

[Click here to view code image](#)

```
del(subject_1['ranking'])  
subject_1  
{'name': 'Paula', 'height': 64, 'gpa': 4.0}
```

That key/value pair is also deleted from the view objects:

[Click here to view code image](#)

```
'ranking' in keys  
False
```

```
1 in values  
False
```

```
('ranking',1) in items  
False
```

Every dictionary view type has a length, which you can access by using the same len function used with sequences:

[Click here to view code image](#)

```
len(keys)
3
```

```
len(values)
3
```

```
len(items)
3
```

As of Python 3.8, you can use the `reversed` function on a `dict_key` view to get a view in reverse order:

[Click here to view code image](#)

```
keys
dict_keys(['name', 'height', 'gpa'])

list(reversed(keys))
['gpa', 'height', 'name']
```

The `dict_key` views are set-like objects, which means that many set operations will work on them. This example shows how to create two dictionaries:

[Click here to view code image](#)

```
admission_record = {'first': 'Julia',
                    'last': 'Brown',
                    'id': 'ax012E4',
                    'admitted': '2020-03-14'}
student_record = {'first': 'Julia',
                  'last': 'Brown',
                  'id': 'ax012E4',
                  'gpa': 3.8,
                  'major': 'Data Science',
                  'minor': 'Math',
                  'advisor': 'Pickerson'}
```

Then you can test the equality of keys:

[Click here to view code image](#)

```
admission_record.keys() == student_record.keys()
False
```

You can also look for a symmetric difference:

[Click here to view code image](#)

```
admission_record.keys() ^ student_record.keys()
{'admitted', 'advisor', 'gpa', 'major', 'minor'}
```

Here is how you look for intersection:

[Click here to view code image](#)

```
admission_record.keys() & student_record.keys()
{'first', 'id', 'last'}
```

Here is how you look for difference:

[Click here to view code image](#)

```
admission_record.keys() - student_record.keys()
{'admitted'}
```

Here is how you look for union:

[Click here to view code image](#)

```
admission_record.keys() | student_record.keys()
{'admitted', 'advisor', 'first', 'gpa', 'id', 'last', 'major', 'minor'}
```

Note

You will learn more about sets and set operations in the next section.

The most common use for `key_item` views is to iterate through a dictionary and perform an operation with each key/value pair. The following example uses a `for` loop (see [Chapter 5](#), “Execution Control”) to print each pair:

[Click here to view code image](#)

```
for k,v in student_record.items():
    print(f"{k} => {v}")
first => Julia
```

```
last => Brown
gpa => 4.0
major => Data Science
minor => Math
advisor => Pickerson
```

You can do similar loops with `dict_keys` or `dict_values`, as required.

Checking to See If a Dictionary Has a Key

You can use the `dict_key` and the `in` operator to check whether a key is used in a dictionary:

[Click here to view code image](#)

```
'last' in student_record.keys()
True
```

As a shortcut, you can also test for a key without explicitly calling the `dict_key` view. Instead, you just use `in` directly with the dictionary:

```
'last' in student_record
True
```

This also works if you want to iterate through the keys of a dictionary. You don't need to access the `dict_key` view directly:

[Click here to view code image](#)

```
for key in student_record:
    print(f"key: {key}")
key: first
key: last
key: gpa
key: major
key: minor
key: advisor
```

The `get` Method

Trying to access a key that is not in a dictionary by using the square bracket syntax causes an error:

[Click here to view code image](#)

```
student_record['name']
```

```
-----  
KeyError                                Traceback (most recent call  
<ipython-input-18-962c04650d3e> in <module>()  
----> 1 student_record['name']  
      KeyError: 'name'
```

This type of error stops the execution of a program that is run outside a notebook. One way to avoid these errors is to test whether the key is in the dictionary before accessing it:

[Click here to view code image](#)

```
if 'name' in student_record:  
    student_record['name']
```

This example uses an `if` statement that accesses the key ‘name’ only if it is in the dictionary. (For more on `if` statements, see [Chapter 5](#).)

As a convenience, dictionaries have a method, `get()`, that is designed to for safely accessing missing keys. By default, this method returns a `None` constant if the key is missing:

[Click here to view code image](#)

```
print( student_record.get('name') )  
None
```

You can also provide a second argument, which is the value to return in the event of missing keys:

[Click here to view code image](#)

```
student_record.get('name', 'no-name')  
'no-name'
```

You can also chain together multiple `get` statements:

[Click here to view code image](#)

```
student_record.get('name', admission_record.get('first', 'no-name'))  
'Julia'
```

This example tries to get the value for the key 'name' from the dictionary `student_record`, and if it is missing, it tries to get the value for the key 'first' from the dictionary `admission_record`, and if that key is missing, it returns the default value 'no-name'.

Valid Key Types

You can change the values of some objects, but other objects have static values. The objects whose values can be changed are referred to as *mutable*. As you have already seen, lists are mutable objects; other objects whose value can be changed are also mutable. On the other hand, you cannot change the value of immutable objects. Immutable objects include integers, strings, range objects, binary strings, and tuples.

Immutable objects, with the exception of certain tuples, can be used as keys in dictionaries:

[Click here to view code image](#)

```
{ 1          : 'an integer',  
  'string'   : 'a string',  
  ('item',)  : 'a tuple',  
  range(12)  : 'a range',  
  b'binary'  : 'a binary string' }
```

Mutable objects, such as lists, are not valid keys for dictionaries. If you try to use a list as a key, you experience an error:

[Click here to view code image](#)

```
{('item',): 'a tuple',  
 1: 'an integer',  
 b'binary': 'a binary string',  
 range(0, 12): 'a range',  
 'string': 'a string',  
 ['a', 'list'] : 'a list key' }
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-31-1b0e555de2b5> in <module>()
----> 1 { ['a', 'list'] : 'a list key' }
TypeError: unhashable type: 'list'
```

A tuple whose contents are immutable can be used as a dictionary key. So, tuples of numbers, strings, and other tuples are all valid as keys:

[Click here to view code image](#)

```
tuple_key = (1, 'one', 1.0, ('uno',))
{ tuple_key: 'some value' }
{(1, 'one', 1.0, ('uno',)): 'some value'}
```

If a tuple contains a mutable object, such as a list, then the tuple is not a valid key:

[Click here to view code image](#)

```
bad_tuple = ([1, 2], 3)
{ bad_tuple: 'some value' }
-----
TypeError                                Traceback (most recent call last)
<ipython-input-28-b2cddfdda91e> in <module>()
      1 bad_tuple = ([1, 2], 3)
----> 2 { bad_tuple: 'some value' }
TypeError: unhashable type: 'list'
```

The `hash` Method

You can think of a dictionary as storing values in an indexed list-like structure, with a method that quickly and reliably maps the key objects to the appropriate index numbers. This method is known as a hash function and can be found on immutable Python objects as the `__hash__()` method. It is designed to be used behind the scenes but can be called directly:

[Click here to view code image](#)

```
a_string = 'a string'
a_string.__hash__()
```

```
4815474858255585337
```

```
a_tuple = 'a', 'b',  
a_tuple.__hash__()  
7273358294597481374
```

```
a_number = 13  
a_number.__hash__()  
13
```

This hash function uses the value of an object to produce a consistent output. Hence for a mutable object, no consistent hash can be produced. You cannot get a hash of a mutable object such as a list:

[Click here to view code image](#)

```
a_list = ['a', 'b']  
a_list.__hash__()  
-----  
TypeError Traceback (most recent call last) <ipython-input-40-c<  
<module>()  
      1 a_list = ['a', 'b']  
----> 2 a_list.__hash__()  
TypeError: 'NoneType' object is not callable
```

Dictionaries and lists are among the most commonly used data structures in Python. They give you great ways to structure data for meaningful, fast lookups.

Note

Although the key/value lookup mechanism does not rely on an order of the data, as of Python 3.7, the order of the keys reflects the order in which they were inserted.

Sets

The Python set data structure is an implementation of the sets you may be familiar with from mathematics. A *set* is an unordered collection of unique

items. You can think of a set as a magic bag that does not allow duplicate objects. The items in sets can be any hashable type.

A set is represented in Python as a list of comma-separated items enclosed in curly braces:

```
{ 1, 'a', 4.0 }
```

You can create a set either by using the `set()` constructor or by using curly braces directly. However, when you use empty curly braces, you create an empty dictionary, not an empty set. If you want to create an empty set, you must use the `set()` constructor:

[Click here to view code image](#)

```
empty_set = set()
empty_set
set()
```

```
empty_set = {}
empty_set
{}
```

You can create a set with initial values by using either the constructor or the curly braces.

You can provide any type of sequence as the argument, and a set will be returned based on the unique items from the sequence:

[Click here to view code image](#)

```
letters = 'a', 'a', 'a', 'b', 'c'
unique_letters = set(letters)
unique_letters
{'a', 'b', 'c'}
```

```
unique_chars = set('mississippi')
unique_chars
{'i', 'm', 'p', 's'}
```

```
unique_num = {1, 1, 2, 3, 4, 5, 5}
unique_num
{1, 2, 3, 4, 5}
```

Much like dictionary keys, sets hash their contents to determine uniqueness. Therefore, the contents of a set must be hashable and, hence, immutable. A list cannot be a member of a set:

[Click here to view code image](#)

```
bad_set = { ['a','b'], 'c' }
```

```
-----  
TypeError                                Traceback (most recent  
    <ipython-input-12-1179bc4af8b8> in <module>()  
----> 1 bad_set = { ['a','b'], 'c' }  
TypeError: unhashable type: 'list'
```

You can add items to a set by using the `add()` method:

[Click here to view code image](#)

```
unique_num.add(6)  
unique_num  
{1, 2, 3, 4, 5, 6}
```

You can use the `in` operator to test membership in a set:

[Click here to view code image](#)

```
3 in unique_num  
True  
  
3 not in unique_num  
False
```

You can use the `len()` function to see how many items a set contains:

```
len(unique_num)  
6
```

As with lists, you can remove and return an item from a set by using the `pop()` method:

[Click here to view code image](#)

```
unique_num.pop()  
unique_num  
{2, 3, 4, 5, 6}
```

Unlike with lists, you cannot rely on `pop()` to remove a set's items in any particular order. If you want to remove a particular item from a set, you can use the `remove()` method:


[Click here to view code image](#)

```
students = {'Karl', 'Max', 'Tik'}
students.remove('Karl')
students
{'Max', 'Tik'}
```

This method does not return the item removed. If you try to remove an item that is not found in the set, you get an error:

[Click here to view code image](#)

```
students.remove('Barb')
-----
KeyError                                Traceback (most recent
    <ipython-input-3-a36a5744ac05> in <module>()
----> 1 students.remove('Barb')
KeyError: 'Barb'
```



You could write code to test whether an item is in a set before removing it, but there is a convenience function, `discard()`, that does not throw an error when you attempt to remove a missing item:

[Click here to view code image](#)

```
students.discard('Barb')
students.discard('Tik')
students
{'Max'}
```

You can remove all of the contents of a set by using the `clear()` method:

[Click here to view code image](#)

```
students.clear()
students
set()
```

Remember that because sets are unordered, they do not support indexing:

[Click here to view code image](#)

```
unique_num[3]
```

```
-----  
TypeError                                Traceback (most recent  
<ipython-input-16-fecab0cd5f95> in <module>()  
----> 1 unique_num[3]  
TypeError: 'set' object does not support indexing
```

You can test equality by using the equals, `=`, and not equals, `!=`, operators (which are discussed in [Chapter 5](#)). Because sets are unordered, sets created from sequences with the same items in different orders are equal:

[Click here to view code image](#)

```
first = {'a', 'b', 'c', 'd'}  
second = {'d', 'c', 'b', 'a'}  
first == second  
True
```

```
first != second  
False
```

Set Operations

You can perform a number of operations with sets. Many set operations are offered both as methods on the set objects and as separate operators (`<`, `<=`, `>`, `>=`, `&`, `|`, and `^`). The set methods can be used to perform operations between sets and other sets, and they can also be used between sets and other iterables (that is, data types that can be iterated over). The set operators work only between sets and other sets (or frozensets).

Disjoint

Two sets are disjoint if they have no items in common. With Python sets, you can use the `disjoint()` method to test this. If you test a set of even numbers against a set of odd numbers, they share no numbers, and hence the result of `disjoint()` is `True`:

[Click here to view code image](#)

```
even = set(range(0,10,2))
even
{0, 2, 4, 6, 8}

odd = set(range(1,11,2))
odd
{1, 3, 5, 7, 9}

even.isdisjoint(odd)
True
```

Subset

If all the items in a set, Set B, can be found in another set, Set A, then Set B is a subset of Set A. The `subset()` method tests whether the current set is a subset of another. The following example tests whether a set of positive multiples of 3 below 21 are a subset of positive integers below 21:

[Click here to view code image](#)

```
nums = set(range(21))
nums
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}

threes = set(range(3,21,3))
threes
{3, 6, 9, 12, 15, 18}

threes.issubset(nums)
True
```



You can use the `<=` operator to test whether a set to the left is a subset of a set to the right:

```
threes <= nums
True
```

As mentioned earlier in this chapter, the method version of this operator works with non-set arguments. The following example tests whether a set of multiples of 3 are in the range 0 through 20:

[Click here to view code image](#)

```
threes.issubset(range(21))  
True
```

The operator does not work with a non-set object:

[Click here to view code image](#)

```
threes <= range(21)  
-----  
TypeError                                Traceback (most recent  
    <ipython-input-30-dbd51effe302> in <module>()  
    ----> 1 threes <= range(21)  
TypeError: '<=' not supported between instances of 'set' and 'range'
```

Proper Subsets

If all the items of a set are contained in a second set, but not all the items in the second set are in the first set, then the first set is a proper subset of the second set. This is equivalent to saying that the first set is a subset of the second and that they are not equal. You use the < operator to test for proper subsets:

[Click here to view code image](#)

```
threes < nums  
True  
  
threes < {'3', '6', '9', '12', '15', '18'}  
False
```

Supersets and Proper Supersets

A superset is the reverse of a subset: If a set contains all the elements of another set, it is a superset of the second set. Similarly, if a set is a superset of another set and they are not equal, then it is a proper superset. Python sets have an issuperset() method, which takes another set or any other iterable as an argument:

[Click here to view code image](#)

```
nums.issuperset(threes)
```

```
True
```

```
nums.issuperset([1,2,3,4])
```

```
True
```

You use the greater-than-or-equal-to operator, `>=`, to test for supersets and the greater-than operator, `>`, to test for proper supersets:

[Click here to view code image](#)

```
nums >= threes
```

```
True
```

```
nums > threes
```

```
True
```

```
nums >= nums
```

```
True
```

```
nums > nums
```

```
False
```

Union

The union of two sets results in a set containing all the items in both sets. For Python sets you can use the `union()` method, which works with sets and other iterables, and the standalone bar operator, `|`, which returns the union of two sets:

[Click here to view code image](#)

```
odds = set(range(0,12,2))
```

```
odds
```

```
{0, 2, 4, 6, 8, 10}
```

```
evens = set(range(1,13,2))
```

```
evens
```

```
{1, 3, 5, 7, 9, 11}
```

```
odds.union(evens)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

```
odds.union(range(0,12))
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

odds | evens

`{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}`

Intersection

The intersection of two sets is a set containing all items shared by both sets. You can use the `intersection()` method or the and operator, `&`, to perform intersections:

[Click here to view code image](#)

```
under_ten = set(range(10))
odds = set(range(1,21,2))
under_ten.intersection(odds)
{1, 3, 5, 7, 9}
```

```
under_ten & odds
{1, 3, 5, 7, 9}
```

Difference

The difference between two sets is all of the items in the first set that are not in the second set. You can use the `difference()` method or the minus operator, `-`, to perform set difference:

[Click here to view code image](#)

```
odds.difference(under_ten)
{11, 13, 15, 17, 19}
```

```
odds - under_ten
{11, 13, 15, 17, 19}
```

Symmetric Difference

The symmetric difference of two sets is a set containing any items contained in only one of the original sets. Python sets have a `symmetric_difference()` method, and the caret operator, `^`, for calculating the symmetric difference:

[Click here to view code image](#)

```
under_ten = set(range(10))
over_five = set(range(5, 15))
under_ten.symmetric_difference(over_five)
{0, 1, 2, 3, 4, 10, 11, 12, 13, 14}

under_ten ^ over_five
{0, 1, 2, 3, 4, 10, 11, 12, 13, 14}
```

Updating Sets

Python sets offer a number of ways to update the contents of a set in place. In addition to using `update()`, which adds the contents to a set, you can use variations that update based on the various set operations.

The following example shows how to update from another set:

[Click here to view code image](#)

```
unique_num = {0, 1, 2}
unique_num.update( {3, 4, 5, 7} )
unique_num
{0, 1, 2, 3, 4, 5, 7}
```

The following example shows how to update from a list:

[Click here to view code image](#)

```
unique_num.update( [8, 9, 10] )
unique_num
{0, 1, 2, 3, 4, 5, 7, 8, 9, 10}
```

The following example shows how to update the difference from a range:

[Click here to view code image](#)

```
unique_num.difference_update( range(0,12,2) )
unique_num
{1, 3, 5, 7, 9}
```

The following example shows how to update the intersection:

[Click here to view code image](#)

```
unique_num.intersection_update( { 2, 3, 4, 5 } )  
unique_num  
{3, 5}
```

The following example shows how to update the symmetric difference:

[Click here to view code image](#)

```
unique_num.symmetric_difference_update( {5, 6, 7 } )  
unique_num  
{3, 6, 7}
```

The following example shows how to update the union operator:

[Click here to view code image](#)

```
unique_letters = set("mississippi")  
unique_letters  
{ 'i', 'm', 'p', 's' }  
  
unique_letters |= set("Arkansas")  
unique_letters  
{ 'A', 'a', 'i', 'k', 'm', 'n', 'p', 'r', 's' }
```

The following example shows how to update the difference operator:

[Click here to view code image](#)

```
unique_letters -= set('Arkansas')  
unique_letters  
{ 'i', 'm', 'p' }
```

The following example shows how to update the intersection operator:

[Click here to view code image](#)

```
unique_letters &= set('permanent')  
unique_letters  
{ 'm', 'p' }  
  
unique_letters ^= set('mud') 2 unique_letters  
{ 'd', 'p', 'u' }
```

Frozensets

Because sets are mutable, they cannot be used as dictionary keys or even as items in sets. In Python, frozensets are set-like objects that are immutable. You can use frozensets in place of sets for any operation that does not change its contents, as in these examples:

[Click here to view code image](#)

```
froze = frozenset(range(10))  
froze  
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

```
froze < set(range(21))  
True
```

```
froze & set(range(5, 15))  
frozenset({5, 6, 7, 8, 9})
```

```
froze ^ set(range(5, 15))  
frozenset({0, 1, 2, 3, 4, 10, 11, 12, 13, 14})
```

```
froze | set(range(5,15))  
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14})
```

Summary

Python's built-in data structures offer a variety of ways to represent and organize your data. Dictionaries and sets are both complements to the sequence types. Dictionaries map keys to values in an efficient way. Sets implement mathematical set operations as data structures. Both dictionaries and sets are great choices where order is not the best operating principle.

Questions

1. What are three ways to create a dictionary with the following key/value pairs:

```
{'name': 'Smuah', 'height':62}
```
2. How would you update the value associated with the key gpa in the dictionary student to be '4.0'?

3. Given the dictionary `data`, how would you safely access the value for the key `settings` if that key might be missing?
4. What is the difference between a mutable object and immutable object?
5. How would you create a set from the string `"lost and lost again"`?

5

Execution Control

An approximate answer to the right problem is worth a good deal more than an exact answer to an approximate problem.

John Tukey

In This Chapter

- [Introduction to compound statements](#)
- [Equality operations](#)
- [Comparison operations](#)
- [Boolean operations](#)
- [if statements](#)
- [while loops](#)
- [for loops](#)

Up until this point in the book, you've seen statements as individual units, executing sequentially one line at a time. Programming becomes much more powerful and interesting when you can group statements together so that they execute as a unit. Simple statements that are joined together can perform more complex behaviors.

Compound Statements

[Chapter 2](#), “Fundamentals of Python,” introduces simple statements, each of which performs an action. This chapter looks at compound statements, which allow you to control the execution of a group of statements. This execution can occur only when a condition is true. The compound statements covered in this chapter include `for` loops, `while` loops, `if` statements, `try` statements, and `with` statements.

Compound Statement Structure

A compound statement consists of a controlling statement or statements and a group of statements whose execution is controlled. A control statement starts with a keyword indicating the type of compound statement, an expression specific to the type of statement, and then a colon:

```
<keyword> <expression>:
```

The controlled statements can be grouped in one of two ways. The first, more common, way is to group them as a *code block*, which is a group of statements that are run together. In Python, code blocks are defined using indentation. A group of statements that share the same indentation are grouped into the same code block. The group ends when there is a statement that is not indented as far as the others. That final statement is not part of the code block and will execute regardless of the control statement. This is what a code block looks like:

[Click here to view code image](#)

```
<control statement>:
    <controlled statement 1>
    <controlled statement 2>
    <controlled statement 3>
< statement ending block>
```

Using indentation to define code blocks is one of the features that differentiates Python from most other popular languages, which use other mechanisms, such as curly brackets, to group code.

Another way to group controlled statements is to list them directly following the control statement and separate the controlled statements with

semicolons:

[Click here to view code image](#)

```
<control statement>:<controlled statement 1>;<controlled statement 2>
```

You should use this second style only when you have very few controlled statements and you feel that limiting the compound statement to one line will enhance, not detract from, the readability of the program.

Evaluating to `True` or `False`

`if` statements, `while` loops, and `for` loops are all compound statements that rely on a controlling expression that must evaluate to `True` or `False`.

Luckily, in Python, pretty much everything evaluates as equal to one of these. The four most commonly used built-in expressions used as controls for compound statements are equality operations, comparison operations, Boolean operations, and object evaluation.

Equality Operations

Python offers the equality operator, `==`, the inequality operator, `!=`, and the identity operator, `is`. The equality and inequality operators both compare the value of two objects and return one of the constants `True` or `False`.

[Listing 5.1](#) assigns two variables with integer values of 1, and another with the value 2. It then uses the equality operator to show that the first two variables are equal, and the third is not. It does the same with the inequality operator, whose results are opposite those of the equality operator with the same inputs.

Listing 5.1 Equality Operations

[Click here to view code image](#)

```
# Assign values to variables
a, b, c = 1, 1, 2
# Check if value is equal
a == b
True
```

```
a == c
False
```

```
a != b
False
```

```
a != c
True
```

You can compare different types of objects by using the equality/inequality operators. For numeric types, such as floats and integers, the values are compared. For example, if you compare the integer 1 to the float 1.0, they evaluate as equal:

```
1 == 1.0
True
```

Most other cross-type comparisons return `False`, regardless of value. Comparing a string to an integer will always return `False`, regardless of the values:

```
'1' == 1
False
```

Web forms often report all user input as strings. A common problem occurs when trying to compare user input from a web form that represents a number but is of type string with an actual number. String input always evaluates to `False` when compared to a number, even if the input is a string version of the same value.

Comparison Operations

You use comparison operators to compare the order of objects. What “the order” means depends on the type of objects compared. For numbers, the comparison is the order on a number line, and for strings, the Unicode value of the characters is used. The comparison operators are less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=).

[Listing 5.2](#) demonstrates the behavior of various comparison operators.

Listing 5.2 Comparison Operations

[Click here to view code image](#)

```
a, b, c = 1, 1, 2
```

```
a < b
```

```
False
```

```
a < c
```

```
True
```

```
a <= b
```

```
True
```

```
a > b
```

```
False
```

```
a >= b
```

```
True
```

There are certain cases where you can use comparison operators between objects of different types, such as with the numeric types, but most cross-type comparisons are not allowed. If you use a comparison operator with noncomparable types, such as a string and a list, an error occurs.

Boolean Operations

The Boolean operators are based on Boolean math, which you may have studied in a math or philosophy course. These operations were first formalized by the mathematician George Boole in the 19th century. In Python, the Boolean operators are `and`, `or`, and `not`. The `and` and `or` operators each take two arguments; the `not` operator takes only one.

The `and` operator evaluates to `True` if both of its arguments evaluate to `True`; otherwise, it evaluates to `False`. The `or` operator evaluates to `True` if either of its arguments evaluates to `True`; otherwise, it evaluates to `False`. The `not` operator returns `True` if its argument evaluates to `False`; otherwise, it evaluates to `False`. [Listing 5.3](#) demonstrates these behaviors.

Listing 5.3 Boolean Operations

[Click here to view code image](#)

True and True

True

True and False

False

True or False

True

False or False

False

not False

True

not True

False

Both the `and` and `or` operators are short-circuit operators. This means they will only evaluate their input expression as much as is needed to determine the output. For example, say that you have two methods, `returns_false()` and `returns_true()`, and you use them as inputs to the `and` operator as follows:

[Click here to view code image](#)

`returns_false()` and `returns_true()`

If `returns_false()` returns `False`, `returns_true()` will not be called, as the result of the `and` operation is already determined. Similarly, say that you use them as arguments to the `or` operation, like this:

[Click here to view code image](#)

`returns_true()` or `returns_false()`

In this case, the second method will not be called if the first returns `True`.

The `not` operator always returns one of the Boolean constants `True` or `False`. The other two Boolean operators return the result of the last expression evaluated. This is very useful with object evaluation.

Object Evaluation

All objects in Python evaluate to `True` or `False`. This means you can use objects as arguments to Boolean operations. The objects that evaluate to `False` are the constants `None` and `False`, any numeric with a value of zero, or anything with a length of zero. This includes empty sequences, such as an empty string (`""`) or an empty list (`[]`). Almost anything else evaluates to `True`.

Because the `or` operator returns the last expression it evaluates, you can use it to create a default value when a variable evaluates to `False`:

[Click here to view code image](#)

```
a = ''  
b = a or 'default value'  
b  
'default value'
```

Because this example assigns the first variable to an empty string, which has a length of zero, this variable evaluates to `False`. The `or` operator evaluates this and then evaluates and returns the second expression.

`if` Statements

The `if` statement is a compound statement. `if` statements let you branch the behavior of your code depending on the current state. You can use an `if` statement to take an action only when a chosen condition is met or use a more complex one to choose among multiple actions, depending on multiple conditions. The control statement starts with the keyword `if` followed by an expression (which evaluates to `True` or `False`) and then a colon. The controlled statements follow either on the same line separated by semicolons:

[Click here to view code image](#)

```
if True:message="It's True!";print(message)  
It's True!
```

or as an indented block of code, separated by newlines:

[Click here to view code image](#)

```
if True:
    message="It's True"
    print(message)
It's True
```

In both of these examples, the controlling expression is simply the reserved constant `True`, which always evaluates to `True`. There are two controlled statements: The first assigns a string to the variable `message`, and the second prints the value of this variable. It's usually more readable to use the block syntax, as in the second example.

If the controlling expression evaluates to `False`, the program continues executing and skips the controlled statement(s):

[Click here to view code image](#)

```
if False:
    message="It's True"
    print(message)
```

The Walrus Operator

When you assign a value to a variable, Python does not return a value. A common situation is to make a variable assignment and then check the value of the variable. For example, you might assign to a variable the value returned by a function, and if that value is not `None`, you may use the returned object. The `search` method of the Python `re` module (covered in [Chapter 15](#), “Other Topics”) returns a `match` object if it finds a match in a string, and it returns `None` otherwise, so if you want to use the `match` object, you need to make sure it's not `None` first:

[Click here to view code image](#)

```
import re
s = '2020-12-14'
match = re.search(r'(\d\d\d\d)-(\d\d)-(\d\d)', s)
if match:
    print(f"Matched items: {match.groups(1)}")
```

```
else:
    print(f"No match found in {s}")
```

Python 3.8 introduced a new operator, the assignment operator (`:=`). It is referred to as the *walrus operator* due to its resemblance to a walrus's head. This operator assigns a value to a variable and returns that value. You could rewrite the match example by using it:

[Click here to view code image](#)

```
import re
s = '2020-12-14'
if match := re.search(r'(\d\d\d\d)-(\d\d)-(\d\d)', s):
    print(f"Matched items: {match.groups(1)}")
else:
    print(f"No match found in {s}")
```

This operator creates less complicated, more readable code.

Here is an example that uses a membership test as the controlling expression:

[Click here to view code image](#)

```
snack = 'apple'
fruit = {'orange', 'apple', 'pear'}
if snack in fruit:
    print(f"Yeah, {snack} is good!")
Yeah, apple is good!
```

This example checks whether the value of the variable `snack` is in the set `fruit`. If it is, an encouraging message is printed.

If you want to run an alternative block of code when the controlling expression is `False`, you can use an `else` statement. An `else` statement consists of the keyword `else` followed by a colon and then a block of code that will execute only if the controlling expression preceding it evaluates to `False`. This lets you branch the logic in your code. Think of it as choosing which actions to take based on the current state. [Listing 5.4](#) shows an `else` statement added to the snack-related `if` statement. The second print

statement executes only if the controlling expression `snack in fruit` is `False`.

Listing 5.4 `else` Statements

[Click here to view code image](#)

```
snack = 'cake'
fruit = {'orange', 'apple', 'pear'}
if snack in fruit:
    print(f"Yeah, {snack} is good!")
else:
    print(f"{snack}!? You should have some fruit")
cake!? You should have some fruit
```

If you want to have multiple branches in your code, you can nest `if` and `else` statements as shown in [Listing 5.5](#). In this case, three choices are made: one if the balance is positive, one if it is negative, and one if it is negative.

Listing 5.5 Nested `else` Statements

[Click here to view code image](#)

```
balance = 2000.32
account_status = None

if balance > 0:
    account_status = 'Positive'
else:
    if balance == 0:
        account_status = 'Empty'
    else:
        account_status = 'Overdrawn'

print(account_status)
Positive
```

While this code is legitimate and will work the way it is supposed to, it is a little hard to read. To perform the same branching logic in a more concise way, you can use an `elif` statement. This type of statement is added after an initial `if` statement. It has a controlling expression of its own, which will be evaluated only if the previous statement's expression evaluates to `False`.

[Listing 5.6](#) performs the same logic as [Listing 5.5](#), but has the nested `else` and `if` statements replaced by `elif`.

Listing 5.6 `elif` Statements

[Click here to view code image](#)

```
balance = 2000.32
account_status = None

if balance > 0:
    account_status = 'Positive'
elif balance == 0:
    account_status = 'Empty'
else:
    account_status = 'Overdrawn'

print(account_status)
Positive
```

By chaining multiple `elif` statements with an `if` statement, as demonstrated in [Listing 5.7](#), you can perform complicated choices. Usually an `else` statement is added at the end to catch the case that all the controlling expressions are `False`.

Listing 5.7 Chaining `elif` Statements

[Click here to view code image](#)

```
fav_num = 13

if fav_num in (3,7):
    print(f"{fav_num} is lucky")
elif fav_num == 0:
    print(f"{fav_num} is evocative")
elif fav_num > 20:
    print(f"{fav_num} is large")
elif fav_num == 13:
    print(f"{fav_num} is my favorite number too")
else:
    print(f"I have no opinion about {fav_num}")
is my favorite number too
```

while Loops

A `while` loop consists of the keyword `while` followed by a controlling expression, a colon, and then a controlled code block. The controlled statement in a `while` loop executes only if the controlling statement evaluates to `True`; in this way, it is like an `if` statement. Unlike an `if` statement, however, the `while` loop repeatedly continues to execute the controlled block as long as its control statement remains `True`. Here is a `while` loop that executes as long as the variable `counter` is below five:

[Click here to view code image](#)

```
counter = 0
while counter < 5:
    print(f'I've counted {counter} so far, I hope there aren't more')
    counter += 1
```



Notice that the variable is incremented with each iteration. This guarantees that the loop will exit. Here is the output from running this loop:

[Click here to view code image](#)

```
I've counted 0 so far, I hope there aren't more
I've counted 1 so far, I hope there aren't more
I've counted 2 so far, I hope there aren't more
I've counted 3 so far, I hope there aren't more
I've counted 4 so far, I hope there aren't more
```

You can see that the loop runs five times, incrementing the variable each time.

Note

It is important to provide an exit condition, or your loop will repeat infinitely.

for Loops

for loops are used to iterate through some group of objects. This group can be a sequence, a generator, a function, or any other object that is iterable. An iterable object is any object that returns a series of items one at a time. for loops are commonly used to perform a block of code a set number of times or perform an action on each member of a sequence. The controlling statement of a for loop consists of the keyword `for`, a variable, the keyword `in`, and the iterable followed by a colon:

```
for <variable> in <iterable>:
```

The variable is assigned the first value from the iterable, the controlled block is executed with that value, and then the variable is assigned the next value. This continues as long as the iterable has values to return.

A common way to run a block of code a set number of times is to use a for loop with a range object as the iterable:

[Click here to view code image](#)

```
for i in range(6):
    j = i + 1
    print(j)
1
2
3
4
5
6
```

This example assigns the values 0, 1, 2, 3, 4, and 5 to the variable `i`, running a code block for each one.

Here is an example of using a list as the iterable:

[Click here to view code image](#)

```
colors = ["Green", "Red", "Blue"]
for color in colors:
    print(f"My favorite color is {color}")
    print("No, wait...")
My favorite color is Green
No, wait...
My favorite color is Red
```

```
No, wait...
My favorite color is Blue
No, wait...
```

Each item in the list is used in the code block, and when there are no items left, the loop exits.

`break` and `continue` Statements

The `break` statement gives you an early exit from a `while` or `for` loop. When the statement is evaluated, the current block ceases to execute, and the loop is ended. This is usually used in conjunction with a nested `if` statement.

[Listing 5.8](#) shows a loop whose controlling expression is always `True`. A nested `if` statement calls `break` when its condition is met, ending the loop at that point.

Listing 5.8 `break` Statement

[Click here to view code image](#)

```
fish = ['mackerel', 'salmon', 'pike']
beasts = ['salmon', 'pike', 'bear', 'mackerel']
i = 0

while True:
    beast = beasts[i]
    if beast not in fish:
        print(f"Oh no! It's not a fish, it's a {beast}")
        break
    print(f"I caught a {beast} with my fishing net")
    i += 1
I caught a salmon with my fishing net
I caught a pike with my fishing net
Oh no! It's not a fish, it's a bear
```

The `continue` statement skips a single iteration of a loop when it is invoked. It is also usually used in conjunction with a nested `if` statement. [Listing 5.9](#) demonstrates the use of a `continue` statement to skip printing names that don't begin with the letter *b*.

Listing 5.9 `continue` Statement

[Click here to view code image](#)

```
for name in ['bob', 'billy', 'bonzo', 'fred', 'baxter']:
    if not name.startswith('b'):
        continue
    print(f"Fine fellow that {name}")
```

Fine fellow that bob
Fine fellow that billy
Fine fellow that bonzo
Fine fellow that baxter

Summary

Compound statements such as `if` statements, `while` loops, and `for` loops are a fundamental part of code beyond simple scripts. With the ability to branch and repeat your code, you can form blocks of action that describe complex behavior. You now have tools to structure more complex software.

Questions

1. What is printed by the following code if the variable `a` is set to an empty list?

[Click here to view code image](#)

```
if a:
    print(f"Hiya {a}")
else:
    print(f"Biya {a}")
```

2. What is printed by the previous code if the variable `a` is set to the string `"Henry"`?
3. Write a `for` loop that prints the numbers from 0 to 9, skipping 3, 5, and 7.

6

Functions

In our lust for measurement, we frequently measure that which we can rather than that which we wish to measure...and forget that there is a difference.

George Udny Yule

In This Chapter

- [Defining functions](#)
- [Docstrings](#)
- [Positional and keyword parameters](#)
- [Wildcard parameters](#)
- [Return statements](#)
- [Scope](#)
- [Decorators](#)
- [Anonymous functions](#)

The last and perhaps most powerful compound statement that we discuss is the function. Functions give you a way to name a code block wrapped as an object. That code can then be invoked by use of that name, allowing the same code to be called multiple times and in multiple places.

Defining Functions

A function definition defines a function object, which wraps the executable block. The definition does not run the code block but just defines the function. The definition describes how the function can be called, what it is named, what parameters can be passed to it, and what will be executed when it is invoked. The building blocks of a function are the controlling statement, an optional docstring, the controlled code block, and a return statement.

Control Statement

The first line of a function definition is the control statement, which takes the following form:

[Click here to view code image](#)

```
def <Function Name> (<Parameters>):
```

The `def` keyword indicates a function definition, `<Function Name>` is where the name that will be used to call the function is defined, and `<Parameters>` is where any arguments that can be passed to the function are defined. For example, the following function is defined with the name `do_nothing` and a single parameter named `not_used`:

```
def do_nothing(not_used):  
    pass
```

The code block in this case consists of a single `pass` statement, which does nothing. The Python style guide, PEP8, has conventions for naming functions (see <https://www.python.org/dev/peps/pep-0008/#function-and-variable-names>).

Docstrings

The next part of a function definition is the documentation string, or *docstring*, which contains documentation for the function. It can be omitted, and the Python compiler will not object. However, it is highly recommended to supply a docstring for all but the most obvious methods.

The docstring communicates your intentions in writing a function, what the function does, and how it should be called. PEP8 provides guidance regarding the content of docstrings (see <https://www.python.org/dev/peps/pep-0008/#documentation-strings>). The docstring consists of a single-line string or a multiline string surrounded in three pairs of double quotes that immediately follows the control statement:

[Click here to view code image](#)

```
def do_nothing(not_used):  
    """This function does nothing."""  
    pass
```

For a single-line docstring, the quotes are on the same line as the text. For a multiline docstring, the quotes are generally above and below the text, as in [Listing 6.1](#).

Listing 6.1 Multiline Docstring

[Click here to view code image](#)

```
def do_nothing(not_used):  
    """  
    This function does nothing.  
    This function uses a pass statement to  
    avoid doing anything.  
    Parameters:  
        not_used - a parameter of any type,  
                   which is not used.  
    """  
    pass
```

The first line of the docstring should be a statement summarizing what the function does. With a more detailed explanation, a blank line is left after the first statement. There are many different possible conventions for what is contained after the first line of a docstring, but generally you want to offer an explanation of what the function does, what parameters it takes, and what it is expected to return. The docstring is useful both for someone reading your code and for various utilities that read and display either the first line or the whole docstring. For example, if you call the `help()`

function on the function `do_nothing()`, the docstring is displayed as shown in [Listing 6.2](#).

Listing 6.2 Docstring from `help`

[Click here to view code image](#)

```
help(do_nothing)
```

```
Help on function do_nothing in module __main__:  
do_nothing(not_used)
```

```
This function does nothing.
```

```
This function uses a pass statement to avoid doing anything.
```

```
Parameters:
```

```
    not_used - a parameter of any type,  
               which is not used.
```

Parameters

Parameters allow you to pass values into a function, which can be used in the function's code block. A parameter is like a variable given to a function when it is called, where the parameter can be different every time you call the function. A function does not have to accept any parameters. For a function that should not accept parameters, you leave the parentheses after the function name empty:

[Click here to view code image](#)

```
def no_params():  
    print("I don't listen to nobody")
```

When you call a function, you pass the values for the parameters within the parentheses following the function name. Parameter values can be set based on the position at which they are passed or based on keywords. Functions can be defined to require their parameters be passed in either or a combination of these ways. The values passed to a function are attached to variables with the names defined in the function definition. [Listing 6.3](#) defines three parameters: `first`, `second`, and `third`. These variables are then

available to the code block that follows, which prints out the values for each parameter.

Listing 6.3 Parameters by Position or Keyword

[Click here to view code image](#)

```
def does_order(first, second, third):
    '''Prints parameters.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')

does_order(1, 2, 3)
First: 1
Second: 2
Third: 3

does_order(first=1, second=2, third=3)
First: 1
Second: 2
Third: 3

does_order(1, third=3, second=2)
First: 1
Second: 2
Third: 3
```

[Listing 6.3](#) defines the function `does_order()` and then calls it three times. The first time, it uses the position of the arguments, (1, 2, 3), to assign the variable values. It assigns the first value to the first parameter, `first`, the second value to the second parameter, `second`, and the third value to the third parameter, `third`.

The second time the listing calls the function `does_order()`, it uses keyword assignment, explicitly assigning the values using the parameter names, (`first=1, second=2, third=3`). In the third call, the first parameter is assigned by position, and the other two are assigned using keyword assignment. Notice that in all three cases, the parameters are assigned the same values.

Keyword assignments do not rely on the position of the keywords. For example, you can assign `third=3` in the position before `second=2` without

issue. You cannot use a keyword assignment to the left of a positional assignment, however:

[Click here to view code image](#)

```
does_order(second=2, 1, 3)
```

```
File "<ipython-input-9-eed80203e699>", line 1
```

```
    does_order(second=2, 1, 3)
                        ^
```

SyntaxError: positional argument follows keyword argument

You can require that a parameter be called only using the keyword method by putting a `*` to its left in the function definition. All parameters to the right of the star can only be called using keywords. [Listing 6.4](#) shows how to make the parameter `third` a required keyword parameter and then call it using the keyword syntax.

Listing 6.4 Parameters Requiring Keywords

[Click here to view code image](#)

```
def does_keyword(first, second, *, third):
```

```
    '''Prints parameters.'''
```

```
    print(f'First: {first}')
```

```
    print(f'Second: {second}')
```

```
    print(f'Third: {third}')
```

```
does_keyword(1, 2, third=3)
```

```
First: 1
```

```
Second: 2
```

```
Third: 3
```

If you try to call a required keyword parameter using positional syntax, you get an error:

[Click here to view code image](#)

```
does_keyword(1, 2, 3)
```

```
-----
      TypeError Traceback (most recent call last)
```

```
<ipython-input-15-88b97f8a6c32> in <module>
```

```
----> 1 does_keyword(1, 2, 3)
```

```
TypeError: does_keyword() takes 2 positional arguments but 3 wer
```

You can make a parameter optional by assigning to it a default value in the function definition. This value will be used if no value is provided for the parameter during the function call. [Listing 6.5](#) defines a function, `does_defaults()`, whose third parameter has the default value 3. The function is then called twice: first using positional assignment for all three parameters and then using the default value for the third.

Listing 6.5 Parameters with Defaults

[Click here to view code image](#)

```
def does_defaults(first, second, third=3):
    '''Prints parameters.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')
```

```
does_defaults(1, 2, 3)
```

```
First: 1
```

```
Second: 2
```

```
Third: 3
```

```
does_defaults(1, 2)
```

```
First: 1
```

```
Second: 2
```

```
Third: 3
```

Much as with the restriction regarding the order of keyword and position arguments during a function call, you cannot define a function with a default value parameter to the left of a non-default value parameter:

[Click here to view code image](#)

```
def does_defaults(first=1, second, third=3):
    '''Prints parameters.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')
```

```
File "<ipython-input-19-a015eaeb01be>", line 1
```

```
    def does_defaults(first=1, second, third=3):
```

```
        ^
```

```
SyntaxError: non-default argument follows default argument
```

Default values are defined in the function definition, not in the function call. This means that if you use a mutable object, such as a list or dictionary, as a default value, it will be created once for the function. Every time you call that function using that default, the same list or dictionary object will be used. This can lead to subtle problems if it is not expected. [Listing 6.6](#) defines a function with a list as the default argument. The code block appends 1 to the list. Notice that every time the function is called, the list retains the values from previous calls.

Listing 6.6 **Mutable Defaults**

[Click here to view code image](#)

```
def does_list_default(my_list=[]):  
    '''Uses list as default.'''  
    my_list.append(1)  
    print(my_list)
```

```
does_list_default()  
[1]
```

```
does_list_default()  
[1, 1]
```

```
does_list_default()  
[1, 1, 1]
```

Generally, it's a good practice to avoid using mutable objects as default parameters to avoid difficult-to-trace bugs and confusion. [Listing 6.7](#) demonstrates a common pattern to handle default values for mutable parameter types. The default value in the function definition is set to None. The code block tests whether the parameter has an assigned value. If it does not, a new list is created and assigned to the variable. Because the list is created in the code block, a new list is created every time the function is called without a value supplied for the parameter.

Listing 6.7 **Default Pattern in a Code Block**

[Click here to view code image](#)

```
def does_list_param(my_list=None):  
    '''Assigns default in code to avoid confusion.'''
```

```
my_list = my_list or []
my_list.append(1)
print(my_list)
```

```
does_list_param()
[1]
```

```
does_list_param()
[1]
```

```
does_list_param()
[1]
```

As of Python 3.8, you can restrict parameters to positional assignment only. A parameter to the left of a forward slash (/) in a function definition is restricted to positional assignment. [Listing 6.8](#) defines the function `does_positional` so that its first parameter, `first`, is positional only.

Listing 6.8 Positional-Only Parameters (Python 3.8 and Later)

[Click here to view code image](#)

```
def does_positional(first, /, second, third):
    '''Demonstrates a positional parameter.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')
```

```
does_positional(1, 2, 3)
First: 1
Second: 2
Third: 3
```

If you try to call `does_positional` by using keyword assignment for `first`, you get an error:

[Click here to view code image](#)

```
does_positional(first=1, second=2, third=3)
```

```
-----
TypeError Traceback (most recent call last)
<ipython-input-24-7b1f45f64358> in <module>
----> 1 does_positional(first=1, second=2, third=3)
TypeError: does_positional() got some positional-only arguments
keyword arguments: 'first'
```

[Listing 6.9](#) modifies `does_positional` to use positional-only and keyword-only parameters. The parameter `first` is positional only, the parameter `second` can be set using positional or keyword assignment, and the last, `third`, is keyword only.

Listing 6.9 Positional-Only and Keyword-Only Parameters

[Click here to view code image](#)

```
def does_positional(first, /, second, *, third):
    '''Demonstrates a positional and keyword parameters.'''
    print(f'First: {first}')
    print(f'Second: {second}')
    print(f'Third: {third}')

does_positional(1, 2, third=3)
First: 1
Second: 2
Third: 3
```

You can use wildcards in function definitions to accept an undefined number of positional or keyword arguments. This is often done when a function calls a function from an outside API. The function can pass the arguments through without requiring that all of the outside API's parameters be defined.

To use a wildcard for positional parameters, you use the `*` character. [Listing 6.10](#) demonstrates the definition of a function with the positional wildcard parameter `*args`. The code block receives any positional arguments given in a function call as items in a list named `args`. This function goes through the list and prints each item. The function is then called with the arguments 'Donkey', 3, and ['a'], each of which is accessed from the list and printed.

Listing 6.10 Positional Wildcard Parameters

[Click here to view code image](#)

```
def does_wildcard_positions(*args):
    '''Demonstrates wildcard for positional parameters.'''
    for item in args:
        print(item)
```

```
does_wildcard_positions('Donkey', 3, ['a'])
Donkey
3
['a']
```

To define a function with keyword wildcard parameters, you define a parameter that starts with **. For example, [Listing 6.11](#) defines the function `does_wildcard_keywords` with the parameter `**kwargs`. In the code block, the keyword parameters are available as keys and values in the dictionary `kwargs`.

Listing 6.11 Keyword Wildcard Parameters

[Click here to view code image](#)

```
def does_wildcard_keywords(**kwargs):
    '''Demonstrates wildcard for keyword parameters.'''
    for key, value in kwargs.items():
        print(f'{key} : {value}')

does_wildcard_keywords(one=1, name='Martha')
one : 1
name : Martha
```

You can use both positional and keyword wildcard parameters in the same function: Just define the positional parameters first and the keyword parameters second. [Listing 6.12](#) demonstrates a function using both positional and keyword parameters.

Listing 6.12 Positional and Keyword Wildcard Parameters

[Click here to view code image](#)

```
def does_wildcards(*args, **kwargs):
    '''Demonstrates wildcard parameters.'''
    print(f'Positional: {args}')
    print(f'Keyword: {kwargs}')

does_wildcards(1, 2, a='a', b=3)
Positional: (1, 2)
Keyword: {'a': 'a', 'b': 3}
```

Return Statements

Return statements define what value a function evaluates to when called. A return statement consists of the keyword `return` followed by an expression. The expression can be a simple value, a more complicated calculation, or a call to another function. [Listing 6.13](#) defines a function that takes a number as an argument and returns that number plus 1.

Listing 6.13 Return Value

[Click here to view code image](#)

```
def adds_one(some_number):  
    '''Demonstrates return statement.'''  
    return some_number + 1  
  
adds_one(1)  
2
```

Every Python function has a return value. If you do not define a return statement explicitly, the function returns the special value `None`:

[Click here to view code image](#)

```
def returns_none():  
    '''Demonstrates default return value.'''  
    pass  
  
returns_none() == None  
True
```

This example omits a return statement and then tests that the value returned is equal to `None`.

Scope in Functions

Scope refers to the availability of objects defined in code. A variable defined in the global scope is available throughout your code, whereas a variable defined in a local scope is available only in that scope. [Listing 6.14](#) defines a variable `outer` and a variable `inner`. Both variables are available in the code block of the function `shows_scope`, where you print them both.

Listing 6.14 Local and Global Scope

[Click here to view code image](#)

```
outer = 'Global scope'

def shows_scope():
    '''Demonstrates local variable.'''
    inner = 'Local scope'
    print(outer)
    print(inner)

shows_scope()
Global scope
Local scope
```

The variable `inner` is local to the function, as it is defined in the function's code block. If you try to call `inner` from outside the function, it is not defined:

[Click here to view code image](#)

```
print(inner)
-----
NameError Traceback (most recent call last)
<ipython-input-39-9504624e1153> in <module>
----> 1 print(inner)
NameError: name 'inner' is not defined
```

Understanding scope is useful when you use decorators, as described in the next section.

Decorators

A decorator enables you to design functions that modify other functions. Decorators are commonly used to set up logging using a set convention or by third-party libraries. While you may not need to write your own decorators, it is useful to understand how they work. This section walks through the concepts involved.

In Python, everything is an object, including functions. This means you can point a variable to a function. [Listing 6.15](#) defines the function `add_one(n)`, which takes a number and adds 1 to it. Next, it creates the variable `my_func`, which has the function `add_one()` as its value.

Note

When you are not calling a function, you do not use parentheses in the variable assignment. By omitting the parentheses, you are referring to the function object and not to a return value. You can see this where [Listing 6.15](#) prints `my_func`, which is indeed a function object. You can then call the function by adding the parentheses and argument to `my_func`, which returns the argument plus 1.

Listing 6.15 A Function as Variable Value

[Click here to view code image](#)

```
def add_one(n):
    '''Adds one to a number.'''
    return n + 1

my_func = add_one
print(my_func)
<function add_one at 0x1075953a0>

my_func(2)
3
```

Because functions are objects, you can use them with data structures such as dictionaries or lists. [Listing 6.16](#) defines two functions and puts them in a list pointed to by the variable `my_functions`. It then iterates through the list, assigning each function to the variable `my_func` during its iteration and calling the function during the `for` loop's code block.

Listing 6.16 Calling a List of Functions

[Click here to view code image](#)

```
def add_one(n):
    '''Adds one to a number.'''
```

```
        return n + 1

def add_two(n):
    '''Adds two to a number.'''
    return n + 2

my_functions = [add_one, add_two]

for my_func in my_functions:
    print(my_func(1))
2
3
```

Python allows you to define a function as part of another function's code block. A function defined in this way is called a *nested function*. [Listing 6.17](#) defines the function `nested()` in the code block of the function `called_nested()`. This nested function is then used as a return value for the outer function.

Listing 6.17 Nested Functions

[Click here to view code image](#)

```
def call_nested():
    '''Calls a nested function.'''
    print('outer')

    def nested():
        '''Prints a message.'''
        print('nested')

    return nested

my_func = call_nested()
outer
my_func()
nested
```

You can also wrap one function with another, adding functionality before or after. [Listing 6.18](#) wraps the function `add_one(number)` with the function `wrapper(number)`. The wrapping function takes a parameter, `number`, which it then passes to the wrapped function. It also has statements before and after calling `add_one(number)`. You can see the order of the print statements

when you call `wrapper(1)` and see that it returns the expected values from `add_one`: 1 and 2.

Listing 6.18 Wrapping Functions

[Click here to view code image](#)

```
def add_one(number):
    '''Adds to a number.'''
    print('Adding 1')
    return number + 1

def wrapper(number):
    '''Wraps another function.'''
    print('Before calling function')
    retval = add_one(number)
    print('After calling function')
    return retval
```

```
wrapper(1)
Before calling function
Adding 1
After calling function
2
```

It is also possible to go a step further and use a function as a parameter. You can pass a function as a value to a function that has a nested function definition wrapping the function that was passed. For example, [Listing 6.19](#) first defines the function `add_one(number)` as before. But now it defines the function `wrapper(number)` nested in the code block of a new function, `do_wrapping(some_func)`. This new function takes a function as an argument and then uses that function in the definition of `wrapper(number)`. It then returns the newly defined version of `wrapper(number)`. By assigning this result to a variable and calling it, you can see the wrapped results.

Listing 6.19 Nested Wrapping Function

[Click here to view code image](#)

```
def add_one(number):
    '''Adds to a number.'''
    print('Adding 1')
    return number + 1
```

```
def do_wrapping(some_func):
    '''Returns a wrapped function.'''
    print('wrapping function')

    def wrapper(number):
        '''Wraps another function.'''
        print('Before calling function')
        retval = some_func(number)
        print('After calling function')
        return retval

    return wrapper
```

```
my_func = do_wrapping(add_one)
wrapping function
```

```
my_func(1)
Before calling function
Adding 1
After calling function
2
```

You can use `do_wrapping(some_func)` to wrap any function that you like. For example, if you have the function `add_two(number)`, you can pass it as an argument just as you did `add_one(number)`:

[Click here to view code image](#)

```
my_func = do_wrapping(add_two)
my_func(1)
wrapping function
Before calling function
Adding 2
After calling function
3
```

Decorators provide syntax that can simplify this type of function wrapping. Instead of calling `do_wrapping(some_func)`, assigning it to a variable, and then invoking the function from the variable, you can simply put `@do_wrapping` at the top of the function definition. Then the function `add_one(number)` can be called directly, and the wrapping happens behind the scenes.

You can see in [Listing 6.20](#) that `add_one(number)` is wrapped in a similar fashion as in [Listing 6.18](#), but with the simpler decorator syntax.

Listing 6.20 **Decorator Syntax**

[Click here to view code image](#)

```
def do_wrapping(some_func):
    '''Returns a wrapped function.'''
    print('wrapping function')

    def wrapper(number):
        '''Wraps another function.'''
        print('Before calling function')
        retval = some_func(number)
        print('After calling function')
        return retval

    return wrapper

@do_wrapping
def add_one(number):
    '''Adds to a number.'''
    print('Adding 1')
    return number + 1

wrapping function

add_one(1)
Before calling function
Adding 1
After calling function
2
```

Anonymous Functions

The vast majority of the time you define functions, you will want to use the syntax for named functions. This is what you have seen up to this point. There is an alternative, however: the unnamed, anonymous function. In Python, anonymous functions are known as lambda functions, and they have the following syntax:

```
lambda <Parameter>: <Statement>
```


where `lambda` is the keyword designating a lambda function, `<Parameter>` is an input parameter, and `<Statement>` is the statement to execute using the parameter. The result of `<Statement>` is the return value. This is how you define a lambda function that adds one to an input value:

```
lambda x: x +1
```

In general, your code will be easier to read, use, and debug if you avoid lambda functions, but one useful place for them is when a simple function is applied as an argument to another. [Listing 6.21](#) defines the function `apply_to_list(data, my_func)`, which takes a list and a function as arguments. When you call this function with the intention of adding 1 to each member of the list, the lambda function is an elegant solution.

Listing 6.21 **Lambda Function**

[Click here to view code image](#)

```
def apply_to_list(data, my_func):  
    '''Applies a function to items in a list.'''  
    for item in data:  
        print(f'{my_func(item)}')
```

```
apply_to_list([1, 2, 3], lambda x: x + 1)
```

```
2  
3  
4
```

Summary

Functions, which are important building blocks in constructing complex programs, are reusable named blocks of code. Functions are documented with docstrings. Functions can accept parameters in a number of ways. A function uses a return statement to pass a value at the end of its execution. Decorators are special functions that wrap other functions. Anonymous, or lambda, functions are unnamed.

Questions

For Questions 1–3, refer to [Listing 6.22](#).

Listing 6.22 Functions for Questions 1–3

[Click here to view code image](#)

```
def add_prefix(word, prefix='before-'):
    '''Prepend a word.'''
    return f'{prefix}{word}'3

def return_one():
    return 1

def wrapper():
    print('a')
    retval = return_one()
    print('b')
    print(retval)
```

1. What would be the output of the following call:
`add_prefix('nighttime', 'after-')`
2. What would be the output of the following call:
`add_prefix('nighttime')`
3. What would be the output of the following call:
`add_prefix()`
4. Which line should you put above a function definition to decorate it with the function `standard_logging` ?

[Click here to view code image](#)

```
*standard_logging
**standard_logging
@standard_logging
[standard_logging]
```

5. What would be printed by the following call:
`wrapper()`

Part II

Data Science Libraries

7

NumPy

Everything should be as simple as it can be, but not simpler.

Roger Sessions (interpreting Einstein)

In This Chapter

- [Introducing third-party libraries](#)
- [Creating NumPy arrays](#)
- [Indexing and slicing arrays](#)
- [Filtering array data](#)
- [Array methods](#)
- [Broadcasting](#)

This is the first of this book's chapters on Data Science Libraries. The Python functionality explored so far in this book makes Python a powerful generic language. The libraries covered in this part of the book make Python dominant in data science. The first library we will look at, NumPy, is the backbone of many of the other data science libraries. In this chapter, you will learn about the NumPy array, which is an efficient multidimensional data structure.

Third-Party Libraries

Python code is organized into libraries. All of the functionality you have seen so far in this book is available in the Python Standard Library, which is part of any Python installation. Third-party

libraries give you capabilities far beyond this. They are developed and maintained by groups outside the organization that maintains Python itself. The existence of these groups and libraries creates a vibrant ecosystem that has kept Python a dominant player in the programming world. Many of these libraries are available in the Colab environment, and you can easily import them into a file. If you are working outside Colab, you may need to install them, which generally is done using the Python package manager, `pip`.

Installing and Importing NumPy

NumPy is preinstalled in the Colab environment, and you just need to import it. If you are working outside Colab, there are a few different ways to install it (enumerated at <https://scipy.org/install.xhtml>), but the most common is to use `pip`:

```
pip install numpy
```

Once you have NumPy installed, you can import it. When you import any library, you can change what it is called in your environment by using the keyword `as`. NumPy is typically renamed `np` during import:

```
import numpy as np
```

When you have the library installed and imported, you can then access any of NumPy's functionality through the `np` object.

Creating Arrays

A NumPy array is a data structure that is designed to efficiently handle operations on large data sets. These data sets can be of varying dimensions and can contain numerous data types—though not in the same object. NumPy arrays are used as input and output to many other libraries and are used as the underpinning of other data structures that are important to data science, such as those in Pandas and SciPy.

You can create arrays from other data structures or initialized with set values. [Listing 7.1](#) demonstrates different ways to create a one-dimensional array. You can see that the array object is displayed as having an internal list as its data. Data is not actually stored in lists, but this representation makes arrays easy to read.

Listing 7.1 Creating an Array

[Click here to view code image](#)

```
np.array([1,2,3])          # Array from list
array([1, 2, 3])

np.zeros(3)              # Array of zeros
array([0., 0., 0.])

np.ones(3)               # Array of ones
array([1., 1., 1.])

np.empty(3)              # Array of arbitrary data
array([1., 1., 1.])

np.arange(3)             # Array from range of numbers
array([0, 1, 2])

np.arange(0, 12, 3)      # Array from range of numbers
array([0, 3, 6, 9])

np.linspace(0, 21, 7)   # Array over an interval
array([ 0. ,  3.5,  7. , 10.5, 14. , 17.5, 21. ])
```

Arrays have dimensions. A one-dimensional array has only one dimension, which is the number of elements. In the case of the `np.array` method, the dimension matches that of the list(s) used as input. For the `np.zeros`, `np.ones`, and `np.empty` methods, the dimension is given as an explicit argument.

The `np.range` method produces an array in a way similar to a range sequence. The resulting dimension and values match those that would be produced by using `range`. You can specify beginning, ending, and step values.

The `np.linspace` method produces evenly spaced numbers over an interval. The first two arguments define the interval, and the third defines the number of items.

The `np.empty` method is useful in producing large arrays efficiently. Keep in mind that because the data is arbitrary, you should only use it in cases where you will replace all of the original data.

[Listing 7.2](#) shows some of the attributes of an array.

Listing 7.2 Characteristics of an Array

[Click here to view code image](#)

```
oned = np.arange(21)
oned
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
        11, 12, 13, 14, 15, 16, 17, 18, 19, 20 ])

oned.dtype      # Data type
dtype('int64')

oned.size        # Number of elements
21

oned.nbytes      # Bytes(memory) consumed by elements of the array
168

oned.shape       # Number of elements in each dimension
(21, )

oned.ndim        # Number of dimensions
1
```

If you check the data type of the array, you see that it is `np.ndarray`:

```
type(oned)
numpy.ndarray
```

Note

`ndarray` is short for *n-dimensional array*.

As mentioned earlier, you can make arrays of many dimensions. Two-dimensional arrays are used as matrixes. [Listing 7.3](#) creates a two-dimensional array from a list of three three-element lists. You can see that the resulting array has 3×3 shape and two dimensions.

Listing 7.3 **Matrix from Lists**

[Click here to view code image](#)

```
list_o_lists = [[1,2,3],
                [4,5,6],
                [7,8,9]]

twod = np.array(list_o_lists)
twod
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

twod.shape
(3, 3)

twod.ndim
2
```

You can produce an array with the same elements but different dimensions by using the `reshape` method. This method takes the new shape as arguments. [Listing 7.4](#) demonstrates using a one-dimensional array to produce a two-dimensional one and then producing one-dimensional and three-dimensional arrays from the two-dimensional one.

Listing 7.4 **Using reshape**

[Click here to view code image](#)

```
oned = np.arange(12)
oned
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

twod = oned.reshape(3,4)
twod
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
twod.reshape(12)  
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
twod.reshape(2,2,3)  
array([[[ 0,  1,  2],  
        [ 3,  4,  5]],  
       [[ 6,  7,  8],  
        [ 9, 10, 11]]])
```

The shape you provide for an array must be consistent with the number of elements in it. For example, if you take the 12-element array `twod` and try to set its dimensions with a shape that does not include 12 elements, you get an error:

[Click here to view code image](#)

```
twod.reshape(2,3)
```

```
-----  
ValueError                                Traceback (most recent  
<ipython-input-295-0b0517f762ed> in <module>  
----> 1 twod.reshape(2,3)
```

```
ValueError: cannot reshape array of size 12 into shape (2,3)
```



Reshaping is commonly used with the `np.zeros`, `np.ones`, and `np.empty` methods to produce multidimensional arrays with default values. For example, you could create a three-dimensional array of ones like this:

[Click here to view code image](#)

```
np.ones(12).reshape(2,3,2)  
array([[[1., 1.],  
        [1., 1.],  
        [1., 1.]],  
       [[1., 1.],  
        [1., 1.],  
        [1., 1.]])
```

Indexing and Slicing

You can access the data in arrays by indexing and slicing. In [Listing 7.5](#), you can see that indexing and slicing with a one-dimensional array is the same as with a list. You can index individual elements from the start or end of an array by supplying an index number or multiple elements using a slice.

Listing 7.5 Indexing and Slicing a one-Dimensional Array

[Click here to view code image](#)

```
oned = np.arange(21)
oned
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
        11, 12, 13, 14, 15, 16, 17, 18, 19, 20 ])

oned[3]
3

oned[-1]
20

oned[3:9]
array([3, 4, 5, 6, 7, 8])
```

For multidimensional arrays, you can supply one argument for each dimension. If you omit the argument for a dimension, it defaults to all elements of that dimension. So, if you supply a single number as an argument to a two-dimensional array, that number will indicate which row to return. If you supply single-number arguments for all dimensions, a single element is returned. You can also supply a slice for any dimension. In return you get a subarray of elements, whose dimensions are determined by the length of your slices. [Listing 7.6](#) demonstrates various options for indexing and slicing a two-dimensional array.

Listing 7.6 Indexing and Slicing a Two-Dimensional Array

[Click here to view code image](#)

```
twod = np.arange(21).reshape(3,7)
twod
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20]])
```

```

twod[2]           # Accessing row 2
array([14, 15, 16, 17, 18, 19, 20])

twod[2, 3]        # Accessing item at row 2, column 3
17

twod[0:2]          # Accessing rows 0 and 1
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13]])

twod[:, 3]         # Accessing column 3 of all rows
array([ 3, 10, 17])

twod[0:2, -3:]    # Accessing the last three columns of rows 0 and 1
array([[ 4,  5,  6],
       [11, 12, 13]])

```

You can assign new values to an existing array, much as you would with a list, by using indexing and slicing. If you assign a values to a slice, the whole slice is updated with the new value. [Listing 7.7](#) demonstrates how to update a single element and a slice of a two-dimensional array.

Listing 7.7 Changing Values in an Array

[Click here to view code image](#)

```

twod = np.arange(21).reshape(3,7)
twod
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20]])

twod[0,0] = 33
twod
array([[33,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20]])

twod[1:, :3] = 0
array([[33,  1,  2,  3,  4,  5,  6],
       [ 0,  0,  0, 10, 11, 12, 13],
       [ 0,  0,  0, 17, 18, 19, 20]])

```

Element-by-Element Operations

An array is not a sequence. Arrays do share some characteristics with lists, and on some level it is easy to think of the data in an array as a list of lists. There are many differences between arrays and sequences, however. One area of difference is when performing operations between the items in two arrays or two sequences.

Remember that when you do an operation such as multiplication with a sequence, the operation is done to the sequence, not to its contents. So, if you multiply a list by zero, the result is a list with a length of zero:

```
[1, 2, 3]*0  
[]
```

You cannot multiply two lists, even if they are the same length:

[Click here to view code image](#)

```
[1, 2, 3]*[4, 5, 6]
```

```
-----  
TypeError                                Traceback (most recent  
<ipython-input-325-f525a1e96937> in <module>  
----> 1 [1, 2, 3]*[4, 5, 6]
```

```
TypeError: can't multiply sequence by non-int of type 'list'
```

You can write code to perform operations between the elements of lists. For example, [Listing 7.8](#) demonstrates looping through two lists in order to create a third list that contains the results of multiple pairs of elements. The `zip()` function is used to combine the two lists into a list of tuples, with each tuple containing elements from each of the original lists.

Listing 7.8 Element-by-Element Operations with Lists

[Click here to view code image](#)

```
L1 = list(range(10))  
L2 = list(range(10, 0, -1))  
L1  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
L2
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

L3 = []
for i, j in zip(L1, L2):
    L3.append(i*j)
L3
[0, 9, 16, 21, 24, 25, 24, 21, 16, 9]
```

While it is possible to use loops to perform element-by-element operations on lists, it is much simpler to use NumPy arrays for such operations. Arrays do element-by-element operations by default. [Listing 7.9](#) demonstrates multiplication, addition, and division operations between two arrays. Notice that the operations in each case are done between the elements of the arrays.

Listing 7.9 Element-by-Element Operations with Arrays

[Click here to view code image](#)

```
array1 = np.array(L1)
array2 = np.array(L2)
array1*array2
array([ 0,  9, 16, 21, 24, 25, 24, 21, 16,  9])

array1 + array2
array([10, 10, 10, 10, 10, 10, 10, 10, 10, 10])

array1 / array2
array([0.         , 0.11111111, 0.25         , 0.42857143, 0.66666667,
        1.         , 1.5         , 2.33333333, 4.         , 9.         ])
```

Filtering Values

One of the most used aspects of NumPy arrays and the data structures built on top of them is the ability to filter values based on conditions of your choosing. In this way, you can use an array to answer questions about your data.

[Listing 7.10](#) shows a two-dimensional array of integers, called `twod`. A second array, `mask`, has the same dimensions as `twod`, but it contains

Boolean values. `mask` specifies which elements from `twod` to return. The resulting array contains the elements from `twod` whose corresponding positions in `mask` have the value `True`.

Listing 7.10 Filtering Using Booleans

[Click here to view code image](#)

```
twod = np.arange(21).reshape(3,7)
twod
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20]])

mask = np.array([[ True,  False,  True,  True,  False,  True,  False],
                 [ True,  False,  True,  True,  False,  True,  False],
                 [ True,  False,  True,  True,  False,  True,  False]])
twod[mask]
array([ 0,  2,  3,  5,  7,  9, 10, 12, 14, 16, 17, 19])
```

Comparison operators that you have seen returning single Booleans before return arrays when used with arrays. So, if you use the less-than operator (`<`) against the array `twod` as follows, the result will be an array with `True` for every item that is below five and `False` for the rest:

```
twod < 5
```

You can use this result as a mask to get only the values that are `True` with the comparison. For example, [Listing 7.11](#) creates a mask and then returns only the values of `twod` that are less than 5.

Listing 7.11 Filtering Using Comparison

[Click here to view code image](#)

```
mask = twod < 5
mask
array([[ True,  True,  True,  True],
       [ True, False, False, False],
       [False, False, False, False]])

twod[mask]
array([0, 1, 2, 3, 4])
```

As you can see, you can use comparison and order operators to easily extract knowledge from data. You can also combine these comparisons to create more complex masks. [Listing 7.12](#) uses `&` to join two conditions to create a mask that evaluates to `True` only for items meeting both conditions.

Listing 7.12 Filtering Using Multiple Comparisons

[Click here to view code image](#)

```
mask = (twod < 5) & (twod%2 == 0)
mask
array([[ True, False,  True, False],
       [ True, False, False, False],
       [False, False, False, False]])

twod[mask]
array([0, 2, 4])
```

Note

Filtering using masks is a process that you will use time and time again, especially with Pandas DataFrames, which are built on top of NumPy arrays. You will learn about DataFrames in [Chapter 9](#), “Pandas.”

Views Versus Copies

NumPy arrays are designed to work efficiently with large data sets. One of the ways this is accomplished is by using views. When you slice or filter an array, the returned array is, when possible, a view and not a copy. A view allows you to look at the same data differently. It is important to understand that memory and processing power are not used in making copies of data every time you slice or filter. If you change a value in a view of an array, you change that value in the original array as well as any other views that represent that item. For example, [Listing 7.13](#) takes a slice from the array `data1` and names it `data2`. It then replace the value 11 in `data2` with -1.

When you go back to `data1`, you can see that the item that used to have a value of 11 is now set to -1.

Listing 7.13 Changing Values in a View

[Click here to view code image](#)

```
data1 = np.arange(24).reshape(4,6)
data1
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])

data2 = data1[:2,3:]
data2
array([[ 3,  4,  5],
       [ 9, 10, 11]])

data2[1,2] = -1
data2
array([[ 3,  4,  5],
       [ 9, 10, -1]])

data1
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, -1],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

This behavior can lead to bugs and miscalculations, but if you understand it, you can gain some important benefits when working with large data sets. If you want to change data from a slice or filtering operation without changing it in the original array, you can make a copy. For example, in [Listing 7.14](#), notice that when an item is changed in the copy, the original array remains unchanged.

Listing 7.14 Changing Values in a Copy

[Click here to view code image](#)

```
data1 = np.arange(24).reshape(4,6)
data1
array([[ 0,  1,  2,  3,  4,  5],
```

```

        [ 6,  7,  8,  9, 10, 11],
        [12, 13, 14, 15, 16, 17],
        [18, 19, 20, 21, 22, 23]])

data2 = data1[:2,3:].copy()
data2
array([[ 3,  4,  5],
       [ 9, 10, 11]])

data2[1,2] = -1
data2
array([[ 3,  4,  5],
       [ 9, 10, -1]])

data1
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])

```

Some Array Methods

NumPy arrays have built-in methods both to get statistical summary data and to perform matrix operations. [Listing 7.15](#) shows methods producing summary statistics. There are methods to get the maximum, minimum, sum, mean, and standard deviation. All these methods produce results across the whole array unless an axis is specified. If an axis value of 1 is specified, an array with results for each row is produced. With an axis value of 0, an array of results is produced for each column.

Listing 7.15 Introspection

[Click here to view code image](#)

```

data = np.arange(12).reshape(3,4)
data
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

data.max()          # Maximum value
11

```

```
data.min()          # Minimum value
0

data.sum()          # Sum of all values
66

data.mean()         # Mean of values
5.5

data.std()          # Standard deviation
3.452052529534663

data.sum(axis=1)     # Sum of each row
array([ 6, 22, 38])

data.sum(axis=0)     # Sum of each column
array([12, 15, 18, 21])

data.std(axis=0)     # Standard deviation of each row
array([3.26598632, 3.26598632, 3.26598632, 3.26598632])

data.std(axis=1)     # Standard deviation of each column
array([1.11803399, 1.11803399, 1.11803399])
```

[Listing 7.16](#) demonstrates some of the matrix operations that are available with arrays. These include returning the transpose, returning matrix products, and returning the diagonal. Remember that you can use the multiplication operator (*) between arrays to perform element-by-element multiplication. If you want to calculate the dot product of two matrices, you need to use the @ operator or the .dot() method.

Listing 7.16 **Matrix Operations**

[Click here to view code image](#)

```
A1 = np.arange(9).reshape(3,3)
A1
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

A1.T          # Transpose
array([[0, 3, 6],
```

```

        [1, 4, 7],
        [2, 5, 8]])

A2 = np.ones(9).reshape(3,3)
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])

A1 @ A2           # Matrix product
array([[ 3.,  3.,  3.],
       [12., 12., 12.],
       [21., 21., 21.]])

A1.dot(A2)       # Dot product
array([[ 3.,  3.,  3.],
       [12., 12., 12.],
       [21., 21., 21.]])

A1.diagonal()    # Diagonal
array([0, 4, 8])

```

An array, unlike many sequence types, can contain only one data type. You cannot have an array that contains both strings and integers. If you do not specify the data type, NumPy guesses the type, based on the data. [Listing 7.17](#) shows that when you start with integers, NumPy sets the data type to `int64`. You can also see, by checking the `nbytes` attribute, that the data for this array takes 800 bytes of memory.

Listing 7.17 Setting Type Automatically

[Click here to view code image](#)

```

darray = np.arange(100)
darray
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])

darray.dtype
dtype('int64')

```

```
darray.nbytes
```

```
800
```

For larger data sets, you can control the amount of memory used by setting the data type explicitly. The `int8` data type can represent numbers from -128 to 127 , so it would be adequate for a data set of $1-99$. You can set an array's data type at creation by using the parameter `dtype`. Listing 7.18 does this to bring the size of the data down to 100 bytes.

Listing 7.18 Setting Type Explicitly

[Click here to view code image](#)

```
darray = np.arange(100, dtype=np.int8)
```

```
darray
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99],
      dtype=int8)
```

```
darray.nbytes
```

```
100
```

Note

You can see the many available NumPy data types at <https://numpy.org/devdocs/user/basics.types.xhtml>.

Because an array can store only one data type, you cannot insert data that cannot be cast to that data type. For example, if you try to add a string to the `int8` array, you get an error:

[Click here to view code image](#)

```
darray[14] = 'a'
```

```
ValueError
```

```
Traceback (most recent
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-335-17df5782f85b> in <module>
----> 1 darray[14] = 'a'

ValueError: invalid literal for int() with base 10: 'a'
```

A subtle error with array type occurs if you add to an array data of a finer granularity than the array's data type; this can lead to data loss. For example, say that you add the floating-point number 0.5 to the int8 array:

```
darray[14] = 0.5
```

The floating-point number 0.5 is cast to an int, which leaves a value of 0:

```
darray[14]
0
```

As you can see, it is important to understand your data when deciding on the best data type.

Broadcasting

You can perform operations between arrays of different dimensions. Operations can be done when the dimension is the same or when the dimension is one for at least one of the arrays. [Listing 7.19](#) adds 1 to each element of the array A1 three different ways: first with an array of ones with the same dimensions (3, 3), then with an array with one dimension of one (1, 3), and finally by using the integer 1.

Listing 7.19 **Broadcasting**

[Click here to view code image](#)

```
A1 = np.array([[1,2,3],
               [4,5,6],
               [7,8,9]])

A2 = np.array([[1,1,1],
               [1,1,1],
               [1,1,1]])
```

```
A1 + A2  
array([[ 2,  3,  4],  
       [ 5,  6,  7],  
       [ 8,  9, 10]])
```

```
A2 = np.array([1,1,1])  
A1 + A2  
array([[ 2,  3,  4],  
       [ 5,  6,  7],  
       [ 8,  9, 10]])
```

```
A1 + 1  
array([[ 2,  3,  4],  
       [ 5,  6,  7],  
       [ 8,  9, 10]])
```

In all three cases, the result is the same: an array of dimension (3, 3). This is called *broadcasting* because a dimension of one is expanded to fit the higher dimension. So if you do an operation with arrays of dimensions (1, 3, 4, 4) and (5, 3, 4, 1), the resulting array will have the dimensions (5, 3, 4, 4). Broadcasting does not work with dimensions that are different but not one.

[Listing 7.20](#) does an operation on arrays with the dimensions (2, 1, 5) and (2, 7, 1). The resulting array has the dimensions (2, 7, 5).

Listing 7.20 Expanding Dimensions

[Click here to view code image](#)

```
A4 = np.arange(10).reshape(2,1,5)  
A4  
array([[[0, 1, 2, 3, 4]],  
       [[5, 6, 7, 8, 9]]])
```

```
A5 = np.arange(14).reshape(2,7,1)  
A5  
array([[[ 0],  
        [ 1],  
        [ 2],  
        [ 3],  
        [ 4],  
        [ 5],  
        [ 6]],
```

```

[[ 7],
 [ 8],
 [ 9],
 [10],
 [11],
 [12],
 [13]])

A6 = A4 - A5
A6
array([[ 0,  1,  2,  3,  4],
       [-1,  0,  1,  2,  3],
       [-2, -1,  0,  1,  2],
       [-3, -2, -1,  0,  1],
       [-4, -3, -2, -1,  0],
       [-5, -4, -3, -2, -1],
       [-6, -5, -4, -3, -2]],
      [[-2, -1,  0,  1,  2],
       [-3, -2, -1,  0,  1],
       [-4, -3, -2, -1,  0],
       [-5, -4, -3, -2, -1],
       [-6, -5, -4, -3, -2],
       [-7, -6, -5, -4, -3],
       [-8, -7, -6, -5, -4]])

A6.shape
(2, 7, 5)

```

NumPy Math

In addition to the NumPy array, the NumPy library offers many mathematical functions, including trigonometric functions, logarithmic functions, and arithmetic functions. These functions are designed to be performed with NumPy arrays and are often used in conjunction with data types in other libraries. This section takes a quick look at NumPy polynomials.

NumPy offers the class `poly1d` for modeling one-dimensional polynomials. To use this class, you need to import it from NumPy:

```
[1] 1 from numpy import poly1d
```

Then create a polynomial object, giving the coefficients as an argument:

```
poly1d((4,5))  
poly1d([4, 5])
```

If you print a `poly1d` object, it shows the polynomial representation:

[Click here to view code image](#)

```
c = poly1d([4,3,2,1])  
print(c)  
      3      2  
4 x + 3 x + 2 x + 1
```

If for a second argument you supply the value `True`, the first argument is interpreted as roots rather than coefficients. The following example models the polynomial resulting from the calculation $(x - 4)(x - 3)(x - 2)(x - 1)$:

[Click here to view code image](#)

```
r = poly1d([4,3,2,1], True)  
print(r)  
      4      3      2  
1 x - 10 x + 35 x - 50 x + 24
```

You can evaluate a polynomial by supplying the `x` value as an argument to the object itself. For example, you can evaluate the preceding polynomial for a value of `x` equal to 5:

```
r(5)  
24.0
```

The `poly1d` class allows you to do operations between polynomials, such as addition and multiplication. It also offers polynomial functionality as special class methods. [Listing 7.21](#) demonstrates the use of this class with polynomials.

Listing 7.21 Polynomials

[Click here to view code image](#)

```
p1 = poly1d((2,3))  
print(p1)  
2 x + 3
```



```

p2 = poly1d((1,2,3))
print(p2)
      2
1 x + 2 x + 3

print(p2*p1)          # Multiplying polynomials
      3      2
2 x + 7 x + 12 x + 9

print(p2.deriv())     # Taking the derivative
2 x + 2

print(p2.integ())     # Returning anti-derivative
      3      2
0.3333 x + 1 x + 3 x

```

The `poly1d` class is just one of the many specialized mathematical tools offered in the NumPy toolkit. These tools are used in conjunction with many of the other specialized tools that you will learn about in the coming chapters.

Summary

The third-party library NumPy is a workhorse for doing data science in Python. Even if you don't use NumPy arrays directly, you will encounter them because they are building blocks for many other libraries. Libraries such as SciPy and Pandas build directly on NumPy arrays. NumPy arrays can be made in many dimensions and data types. You can tune them to control memory consumption by controlling their data type. They are designed to be efficient with large data sets.

Questions

1. Name three differences between NumPy arrays and Python lists.
2. Given the following code, what would you expect for the final value of `d2`?

[Click here to view code image](#)

```
d1 = np.array([[0, 1, 3],
               [4, 2, 9]])
d2 = d1[:, 1:]
```

3. Given the following code, what would you expect for the final value of `d1[0,2]`?

[Click here to view code image](#)

```
d1 = np.array([[0, 1, 3],
               [4, 2, 9]])
d2 = d1[:, 1:]
d2[0,1] = 0
```

4. If you add two arrays of dimensions (1, 2, 3) and (5, 2, 1), what will be the resulting array's dimensions?
5. Use the `poly1d` class to model the following polynomial:

$$6x^4 + 2x^3 + 5x^2 + x - 10$$

8

SciPy

Most people use statistics like a drunk man uses a lamppost; more for support than illumination.

Andrew Lang

In This Chapter

- Math with NumPy
- [Introduction to SciPy](#)
- [scipy.misc submodule](#)
- [scipy.special submodule](#)
- [scipy.stats submodule](#)

[Chapter 7](#), “NumPy,” covers NumPy arrays, which are foundational building blocks for many data science–related libraries. This chapter introduces the SciPy library, which is a library for mathematics, science, and engineering.

SciPy Overview

The SciPy library is a collection of packages that build on NumPy to provide tools for scientific computing. It includes submodules that deal with optimization, Fourier transformations, signal processing, linear algebra, image processing, and statistics, among others. This chapter touches on three submodules: the `scipy.misc` submodule, the `scipy.special`

submodule, and `scipy.stats`, which is the submodule most useful for data science.

This chapter also uses the library `matplotlib` for some examples. It has visualization capabilities for numerous plot types as well as images. The convention for importing its plotting library is to import it with the name `plt`:

```
import matplotlib.pyplot as plt
```

The `scipy.misc` Submodule

The `scipy.misc` submodule contains functions that don't have a home elsewhere. One fun function in this module is `scipy.misc.face()`, which can be run with this code:

[Click here to view code image](#)

```
from scipy import misc
import matplotlib.pyplot as plt
face = misc.face()
plt.imshow(face)
plt.show()
```

You can try this yourself to generate the output.

The `ascent` function returns a grayscale image that is available for use and demos. If you call `ascent()`, the result is a two-dimensional NumPy array:

[Click here to view code image](#)

```
a = misc.ascent()
print(a)
[[ 83  83  83 ... 117 117 117]
 [ 82  82  83 ... 117 117 117]
 [ 80  81  83 ... 117 117 117]
 ...
 [178 178 178 ...  57  59  57]
 [178 178 178 ...  56  57  57]
 [178 178 178 ...  57  57  58]]
```

If you pass this array to the `matplotlib` plot object, you see the image shown in [Figure 8.1](#):

```
plt.imshow(a)  
plt.show()
```



Figure 8.1 Demo Image from the `scipy.misc` Submodule

As you can see in this example, you use the `plt.imshow()` method to visualize images.

The `scipy.special` Submodule

The `scipy.special` submodule contains utilities for mathematical physics. It includes Airy functions, elliptical functions, Bessel functions, Struve functions, and many more. The majority of these functions support broadcasting and are compatible with NumPy arrays. To use the functions, you simply import `scipy.special` from SciPy and call the functions directly. For example, you can calculate the factorial of a number by using the `special.factorial()` function:

[Click here to view code image](#)

```
from scipy import special
special.factorial(3)
6.0
```

You can calculate the number of combinations or permutations as follows:

[Click here to view code image](#)

```
special.comb(10, 2)
45.0

special.perm(10, 2)
90.0
```

This example shows 10 items and choosing 2 of them at a time.

Note

`scipy.special` has a `scipy.stats` submodule, but it is not meant for direct use. Rather, you use the `scipy.stats` submodule for your statistics needs. This submodule is discussed next.

The `scipy.stats` Submodule

The `scipy.stats` submodule offers probability distributions and statistical functions. The following sections take a look at just a few of the distributions offered in this submodule.

Discrete Distributions

SciPy offers some discrete distributions that share some common methods. These common methods are demonstrated in [Listing 8.2](#) using a binomial distribution. A binomial distribution involves some number of trials, with each trial having either a success or failure outcome.

Listing 8.2 Binomial Distribution

[Click here to view code image](#)

```
from scipy import stats
B = stats.binom(20, 0.3) # Define a binomial distribution consisting
                        # 20 trials and 30% chance of success

B.pmf(2) # Probability mass function (probability that a sample is e
0.02784587252426866

B.cdf(4) # Cumulative distribution function (probability that a
        # sample is less than 4)
0.2375077788776017

B.mean # Mean of the distribution
6.0

B.var()# Variance of the distribution
4.199999999999999

B.std()# Standard deviation of the distribution
2.0493901531919194

B.rvs()# Get a random sample from the distribution
5

B.rvs(15) # Get 15 random samples
array([ 2,  8,  6,  3,  5,  5, 10,  7,  5, 10,  5,  5,  5,  2,  6])
```

If you take a large enough random sample of the distribution:

[Click here to view code image](#)

```
rvs = B.rvs(size=100000)  
rvs  
array([11,  4,  4, ...,  7,  6,  8])
```

You can use `matplotlib` to plot it and get a sense of its shape (see [Figure 8.2](#)):

[Click here to view code image](#)

```
import matplotlib.pyplot as plt  
plt.hist(rvs)  
plt.show()
```

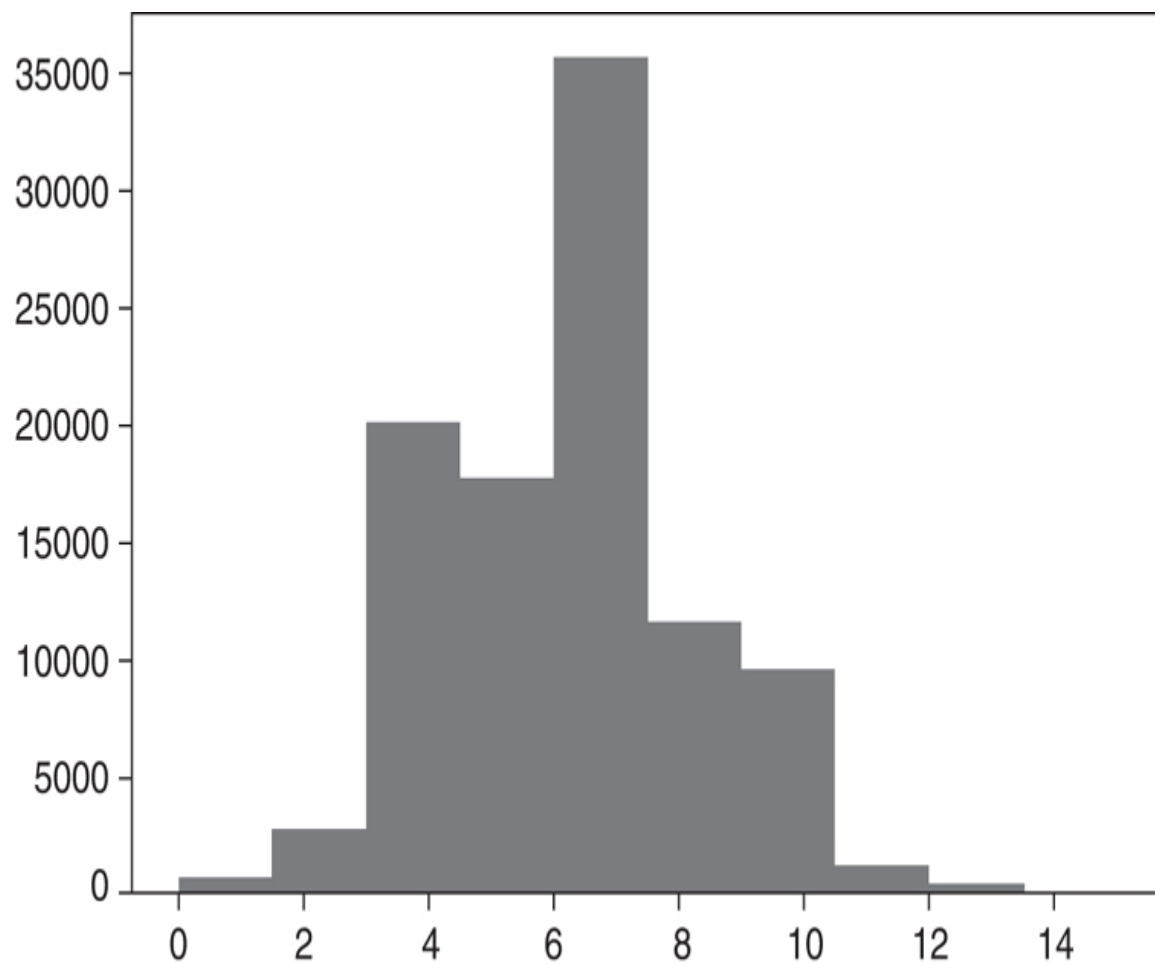


Figure 8.2 Binomial Distribution

The numbers along the bottom of the distribution in [Figure 8.2](#) represent the number of successes in each 20-trial experiment. You can see that 6 out of 20 is the most common result, and it matches the 30% success rate.

Another distribution modeled in the `scipy.stats` submodule is the Poisson distribution. This distribution models the probability of a certain number of individual events happening across some scope of time. The shape of the distribution is controlled by its mean, which you can set by using the `mu` keyword. For example, a lower mean, such as 3, will skew the distribution to the left, as shown in [Figure 8.3](#):

[Click here to view code image](#)

```
P = stats.poisson(mu=3)  
rvs = P.rvs(size=10000)  
rvs  
array([4, 4, 2, ..., 1, 0, 2])  
  
plt.hist(rvs)  
plt.show()
```

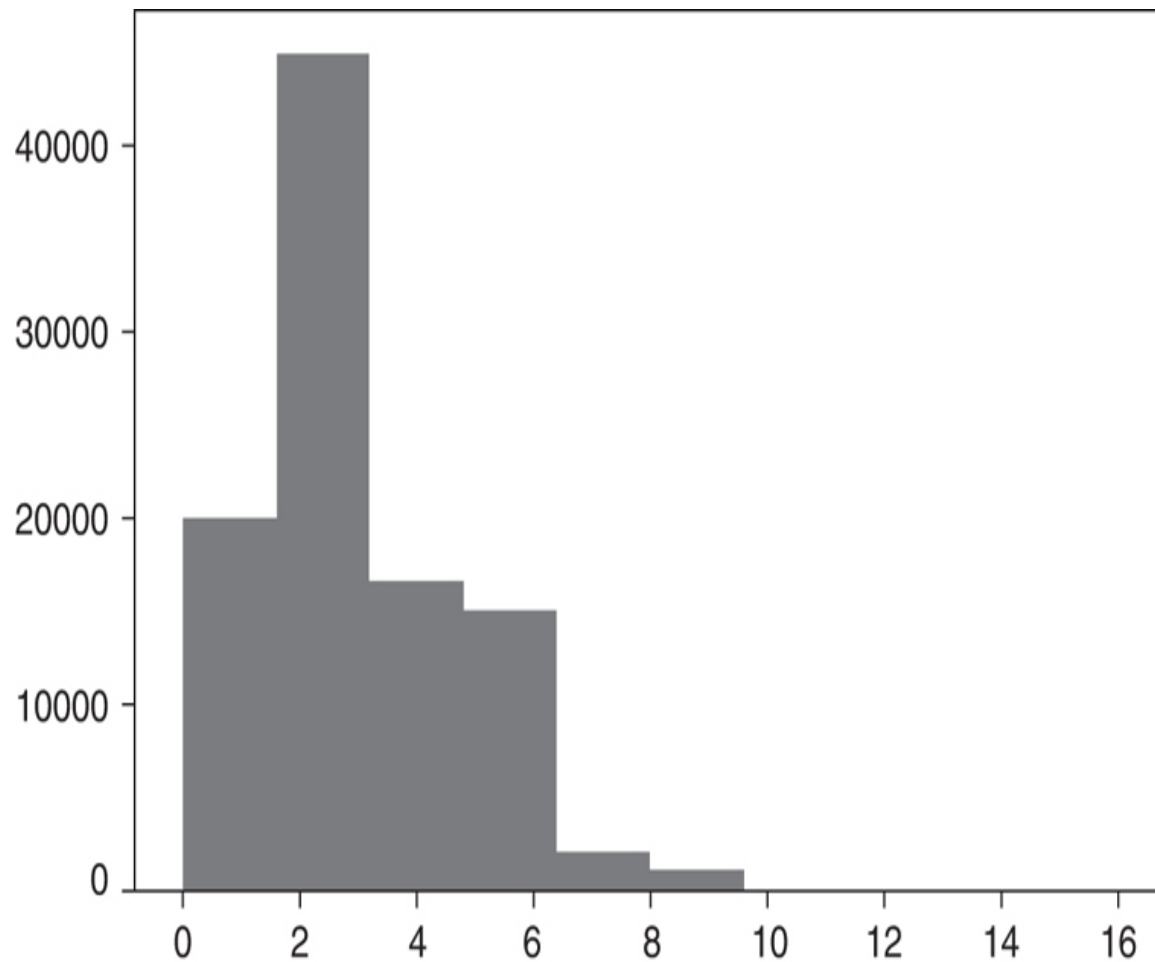


Figure 8.3 Poisson Distribution Skewed Left

A higher mean, such as 15, pushes the distribution to the right, as you can see in [Figure 8.4](#):

[Click here to view code image](#)

```
P = stats.poisson(mu=15)  
rvs = P.rvs(size=100000)  
plt.hist(rvs)  
plt.show()
```

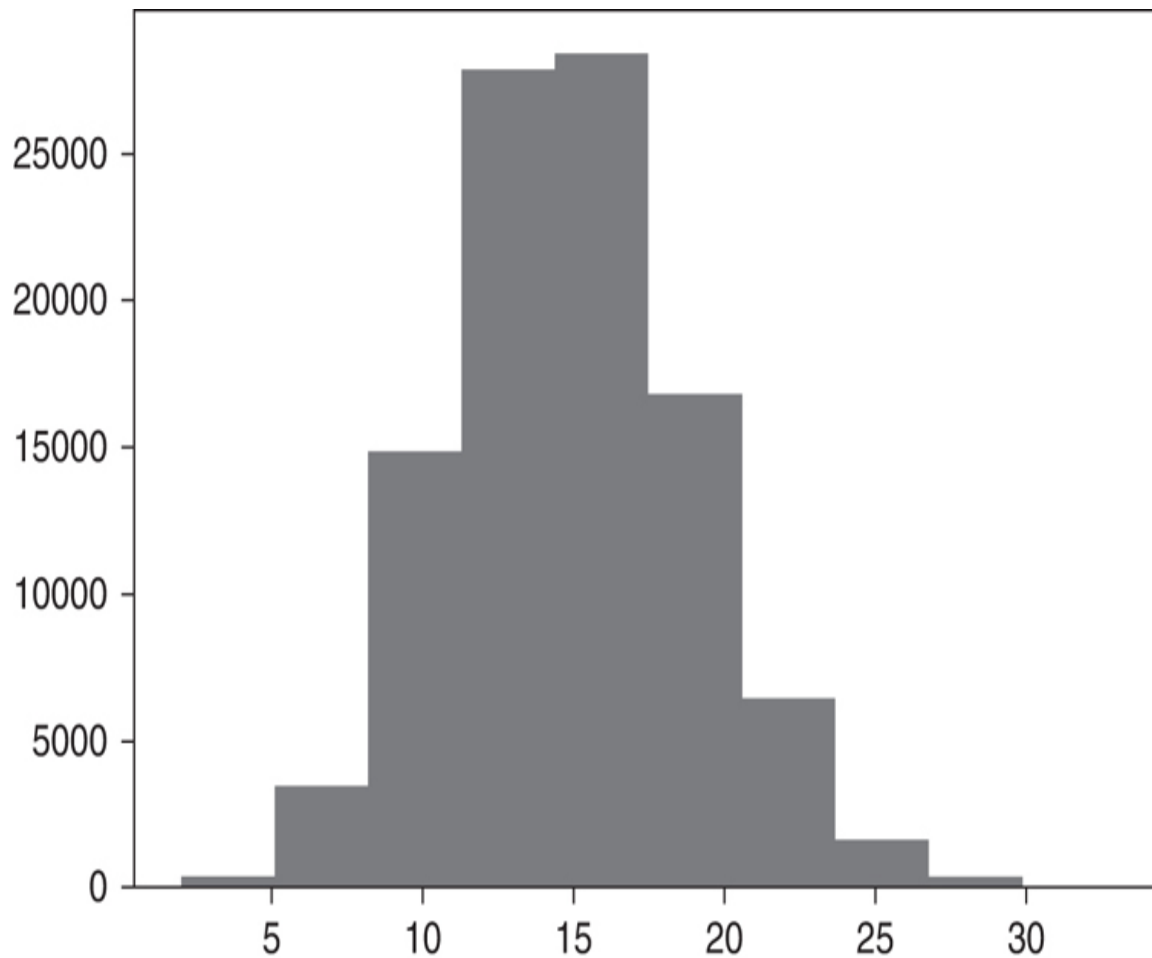


Figure 8.4 Poisson Distribution Skewed Right

Other discrete distributions modeled in the `scipy.stats` submodule include the Beta-binomial, Boltzmann (truncated Planck), Planck (discrete exponential), geometric, hypergeometric, logarithmic, and Yule–Simon, among others. At the time of this writing, there are 14 distributions modeled in the `scipy.stats` submodule.

Continuous Distributions

The `scipy.stats` submodule includes many more continuous than discrete distributions; it has 87 continuous distributions as of this writing. These distributions all take arguments for location (`loc`) and scale (`scale`). They all default to a location of 0 and scale of 1.0.

One continuous distribution modeled is the Normal distribution, which may be familiar to you as the bell curve. In this symmetric distribution, half of the data is to the left of the mean and half to the right. Here's how you can make a normal distribution using the default location and scale:

[Click here to view code image](#)

```
N = stats. norm()  
rvs = N.rvs(size=100000)  
plt.hist(rvs, bins=1000)  
plt.show()
```

[Figure 8.5](#) shows this distribution plotted.

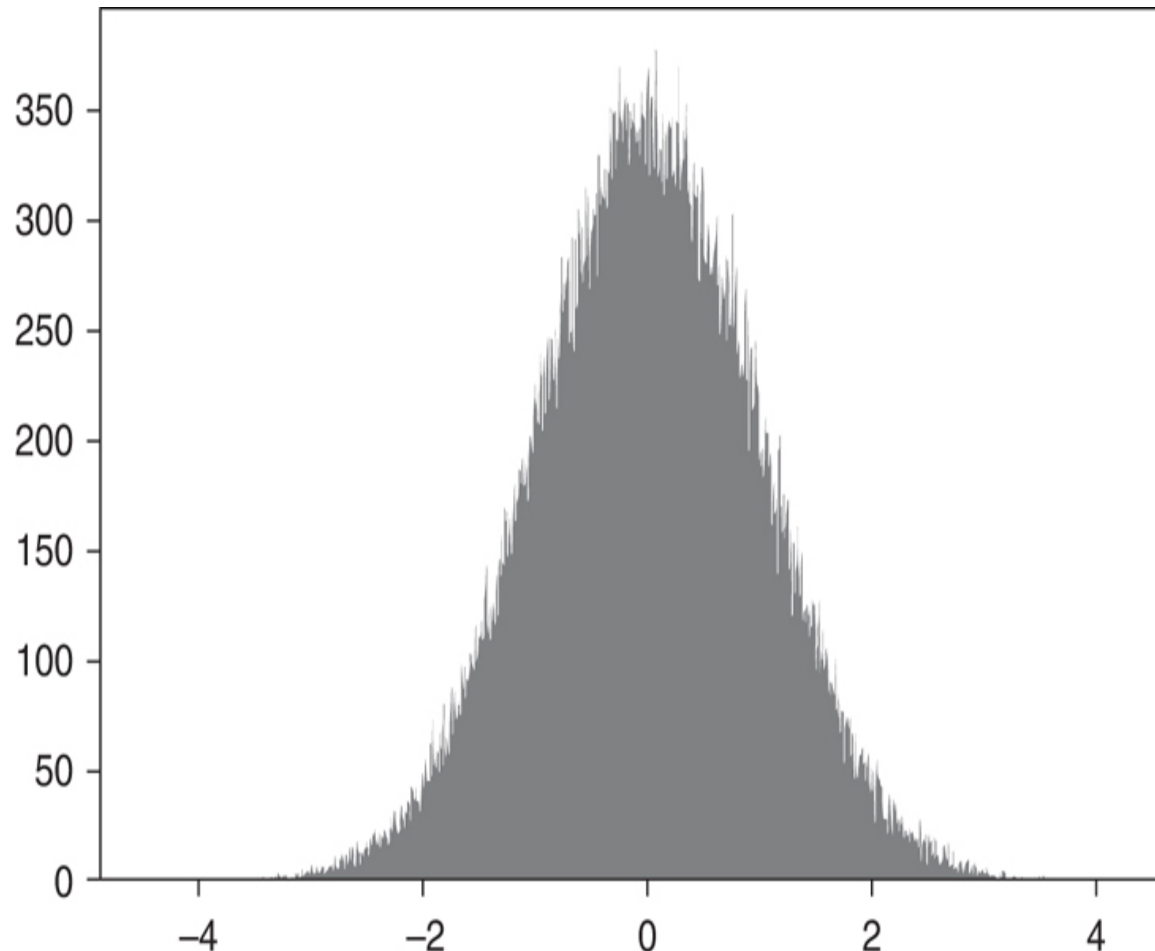


Figure 8.5 Bell Curve

You can see that the distribution is centered on 0 and is encompassed roughly between -4 and 4 . [Figure 8.6](#) shows the effects of creating a second normal distribution—this time setting the location to 30 and the scale to 50:

[Click here to view code image](#)

```
N1 = stats.norm(loc=30,scale=50)  
rvs = N1.rvs(size=100000)  
plt.hist(rvs, bins=1000)  
plt.show()
```

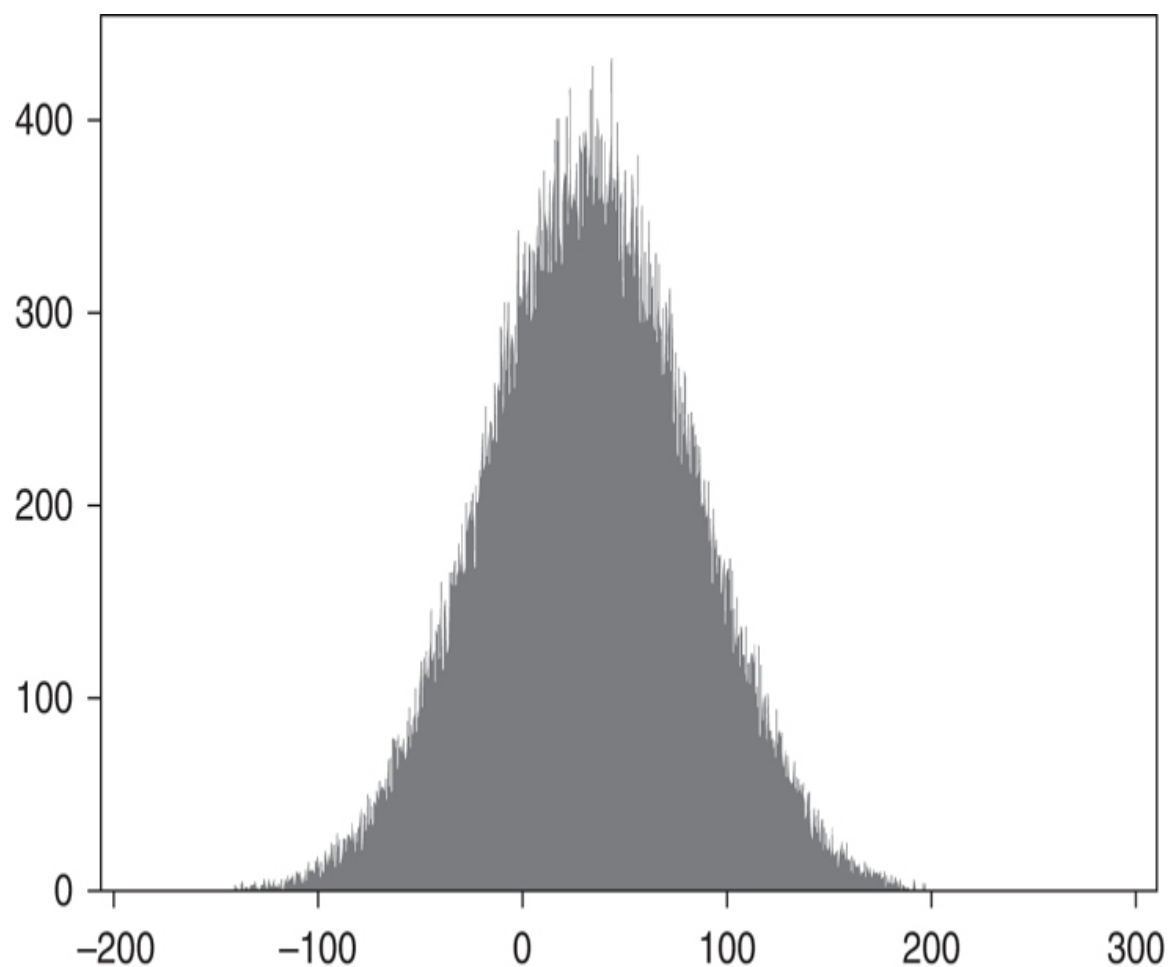


Figure 8.6 Offset Bell Curve

Notice that the distribution is now centered around 30 and encompasses a much wider range of numbers. Continuous distributions share some

common functions, which are modeled in [Listing 8.3](#). Notice that this listing uses the second Normal distribution with the offset location and greater standard deviation.

Listing 8.3 Normal Distribution

[Click here to view code image](#)

```
N1 = stats.norm(loc=30, scale=50)
N1.mean() # Mean of the distribution, which matches the loc value
30.0

N1.pdf(4) # Probability density function
0.006969850255179491

N1.cdf(2) # Cumulative distribution function
0.28773971884902705

N1.rvs() # A random sample
171.55168607574785

N1.var() # Variance
2500.0

N1.median()# Median
30.0

N1.std() # Standard deviation
50.0
```

Note

If you try the examples shown here, some of your values may differ due to random number generation.

The following continuous distribution is an exponential distribution, which is characterized by an exponentially changing curve, either up or down (see [Figure 8.7](#)):

[Click here to view code image](#)

```
E = stats.expon()  
rvs = E.rvs(size=100000)  
plt.hist(rvs, bins=1000)  
plt.show()
```

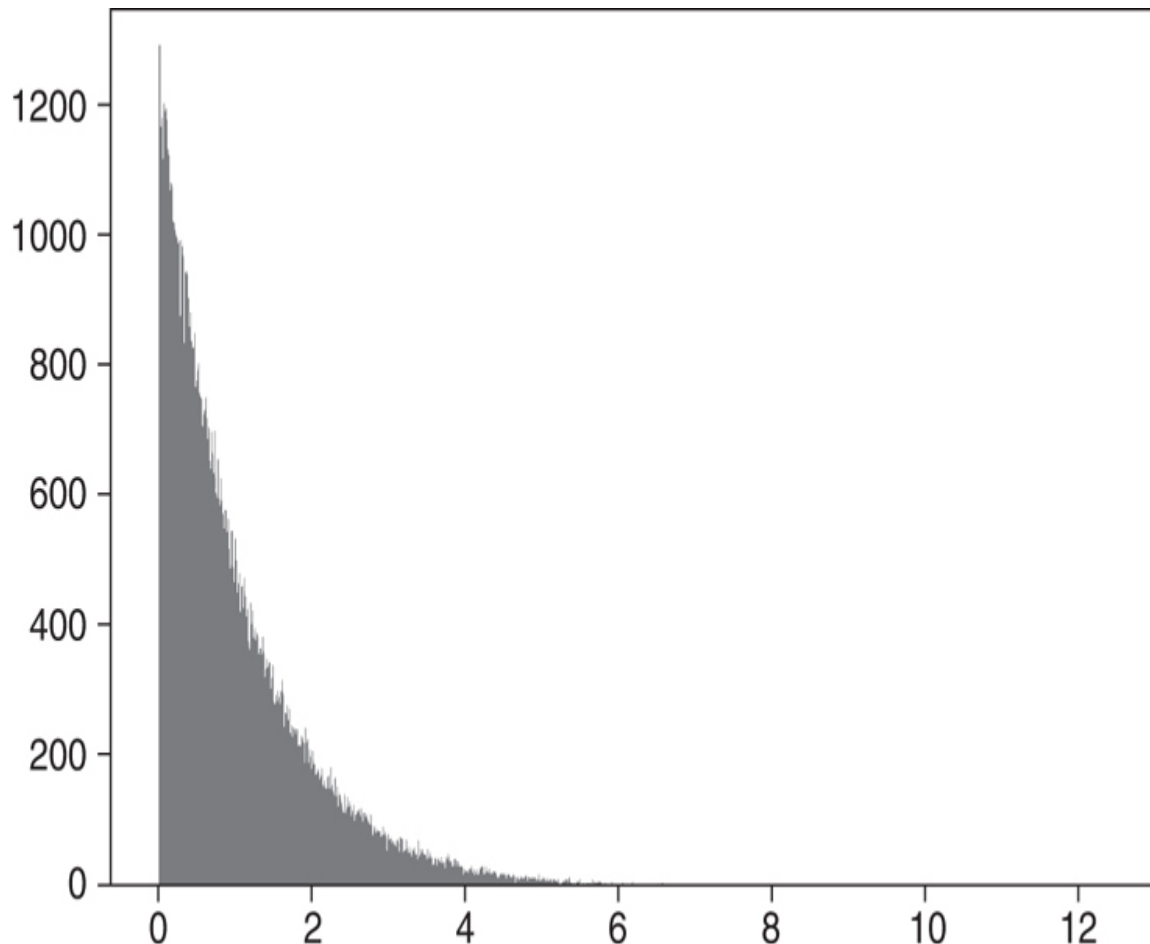


Figure 8.7 Exponentially Changing Distribution

You can see that [Figure 8.7](#) displays a curve as you would expect from an exponential function. The following is a uniform distribution, which is has a constant probability and is also known as a rectangular distribution:

[Click here to view code image](#)

```
U = stats.uniform()  
rvs = U.rvs(size=10000)  
rvs
```

```
array([8.24645026e-01, 5.02358065e-01, 4.95390940e-01, ...,  
      8.63031657e-01, 1.05270200e-04, 1.03627699e-01])
```

```
plt.hist(rvs, bins=1000)  
plt.show()
```

This distribution gives an even probability over a set range. Its plot is shown in [Figure 8.8](#).

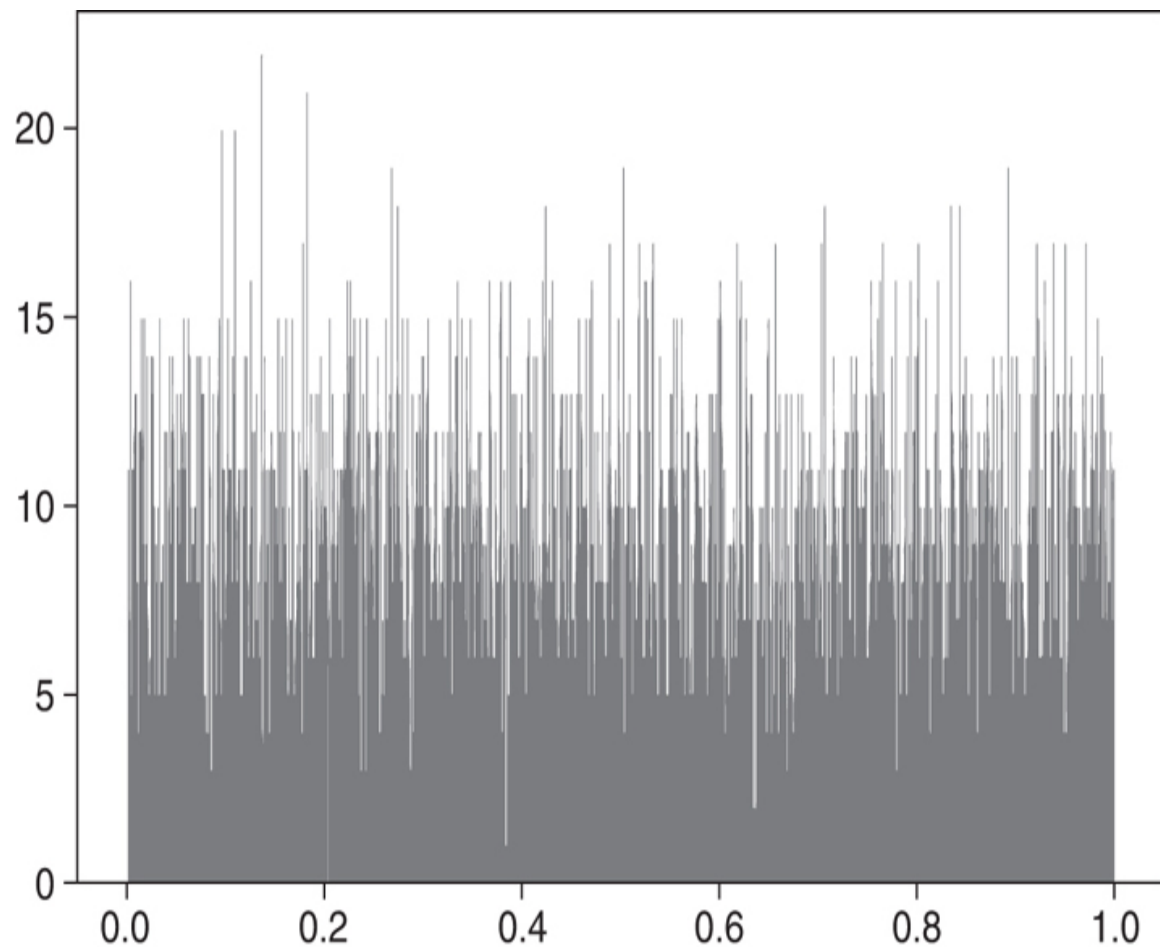


Figure 8.8 Uniform Distribution

Summary

The NumPy and SciPy libraries both offer utilities for solving complex mathematical problems. These two libraries cover a great breadth and depth of resources, and entire books have been devoted to their application. You have seen only a few of the many capabilities. These libraries are the first places you should look when you embark on solving or modeling complex mathematical problems.

Questions

1. Use the `scipy.stats` submodule to model a Normal distribution with a mean of 15.
2. Generate 25 random samples from the distribution modeled in Question 1.
3. Which `scipy` submodule has utilities designed for mathematical physics?
4. What method is provided with a discrete distribution to calculate its standard deviation?

9

Pandas

*To clarify, *add* data.*

Edward R. Tufte

In This Chapter

- [Introduction to Pandas DataFrames](#)
- [Creating DataFrames](#)
- [DataFrame introspection](#)
- [Accessing data](#)
- [Manipulating DataFrames](#)
- [Manipulating DataFrame data](#)

The Pandas DataFrame, which is built on top of the NumPy array, is probably the most commonly used data structure. DataFrames are like supercharged spreadsheets in code. They are one of the primary tools used in data science. This chapter looks at creating DataFrames, manipulating DataFrames, accessing data in DataFrames, and manipulating that data.

About DataFrames

A Pandas DataFrame, like a spreadsheet, is made up of columns and rows. Each column is a `pandas.Series` object. A DataFrame is, in some ways, similar to a two-dimensional NumPy array, with labels for the columns and index. Unlike a NumPy array, however, a DataFrame can contain different data types. You can think of a `pandas.Series` object as a one-dimensional

NumPy array with labels. The `pandas.Series` object, like a NumPy array, can contain only one data type. The `pandas.Series` object can use many of the same methods you have seen with arrays, such as `min()`, `max()`, `mean()`, and `medium()`.

The usual convention is to import the Pandas package aliased as `pd`:

```
import pandas as pd
```

Creating DataFrames

You can create DataFrames with data from many sources, including dictionaries and lists and, more commonly, by reading files. You can create an empty DataFrame by using the `DataFrame` constructor:

[Click here to view code image](#)

```
df = pd.DataFrame()
print(df)
Empty DataFrame
Columns: []
Index: []
```

As a best practice, though, DataFrames should be initialized with data.

Creating a DataFrame from a Dictionary

You can create DataFrames from a list of dictionaries or from a dictionary, where each key is a column label with the values for that key holding the data for the column. [Listing 9.1](#) shows how to create a DataFrame by creating a list of data for each column and then creating a dictionary with the column names as keys and these lists as the values. The listing shows how to then pass this dictionary to the `DataFrame` constructor to construct the DataFrame.

Listing 9.1 Creating a DataFrame from a Dictionary

[Click here to view code image](#)

```
first_names = ['shanda', 'rolly', 'molly', 'frank',
               'rip', 'steven', 'gwen', 'arthur']
```