Introduction to Python for Engineers and Scientists

Open Source Solutions for Numerical Computation

Sandeep Nagar

Introduction to Python for Engineers and Scientists

Open Source Solutions for Numerical Computation

Sandeep Nagar

Apress®

Introduction to Python for Engineers and Scientists: Open Source Solutions for Numerical Computation

Sandeep Nagar
New York, USA

Cover image by Freepik (www.freepik.com)

Dedicated to my wife, Rashmi, and my daughter, Aliya.

Table of Contents

About the Author

Sandeep Nagar, PhD (Material Science, KTH, Sweden) teaches and consults on the use of Python and other open source software for data science/analysis. In addition to teaching at universities, he frequently gives workshops covering open source software and is interested in developing hardware for scientific experiments.

About the Technical Reviewer

Michael Thomas has worked in software development for more than 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has more than 10 years of experience working with mobile devices. His current focus is using mobile devices to accelerate information transfer between patients and health care providers in the medical sector.

Acknowledgments

I wish to thank Steve, Mark, and the whole team at Apress for bringing out this book in such a nice format. I also wish to thank the Python community for answering questions on forums, which helped me understand difficult concepts.

CHAPTER 1

Philosophy of Python

1.1 Introduction

Python emerged as a leader [1] among well-established and optimized languages including C, C++, and Java for very simple reasons. Python incorporates the principles of the philosophy that complex tasks can be done in simple ways. We tend to think that real-world, complex problems need complex pathways to produce complex solutions. The developers of Python embraced the exact opposite philosophy. Python was created to have an extremely fast and simple learning curve and development process for software engineers. As a result, it is considered the most general-purpose programming language since users can work in almost any study domain and still be able to find a useful piece of code for themselves. Python harnessed the power of the open source movement, which helped it amass a huge user base from virtually all walks of life. Being open source in nature, Python allowed people to make small programs and share them with each other with ease. In Python, a group of programs for performing various tasks makes up a module (package). At the time of writing, there are over 117,181 [2] modules that have been submitted by an even larger number of developers around the world. The large number of modules and developers has allowed Python's use to jump rapidly within the computer science community and finally grab the number-one position as the most favored [1] programming language.

The philosophy of Python can be defined in a single sentence:

Python is a multipurpose, portable, object-oriented, high-level programming language that enables an interactive environment to code in a minimalistic way.

Let's look at the individual characteristics of Python.

1.1.1 High-Level Programming

A high-level programming language is a language that is separated from the details of a particular computer. A low-level language is one where users directly feed the machine code to a processor for obtaining results. A high-level language provides facilities like library functions for performing the low-level tasks and also provides ways to define the code in a form readable to humans, which is then translated into machine language to be fed to a processor.

1.1.2 Interactive Environment

To a large extent, Python derives its philosophy from the ABC language. The syntax structure was largely derived from C and UNIX's Bourne shell environments. They served as inspiration for the interpretative nature of the working environment. The interpretive nature means that Python presents a REPL-based interactive environment to a developer. The interactive shell of the Python programming language is commonly known as REPL (Read-Evaluate-Print-Loops) because it

- reads what a user types,

- evaluates what it reads,

- prints out the return value after evaluation, and

- loops back and does it all over again.

This kind of interactive working environment proves especially useful for debugging. It also helps in prototyping a problem where each step can be visualized for its output in a live fashion. Users can check the results of a particular code as soon as they finish writing it. The way to work with Python's REPL is to write the code, analyze the results, and continue this process until the final result is computed. In addition to allowing quick and easy evaluation of Python statements, the language also showcases the following:

- A searchable history
  - Users can press the Up and Down keys on the keyboard to browse through past commands instead of writing them again.

- Tab completion
  - Users can simply press the Tab key after writing a few letters for a command and it will auto-complete it.
  - Consequently, tab completion eliminates syntax errors.
  - If more than one option matches when the Tab key is pressed, the options are displayed at the command prompt so users can choose which one they intended.

- Many helpful key bindings
  - The key bindings depend on the operating systems.
  - The key bindings help in quick operations where key combinations are equivalent to a particular operation.

- Help and documentation

  - Getting help on topics and locating documentation is quite easy in python.

  - Users can feed any argument as a string (that is, characters enclosed within double quotation marks) to the built-in function `help()`.

In addition to being interactive, Python is an interpreted language. Whereas other languages require source code to be converted into an executable and then run on a machine—in other words, AOT (ahead-of-time) compilation—Python runs the program directly from source code. Python converts the source code into an intermediate form called *bytecodes* and then translates the bytecodes into the native language of the computer and runs it. This is a type of OTF (on-the-fly) compilation, enabling portability and making it easier for a developer to write and check the program in an interactive manner.

1.1.3 Object Orientation

Most primitive programming languages were procedural in nature. In other words, a set of procedures was defined to compute a problem and the flow of information was controlled within these procedures to obtain a desired output. Hence, a program was merely divided into blocks of codes that interacted with each other where one block of code defined a computation subtask belonging to a computational problem understudy. Conversely, an object-oriented programming (OOP) language deals with data as an object on which different methods act to produce a desired result. Everything computable is treated as an object. Its nature is defined as its properties. These properties can be probed by functions, which are called methods. The abstract nature of objects makes it possible to invent objects of the user's choice and apply the programming concepts for a variety of applications.

1.1.4 Multipurpose Nature

As discussed in the preceding section, the OOP-based architecture of Python enables developers from different walks of life to use and enrich the language in their fields of expertise. Virtually all fields of computations have used Python. You can define a module specific for one kind of problem. In fact, Python modules exist for specific fields of studies, as shown in Table 1-1.

Table 1-1. *List of Fields of Study and Corresponding Python Modules*

Field of Study	Name of Python Module
Scientific Computation	`scipy, numpy, sympy`
Statistics	`pandas`
Networking	`networkx`
Cryptography	`pyOpenSSL`
Game Development	`PyGame`
Graphic User Interface	`pyQT`
Machine Learning	`scikit-learn, tensorflow`
Image Processing	`scikit-image`
Plotting	`Matplotlib`
Database	`SQLAlchemy`
HTML and XML parsing	`BeautifulSoup`
Natural Language Processing	`nltk`
Testing	`nose`

It is impossible to list all the modules for a given application as the modules are being created on a daily basis. Table 1-1 lists the most widely used modules at the time of writing. Developers from all over the world

are using and developing modules at a tremendous pace. Since Python can be used in a wide arena of computing domains, it is truly the most multipurpose programming language yet.

1.1.5 Minimalistic Design

The minimalistic design philosophy of Python means that it emphasizes code readability. It also provides a syntax structure that allows programmers to express concepts in fewer lines of code than in languages such as C++ and Java. Moreover, Python provides the means to write programs that can be scaled up easily. Python features a dynamic type system where a Python interpreter guesses the type of an object by its definition, which avoids the user's need to define the same.

The core philosophy of the language is summarized by Tim Peters in the document *The Zen of Python* (PEP 20) [3], which includes the following aphorisms:

- Beautiful is better than ugly.

- Explicit is better than implicit.

- Simple is better than complex.

- Complex is better than complicated.

- Flat is better than nested.

- Sparse is better than dense.

- Readability counts.

- Special cases aren't special enough to break the rules.

- Although practicality beats purity.

- Errors should never pass silently.

- Unless explicitly silenced.

- In the face of ambiguity, refuse the temptation to guess.

- There should be one–and preferably only one–obvious way to do it.

- Although that way may not be obvious at first unless you're Dutch.

- Now is better than never.

- Although never is often better than *right now*.

- If the implementation is hard to explain, it's a bad idea.

- If the implementation is easy to explain, it may be a good idea.

- Namespaces are one honking great idea—let's do more of those!

1.1.6 Portability

Since Python belongs to an open source community, it has been ported (that is, adapted to work on) to many many platforms so that Python code written on one platform can run without modification on others (except system-dependent features). Python has been posted for popular operating systems including Linux, Windows, Macintosh, Solaris, and Sony PlayStation.

1.1.7 Extensibility

Rather than providing all functionalities in its core program, Python's creators designed it to be highly extensible. Users can thus choose to have functionality as per their requirements. For example, if a user needs to work on differential equations, then that user can use a module for differential equations rather than all users having that functionality but

never using it. Python can also be embedded in existing applications that need a programmable interface.

Python programs can be embedded into other programs written in programming languages such as Julia, C, C++, and so on. Furthermore, other programming language codes can be embedded into Python. This feature has enabled the use of a lot of legacy code written in other languages and already optimized for speed and stability, thus avoiding the replication of efforts and hereby tremendously increasing the productivity of an organization. Hence, Python has been embraced with open arms by industry and academia alike.

1.2 History

The development of the Python programming language dates back to the 1980s. Guido van Rossum at CWI in the Netherlands began implementing it in December 1989. This was an era when computing devices were becoming increasingly powerful and reliable with every advancing day. Van Rossum named the language after the BBC TV show *Monty Python's Flying Circus*. Python 1.0 was released to the public in 1994, Python 2.0 in 2000, and Python 3.0 in 2008. Howver, Python 3 was not created to be backward compatible with Python 2, which made it less practical for users who were already developing with Python 2. As a result, a lot of developers have continued using Python 2, even now. Nonetheless, the future belongs to Python 3, which has been developed in a more efficient manner. Hence, we will discuss Python-3-based codes in this book.

1.2.1 Python 2 vs. Python 3

At this point, it is important to note that Python 3 is not backward compatible. The Internet is full of codes written in Python 2. It is important to learn how to convert these codes from Python 2 to Python 3 [4, 5].

You can understand the technical details of their differences when you understand their basic structures and basic usage.

1.3 Python and Engineering

Engineering problems employ numerical computations both on a small scale and on a large scale. Thus, engineering applications require a programming language to fit well in both these regimes. There are very few languages that can boast this quality, so Python is definitely a winner here. While running large computational tasks on bigger computational architectures, memory management, speed, and reliability are the key parameters. Python, being an interpretative language, is generally considered to be a slower option in this regard, but its ability to use faster codes written in C, Java, and Fortran using the interlinking packages cython, jython, and f2p allows speed-intensive tasks to be run in their native language within a Python code. This ability has relieved a lot of coders around the world who wondered if already optimized codes must be rewritten in Python.

Another engineering concern is the ability of a programming language to communicate with physical devices efficiently. Electronic devices are connected via wires and Bluetoorh wireless technology to the Internet. Using an appropriate Python module, users can connect to a compatible device to derive data from it and then visualize it in the desired platform. A variety of microcontrollers allow Python to run its hardware with ease. MicroPython [6] is specially designed for this purpose. MicroPython is a lean and efficient implementation of the Python 3 programming language, which includes a small subset of the Python standard library and is optimized to run on microcontrollers and in constrained environments. Even microcomputers like Raspberry Pi allow the running of Python programs accessing the input-output devices. This enables cost-effective prototyping of an engineering problem.

Users of MATLAB argue that Simulink is one of the easiest ways of prototyping and simulating an engineering problem because they don't need to code. Instead, users just stitch together pieces of codes represented by graphical blocks on a graphics terminal. (Scilab also provides a similar platform called Xcos.) Python still lacks this ability and budding programmers can take this up as a challenge. A large community of developers is eagerly waiting for such a solution, but most engineers won't mind investing a day or two to learn a new programming language that can enhance their productivity that a ready-made tool cannot provide.

1.4 Modular Programming

The modular nature of Python programming incorporates the complex tasks being divided into small modules that seamlessly interact with each other. Modules make both development and debugging easier, and they can be simply imported to enable the use of various functions.

Python comes with thousands of modules to perform various tasks. Since this book is an introductory text for scientific computation, the usage of just a few basic necessary modules (shown in Table 1-2) will be discussed.

Table 1-2. *Basic Python Modules*

Package Name	Meaning	Purpose
numpy	Numerical Python	Numerical computation
scipy	Scientific Python	Scientific computations
sympy	Symbolic Python	Symbolic computing
matplotlib	Mathematical Plotting Library	For plotting graphs

It is important to note that whereas modern-day personal computers offer large memories, microcomputers like Raspberry Pi have limited memories. Hence, judicious use of these memory resources is highly recommended. Since all modules occupy some memory, they should be installed on a need-to basis. Also, they should be imported in the program as and when required. Python allows selective import of specific functions to optimize memory usage. It is considered a good practice to write programs that avoid wasteful use of resources.

Mentioning the use of each module is beyond the scope of this book. Modules will be introduced as required by the topic at hand. Users are encouraged to explore various modules and their documentation for usage. A general use of modules and their functions will be dealt with at a later point in this book.

1.5 Summary

Python has gained a lot of attention worldwide owing to its flat learning and steep development curves. It has gained the number-one spot in recent times in terms of popularity of programming languages. Owing to a large base of developers due to its open source model, Python has achieved a rich library of modules for various tasks required to solve many engineering problems. Hence, Python-educated engineers can fulfill the demands of modern industry, which demands fast and efficient solutions to its problems.

1.6 Bibliography

[1] https://spectrum.ieee.org/computing/
     software/the-2017-top-programming-languages.

[2] https://pypi.python.org/pypi.

[3]   https://www.python.org/dev/peps/pep-0020/.

[4]   https://wiki.python.org/moin/PortingToPy3k/
       BilingualQuickRef.

[5]   https://docs.python.org/3/howto/pyporting.
       html.

[6]   https://micropython.org/.

CHAPTER 2

Introduction to Python Basics

2.1 Introduction

The best way to learn Python is to try Python on the Python interpreter. In this chapter, we will illustrate how to install and use the Python interpreter. Installation can be done on all popular operating systems. As examples, we will demonstrate installion on Windows, Ubuntu (Linux), and Mac OS X.

2.2 Installation

The Python interpreter has been provided for a variety of platforms. An operating system defines a scheme to store files and run codes in a specific manner. These different schemes require different settings of the Python interpreter so that it can be compatible with the host it uses for computing resources. The three most popular choices for operating systems are Windows, Linux, and Mac OS X. While the Python interpreter comes pre-installed in Linux and Mac OS X, a Windows operating system would need a separate installation software. The Python community's web site [1] provides the primary means of knowing the progress in this domain. It also provides the authentic downloads [2] of source code for installation.

2.2.1 Windows

Users can download a 32-bit or a 64-bit installer file as per the system's configurations and simply click it to start the installation process. As soon as it is finished, a symbol in the applications installation appears. When this symbol is double-clicked, it opens the Python shell (Figure 2-1), which is a program that runs the Python interpreter.
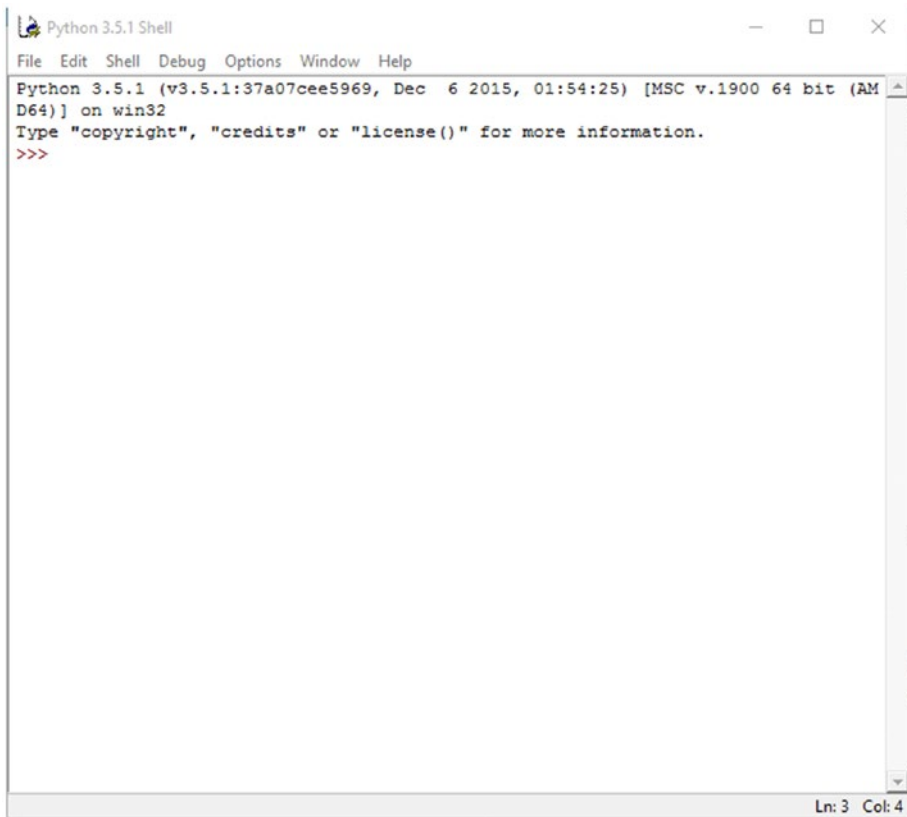
Figure 2-1. Python IDLE on Windows OS

The downloaded IDLE also provides an option for a text editor that can run scripts. Alternatively, if you have properly defined the path of the interpreter to the system, you can simply open the terminal and type the following:
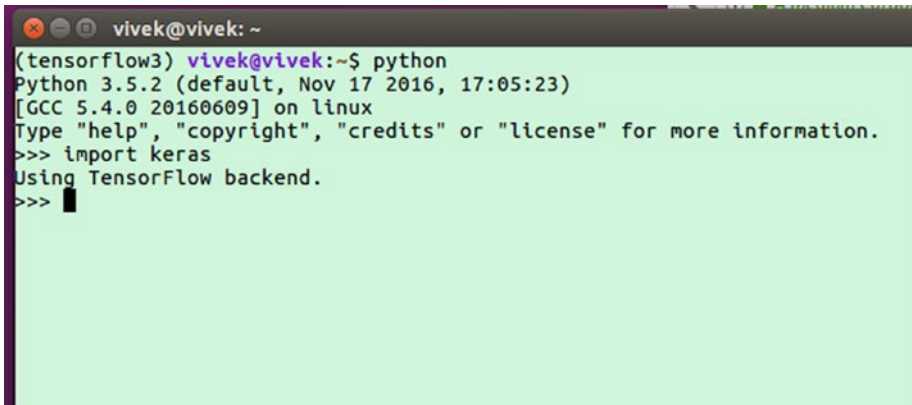
```
1   $python hello.py
```

Here we are assuming that you wish to run a Python program named `hello.py`. (All Python programs are saved with the extension `.py`.)

We suggest learning the usage of the Python terminal for interactive programming and executing multiline codes in the manner previously discussed. Henceforth, the code written at the Python prompt (`>>>`) will signify that it is being written line-by-line and being executed in a similar fashion so that the programmer is working interactively at the Python prompt. The `.py` files used in this book are executed at the system's terminal.

2.2.2 Ubuntu

The Python interpreter is pre-installed in almost all Linux distributions. We will look at an example of Ubuntu Linux System here. The Python 3 interpreter can be called from a Linux command terminal by typing `python3` and pressing Enter. The terminal goes to the Python interpreter session (Figure 2-2) where the programmer can type python expressions.

15

Figure 2-2. *Python interpreter running on Ubuntu*

To exit the interpreter, you need to type exit() on the Python interpreter. When you come back to the Linux terminal, you can now type Linux expressions.

2.2.3 Mac OS X

Mac OS X works in the same way as all Linux distributions. The Python interpreter comes pre-installed. You can start the Python interpreter session by simply typing python3 on the terminal. (See Figure 2-3.)

```
$python3
Python 3.6.2 |Anaconda custom (x86_64)| (default, Jul 20 2017, 13:14:59)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 2-3. *Python interpreter running on Mac OS X*

2.3 Using the Python Interpreter

Python is an interpreted language as opposed to compiled languages
like C, C++, Java, and so on. Each line of code is interpreted and executed
as bytecode as opposed to a single machine code file. This makes the
architecture of computation quite different from traditional languages. For
example, suppose line 5 of a multiline Python program has a syntax error.
In this case, the program will execute all commands up to line 4 and will
then show an error. In the case of compiled languages, the compilation
steps would show an error and the program would not run at all in the
sense that it would not make the machine code file. To understand
this difference in detail, you first need to understand the processes of
compilation and interpretation.

In the case of compiled languages, a compiler translates the *human-readable* code into a *machine-readable* assembly language. Machine readable code is called *object code* given by object files. These object files can be run directly on machines. As an example, let's assume that a C code is defined as follows and stored in a file named `hello.c`:

```
1  /* Hello World program */
2
3  #include <stdio.h>
4
5  main()
6  {
7  printf ("Hello World");
8
9  }
```

To compile this code on a UNIX-like machine with a `gcc` compiler, we give the following command:

```
gcc hello.c -o hello
```

This creates an object code named `hello.` During compilation, the header `stdio.h` is used to understand the input-output statements such as `printf("Hello World")`. The object code can then be executed by writing on a UNIX terminal:

```
./hello
```

The user can share the object file and, if the microprocessor architecture is the same as that of another user, it will be executed uniformly. If, however, the architecture of the machine is different, the source code must first be compiled for the target machine and then made to execute.

This is not the case with Python. Being an interpreted language, it employs an *interpreter* that interprets the code into an intermediate code and then to machine code. An interpreter reads the source text of a program, analyzes it, and executes it one line at a time. This process is slow since the interpreter spends a lot of time analyzing strings of characters to figure out what they mean. For example, to type `hello world` as in a C program, Python will require the following:

```
1  >>>print ("hello world")
2  >>>hello world
```

In just one line, an interpreter scans the world *print* and looks for what it means. In the Python interpreter, it means to print to a particular device. A device can be a computer terminal, printer plotter, and so forth. By default, it is a computer terminal. The print command also demands *arguments* that are scanned in the second step as a string `hello world`. (A string in Python can be enclosed in " or ""). Hence, the complete interpretation of the line is to print the string `hello world` on a computer terminal.

When a program composes hundreds and thousands of lines, a compilation process will yield a faster result because the object code needs to be only compiled once and then run directly on a microprocessor. An interpreted code will check for interpretations each time it needs to be processed. On the other hand, even a single syntax error will not let the compilation process complete, resulting in compilation error. Conversely, a Python interpreter will interpret the Python code up to the point where any kind of error is encountered.

Despite its being inherently slow, Python has become a favorite among scientists and engineers for being extremely simple, intuitive, and powerful due to its rich library of modules for various computational tasks.

2.4 Anaconda IDE

Anaconda IDE (Integrated Development Environment) is one of the most convenient and complete ways to get a working environment for a beginner as well as for an experienced developer. An IDE presents a Python command line interpreter—a text editor with advanced features like syntax completion, keyword highlighting, and file browser. Anaconda provides all of these and many more features in a simple and intuitive way. It can be installed in a very simple manner using the graphical installer provided at its web site [3].

The Spyder IDE within Figure 2-4 provides an easy way to use an IDE environment for development of Python files. It resents three panes:
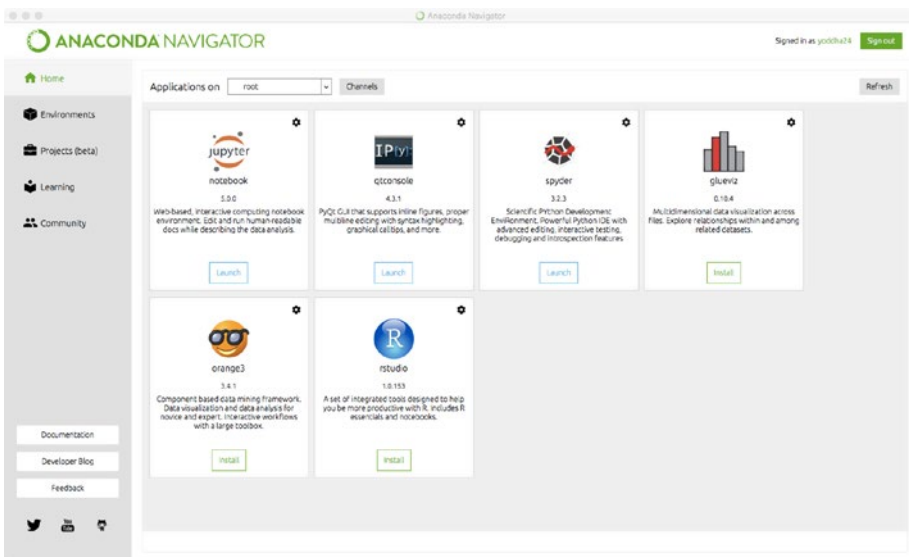
Figure 2-4. *Anaconda IDE*

1. Editor
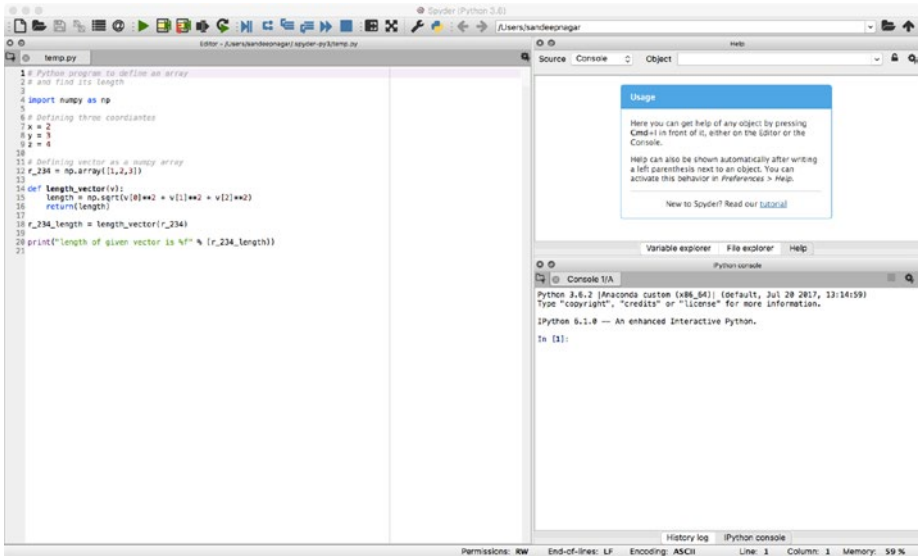
Figure 2-5. *Anaconda IDE*

– You can write a multiline program here and execute
it by clicking Play or clicking the Run option in the
main menu bar.

2. IPython Console

– You can write Python expressions here and execute
them by pressing the Enter key.

3. Variable Explorer / File Explorer / Help

– Variable Explorer

* The variable explorer presents the details of
all variables (explained later) created during a
session and their properties (name, type, size,
and values).

    – File Explorer

        * The file explorer presents the details of all files created during a session and their properties (name, size, kind, and date modified).

    – Help

        * This tool presents useful documentation on any topic inquired by a developer.

We have written this book using Anaconda IDE on a Mac OS X 10.12.6 system. Since Python programs are portable, they will run in a similar fashion to any Python 3 interpreter.

2.5 Python as a Calculator

Let's start working with the Python interpreter to understand it more clearly. To begin, type the command following the Python interpreter's command prompt >>> and press the Enter key on the keyboard.

In its simplest form, the Python interpreter can be used as a calculator. On the Python command prompt, the following commands can be checked for 2 + 4:

```
 1  >>>2+4
 2  6
 3  >>>2.+4
 4  6.0
 5  >>>2.0+4
 6  6.0
 7   >>>2.0+4.
 8  6.0
 9  >>>2.0+4.0
10  6.0
```

As per these calculations, 2 + 4 yields 6, whereas 2 + 4. yields 6.0. 6 and 6.0 are two different objects for a computer. 6 is an *integer* stored in lesser space than 6.0, which is a *floating point number*. Just like two types of numerical data, data can be classified in different kinds of objects (need not be numeric). Python treats all entries as objects. An object needs to be defined with its attributes/properties. For example, a floating point number has different rules of addition, subtraction, printing on screen, and representation of graphs when compared to an integer. For this reason, a floating point number is a quite different `data type` when compared to an integer. A detailed list of data types will be discussed later.

It is also important to note that to define the floating point number 4.0, writing even `4.` is sufficient. `4.0` and `4.` are equivalent. Similarly, `0.4` and `.4` both mean the mathematical number 0.4. Also, Python dynamically assigned the `type` to the objects defined by analyzing their values, so 2 became an `int` (integer type) and 2.0 became a `float` (floating point number type) of numerical data. We did not explicitly define an object's type; it was dynamically assigned by the interpreter. We can explicitly define the type, too. This involves using the functions for a particular type of data. The following code explains this phenomenon by using two functions, `int()` and `float()`, which convert the given input (inside the parentheses) to integers and floating point numbers respectively.

```
1  >>> int (2)
2  2
3  >>> int (2.0)
4  2
5  >>> float (2)
6  2 . 0
7  >>> float (2.0)
8  2.0
```

2.6 Modules

Python Multiverse has expanded with thousands of modules and, being open source, most of them are readily available. Modules are collections of Python programs for accomplishing specific tasks. For example, `numpy` has various facilities for numerical computation. It was further expanded into `scipy` for scientific computation in general. `matplotlib` is an acronym for *mathematical plotting library*, which has rich features for plotting a variety of publication-ready graphs. `pandas` is the library for data analysis, `scikit-learn` for machine learning, `scikit-image` for image processing, and `sympy` for symbolic computing.

To use a module, it must be installed in the machine first. Installation includes downloading the files into an appropriate folder or directory, unzipping them, and defining proper paths. There is an easier way for Ubuntu users where the short command-line-based program `pip` performs these tasks seamlessly.

Running the simple command

```
sudo apt-get install python3-pip
```

installs the program `pip` first. It can then be used to install a package, say `numpy`, by simply issuing the command

```
sudo pip3 install numpy
```

Replacing the name of the package with the desired package will do the trick of installing the packages hassle-free. Also, modules can be upgraded to the latest version by typing

```
sudo pip install [package_name] --upgrade
```

Installing `scipy stack` [4] is very useful for our purposes because it installs a variety of programs that we will be using. It can be installed by issuing the following command on an Ubuntu terminal:

```
$ sudo apt-get install python-numpy python-scipy
python-matplotlib ipython ipython-notebook
python-pandas python-sympy python-nose
```

This command in a single line installs the following:

- `numpy` (numerical Python)

- `scipy` (scientific Python)

- `matplotlib` (mathematical plotting library)

- `ipython` (an interactive environment for the Python interpreter)

- `ipython-notebook` (a web-based interactive environment for `ipython`)

- `pandas` (used for statistical computations)

- `sympy` (used for performing symbolic computation)

- `nose` (used for testing)

Anaconda IDE comes pre-installed with all the modules this book requires. Any extra modules can be installed in a similar manner using a conda terminal. To open a conda terminal, first click `Environments` and then the arrow button next to `root` (Figure 2-6). You can choose the option `Open Terminal` to open a Linux-like terminal in any operating system.
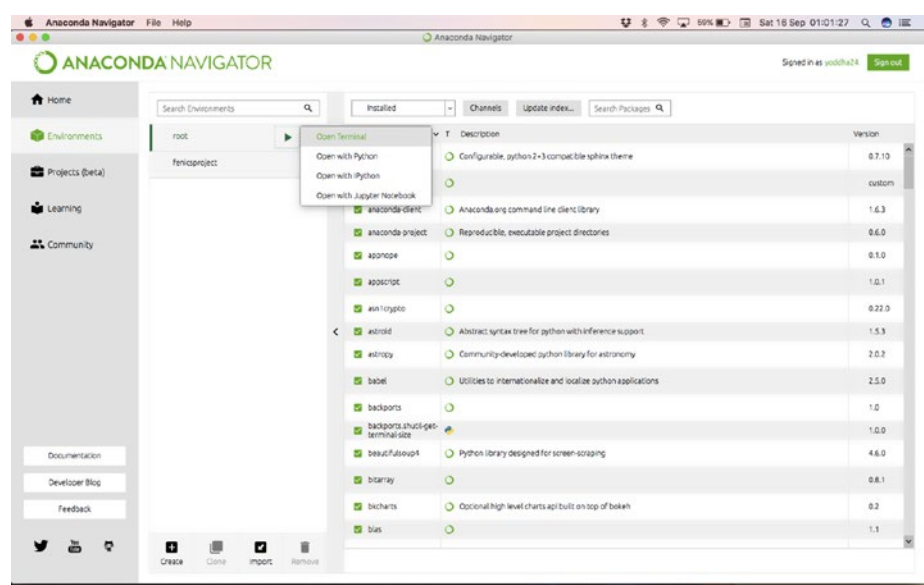
Figure 2-6. *Conda terminal*

The Environment option also presents the list of installed modules. You can uninstall and upgrade a module to the required version by choosing the desired module and clicking Apply at the bottom at the bottom of the screen.

2.6.1 Using a Module

To use a particular module and its functionalities, first import it inside the workspace using the command import. For example:

```
import math
```

This command imports the module named `math` and all its objects are now available for usage. The procedure to use them is as follows:

1. `import math`

    - We need to mention the name of the module before using any function defined within itself. For example, if we wish to use the `sqrt()` function, we need to write `math.sqrt(2)`, which will calculate $\sqrt{2}$ using the `math` module.

2. `from math import *`

    - `"*"` signifies everything (in the language of regular expressions). Hence, the statement means that we can use all the functions without the need to write the name of the module separated with a `"."` (dot operator) before the name of the function.

3. `from math import sqrt`

    - This statement just imports the `sqrt()` function being used. Using this style of Python programming is essential when you have limited memory and are energy-conscious because you don't need to import unnecessary objects. You still need to write the name of module separated with a `"."` (dot operator) before the name of the function.

2.7 Python Environment

A professional way of working on a Python project is to establish a dedicated Python environment for the particular project. This has several advantages. The environment is isolated to the system editions, updates, and upgrades. In other words, a separate set of a software dependency's

repository (Python packages) can be maintained, which avoids problems if the dependency's upgrades affect the performance of code. (A beginner can skip this section for now and then come back after gaining more practice with Python.)

Anaconda 2.4 presents the option of defining an environment as a clickable tab in the left pane of the main window where you can create and maintain a virtual environment graphically. You can do the same using a terminal. `virtualenv` is a command-line-based tool that can create a dedicated and isolated Python virtual environment. `virtualenv` creates a folder that contains all the necessary executables to use the packages that a Python project would need.

2.7.1 Installing virtualenv

To install `virtualenv`, you need a command-line tool named `pip` (explained in detail in Section **??**). On Ubuntu Linux-based machines, issue the following command to install `virtualenv`:

```
1  $pip install virtualenv
```

Suppose you have created a folder named `my_project_folder`. To create a virtual environment for a project named *proj1*, you have to issue the following commands at a Linux terminal:

```
1  $cd my_project_folder
2  $virtualenv proj1
```

The second command will create a folder in the current directory. This folder will contain the Python executable files and a copy of the `pip` library, which you can use to install other packages.

2.7.2 Activating virtualenv

To use the virtual environment that has been created, it must first be activated. The following command must be issued on a Linux terminal:

```
1   $source proj1/bin/activate
```

If there haven't been any errors, the command prompt will be preceded by the name of the virtual environment. This indicates that it is active now for further usage. Within a virtual environment, packages can be installed in the usual manner as installed on a system in general.

You can generate a list of dependencies with their installed version numbers by issuing the command within the environment:

```
1   $pip freeze > requirements.txt
```

This command creates a file named requirements.txt that has a list of dependencies along with their version numbers. When the project is shared and new users wish to install the package, they can use this file for installation by issuing the following command:

```
1 $pip install —r requirements.txt
```

2.7.3 Deactivating the Virtual Environment

Since a developer may need to work on a number of projects at the same time, it is helpful for each one to have a separate virtual environment because the developer can work on only one virtual environment at a time. When developers need to switch to some other task outside the virtual environment, they must deactivate the present virtual environment by issuing the following command:

```
1   $deactivate
```

The command puts the developer back to the system's default Python interpreter with all its installed libraries. In case developers need to remove the virtual environment, they can remove its folder by issuing the following command:

```
1  $rm – rf proj1
```

2.8 Summary

Python has an extremely flat learning curve owing to its interpretive language, which can be used to insert instructions line-by-line and run them subsequently. This methods avoids compilation and subsequent errors that are major stumbling blocks for a beginner who has limited knowledge of the inner workings of the programming language. Being open source in nature, the Python interpreter can be downloaded for free from the Internet. Anaconda IDE presents a good integrated development environment for Python developers, and we recommend that you use it for practicing the codes in this book.

2.9 Bibliography

[1] https://www.python.org/.

[2] https://www.python.org/downloads/.

[3] https://www.anaconda.com/download/.

[4] https://www.scipy.org/stackspec.html.

IPython

3.1 Introduction

The biggest challenge the creators of a programming language face is attracting developers and making them want to learn the language. Usually developers use documentation or help books to gain insights into the language, but an interactive environment allows developers to experiment with commands and programs as they learn. This environment has already been provided by Python's interactive shell.
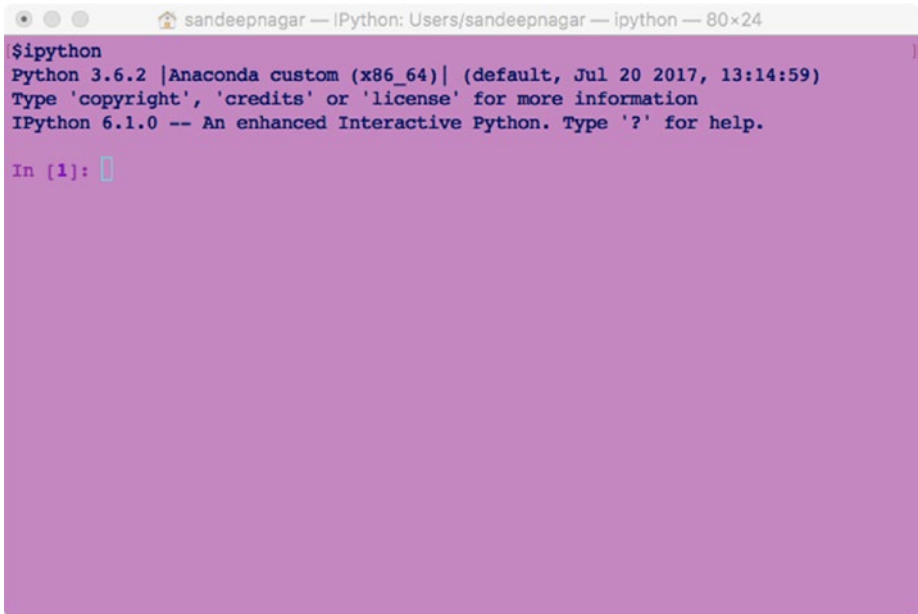
Another option for providing the same environment is IPython [1]. IPython was inspired [2] by the notebook environments of Maple and Mathematica. It launched in 2001 and immediately started attracting developers since it conceptualized a simple and intuitive environment to execute Python code with the following features:

- An interactive shell for the Python interpreter

- A browser-based notebook with support for code, text, mathematical expressions, inline plots, and other media such as `ipython-notebook`

- Support for interactive data visualization and use of GUI toolkits

- Flexible, embeddable interpreters to load into a project

- Tools for parallel computing

Since 2001, the IPython environment has grown tremendously and the popular web-browser-based interactive environment has even changed its name to `Jupyter Notebook` [3] to highlight the fact that, in addition to Python, it can run other kernels. At the time of writing, it supports 40 programming languages including Julia, Python, Scala, R, C, and C++. Jupyter needs a native installation of the Python interpreter to work, so users must install the Python interpreter as illustrated in Chapter 2 and then they can work within the IPython environment with Python as one of its kernel.

3.2 Installing IPython

If you are working with Anaconda IDE (works on all operating systems including Windows, Linux, and Mac OS X), IPython is already pre-installed. Figure 3-1 shows the screenshot of a Spyder IDE session. It has an IPython shell running in the lowermost right corner. You can type Python expressions just like a Python shell here and expect it to get evalauted.

Figure 3-1. *IPython session*

If you prefer to work on an Ubuntu Linux command shell, you can install IPython by issuing the following command on the terminal:

```
1  $pip3 install ipython
```

After installation, you can start an IPython session by issuing the following command on the terminal:

```
1  $ipython
```

Instead of calling the interpreter by typing python, you call the application ipython, which starts an IPython session in which the Python interpreter is used as the kernel.

You can exit the IPython session by typing exit at its prompt. At the Python prompt, you had to define exit as a function and type exit(), but here exit executes the exit function. As seen in Figure 3-1, the IPython prompt shows In[1] at the start of a session. This signifies that

the IPython prompt running the IPython REPL is waiting for the user to type the first input. When you type an input and execute the expression by pressing the Enter key, you obtain output with the prompt Out[1], signifying it is output of the first Python expression at In[1]. The numbers within parentheses keep increasing by one, which makes it useful while teaching Python as well as when tracking workflow during a session. These numbers get reset once a session has restarted.

The IPython shell also supports Unix/Linux system commands. Some sets of useful commands are the following:

- pwd: Print working directory

    – Users can print the directory in which they are running the session.

- cd <name1>: Change directory from present to name1

    – If name2 directory is within the present working directory, then the command makes name1 the present working directory.

- rm -r <name_of_directory>: Remove directory <name_of_directory>

    – Users can remove the contents of the directory. If the directory further contains other subdirectories, users have to remove it forcefully by using the flag -rf.

- cd.. and cd

    – The command cd.. enables a user to step one step back in the directory tree structure.
    – The command cd places a user in the home directory.

There is a number of other Unix/Linux commands that can be used at the IPython shell. A special set of commands called *magic functions* [4] can be used to enhance the capabilities of an IPython session. There are two

kinds of magics: line-oriented and cell-oriented. Line magics are prefixed with the % character and work much like OS command-line calls: They get as an argument the rest of the line, where arguments are passed without parentheses or quotes. Cell magics are prefixed with a double %%, and they are functions that get as an argument not only the rest of the line, but also the lines below it in a separate argument. Discussing all magic functions is beyond the scope of this book, so only a few useful magic functions are discussed here:

- `%history`
  - All input history from the current session is displayed.
- `%logstart, logoff, logon, %logstop`
  - `%logstart:` Start logging anywhere in a session.
  - `logoff:` Temporarily stop logging.
  - `logon:` Restart logging.
  - `%logstop:` Fully stop logging and close log file.
- `%lsmagic`
  - List currently available magic functions.
- `%magic`
  - Print information about the magic function system.
- `%run`
  - Run the named file inside IPython as a program.
- `%time`
  - Time execution of a Python statement or expression.

The following IPython shell code will illustrate the usage. It is important to note that comments in Python start with a # sign. Any expression after the # sign is ignored by the interpreter, so informative

comments can be inserted for each line of code. It is good practice to insert useful comments to yourself in case you revisit the code after a long time or if you share the code for a project so that others can understand the logic behind the usage of commands in the given code file:

```
 1  In  [1]: 2+3 # adding two integers
 2  Out [1]: 5
 3
 4  In  [2]: 2.0+3 # Adding a floating point number and an integer
 5  Out [2]: 5.0
 6
 7  In [3]: %history # probing history of commands used in
                         these ssion
 8  2+3
 9  2.0+3
10  %history
11
12  In [4]: %history # probing history again shows history
                        command used at IN [3]
13  2+3
14  2.0+3
15  %history
16  %history
17
18  In  [5]: %lsmagic # listing all magic functions available
19  Out [5]:
20  Available line magics :
21  %alias  %alias_magic    %autocall   %autoindent   %automagic
        %bookmark    %cat    %cd    %clear    %colors    %config
        %cp    %cpaste        %debug    %dhist    %dirs
        %doctestmode    %ed    %edit    %env    %gui
```

```
      %hist   %history   %killbgscripts   %ldir   %less   %lf
      %lk    %ll          %load   %load_ext   %loadpy
      %logoff   %logon   %logstart
      %logstate   %logstop   %ls   %lsmagic   %lx   %macro
      %magic   %man          %matplotlib   %mkdir   %more
        %mv   %notebook   %page   %paste
      %pastebin   %pdb   %pdef   %pdoc   %pfile   %pinfo
      %pinfo2   %popd          %pprint   %precision   %profile
      %prun   %psearch   %psource
      %pushd   %pwd   %pycat   %pylab   %quickref   %recall
      %rehashx   %reload_ext   %rep   %rerun   %reset
      %reset_selective   %rm   %rmdir   %run   %save
      %sc   %setenv   %store   %sx   %system   %tb
       %time   %timeit   %unalias   %unload_ext   %who
       %who_ls   %whos          %xdel   %xmode
22
23 Available cell magics :
24 %%!   %%HTML   %%SVG   %%bash   %%capture   %%debug   %%file
   %%html   %%javascript   %%js   %%latex   %%markdown
   %%perl   %%prun
     %%pypy   %%python   %%python2   %%python3   %%ruby
     %%script   %%sh   %%svg   %%sx   %%system   %%time
     %%timeit   %%writefile
25
26 Automagic is ON, #prefix IS NOT needed for line magics.
27
28 In [6]: %time print ("Hello World !") #timing execution of
                                      a print statemnet
29 Hello World!
30 CPU times: user 110 micros, sys: 52 micros, total:
31 162 micros Wall time : 149 micros
32
```

```
33  In [7]: %history # Probing history again showing other used
                       expressions
34  2+3
35  2.0+3
36  %history
37  %history
38  %lsmagic
39  %time print ("Hello World !")
40  %history
41
42  In [8]:
```

It has been noted that the Windows OS version of the Python interpreter does not list `cputime` when `%time print("Hello World!")`.

3.3 IPython Notebooks

Just as IPython sessions are started within terminal sessions, IPython Notebook sessions are started within web-browser sessions. As a result, users only need a web browser to work with IPython Notebooks. All commands explained in the preceding section work in a similar fashion here too except for the fact that users need to press the `Shift+Enter` key combination for execution. Ipython Notebooks have been renamed as Jupyter Notebooks since, in addition to Python, 40 programming languages are now supported. Users first need to install a Jupyter Notebook before working.

3.3.1 Installing a Jupyter Notebook

Anaconda IDE has pre-installed the Jupyter Notebook. Figure 3-2 shows the Jupyter Notebook as one of the first options in the central panel. Clicking `Launch` will open a window in the installed browser. Most of the

browsers are supported for this case. This book uses Google Chrome (Version 60.0.3112.113 (Official Build) (64-bit)) for the purpose of testing the code used henceforth.
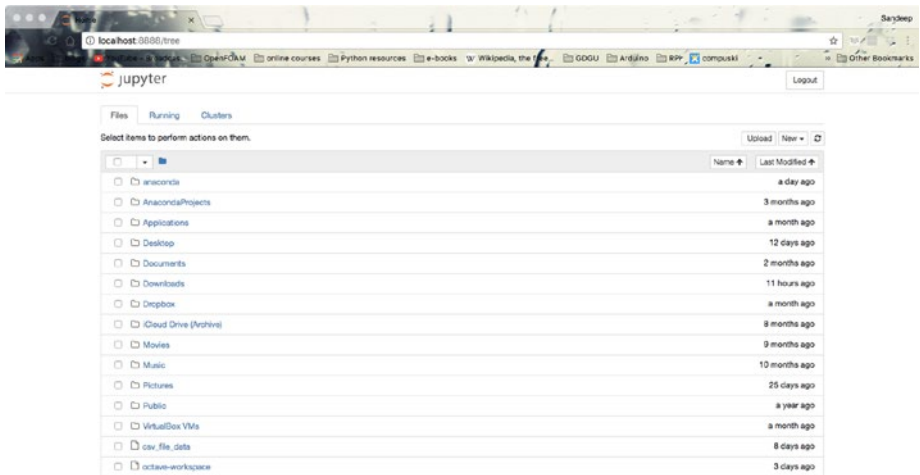
Figure 3-2. *IPython Notebook session*

The web browser window will list the directories and files in the home folder. Users can navigate to a target directory and start a Jupyter session for Python 3. They can click the New button and choose a kernel (Python3 in this case). This will open a screen similar to Figure 3-3.
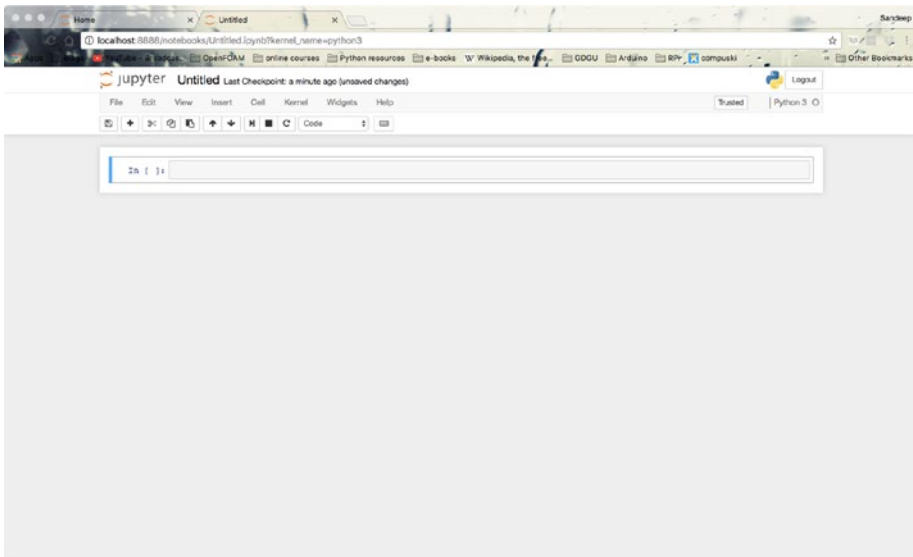
Figure 3-3. *Jupyter Notebook session*

Following are various features of this Jupyter Notebook session:

- Naming a notebook

  - A new notebook is titled Untitled by default.
    Clicking this title will open a graphical view to enter a new
    name, which will be updated accordingly.

- Cells

  - A Jupyter Notebook is composed of cells that are
    executed by pressing the Shift+Enter key
    combination.

  - There are code kinds of cells, as shown in Figure 3-3.

    - Code

      - Users can write Python code in the cell assigned
        to Code type and execute single or multiline code
        by pressing the Shift+Enter key combination.

- – Users can also *run* a cell (that is, execute a cell) by pressing the `Cell` ⇒ `Run Cells` button in the main menu bar.

- – Special display formats also exist for code cells, but that discussion is out of the scope of this book. Users are advised to refer to the main documentation for learning about various display formats [5].

- Markdown [6]

  - – Cells that are assigned `Markdown` type can be filled with markdown code to be executed for formatted printing.

  - – Users can use LaTeX commands here to define mathematical equations and special symbols.

  - – Users can define a heading by putting a single # for the main heading and multiple symbols for successive subheadings.

- Raw NBConvert

  - – The notebook authoring environment does not render raw cells.

  - – They can have different formats. This information is stored in the notebook metadata.

  - – A raw cell is defined as content that should be included unmodified in `nbconvert` output.

  - – By default (if no cell format is selected), the cell content is included (without any conversion) in both the HTML and LaTeX output.

- Heading

    - A cell type `Heading` can be used to create a heading for a separate section of the document.

    - An alternative is to use the cell type `Markdown` and then use the # symbol to make one or more headings.

Users learn best to use the Jupyter Notebook by trying it. A sample notebook is shown in Figure 3-4.
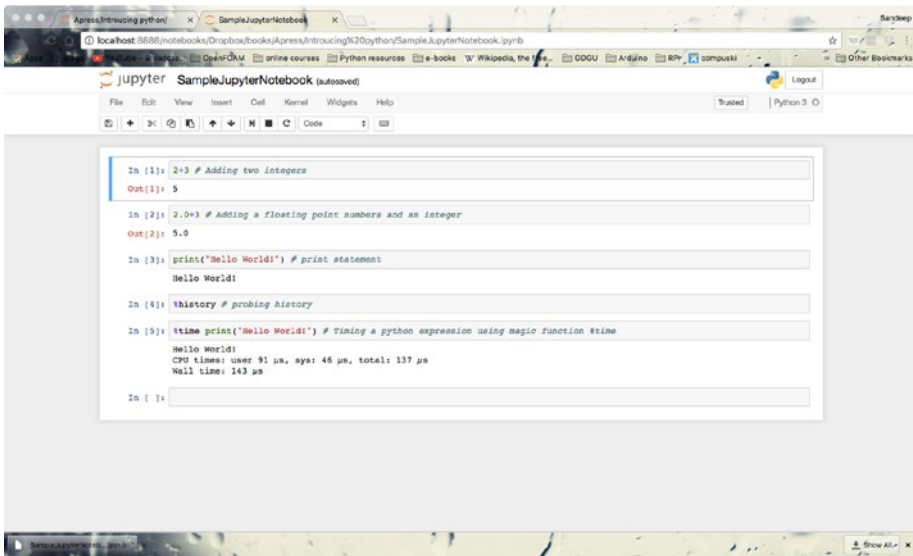
Figure 3-4. *A sample Jupyter Notebook session*

3.4 Saving a Jupyter Notebook

Users can save a Jupyter notebook as a `.ipnyb` file and open it within the Jupyter Notebook environment on any computer. This makes it easier to teach and learn Python. A developer can illustrate the usage of Python in a notebook by dividing it into sections and giving the sections headings and subheadings, writing codes in code cell types, and writing explanatory

details in markdown cell types. The markdown cell types enable writing mathematical equations using LaTeX commands.

Users can also save the Jupyter Notebook as a `.py` file (Python program to be executed at the Python interpreter). Here all cells except the code cells are converted as comments by adding the symbol # at the start.

Jupyter Notebooks can be saved in an HTML format to be used when embedding them on a web site. This is useful for online course development. They can also be saved as LaTeX files to be built in a particular format or directly as a PDF document.

The saved files can then be shared with people who wish to learn a particular topic. Users can simply run the notebook within the Jupyter Notebook environment. This feature benefits classroom teaching quite a lot. The teacher and students can execute their Jupyter Notebooks cell-by-cell and understand the concepts in an easier fashion. Otherwise, a teacher would have to switch between a presentation file and the Python interpreter for a Python teaching session. Users can also define a test as well as assignments using this facility.

3.5 Online Jupyter Environment

The main web site of Jupyter Notebooks [7] provides an online environment on a remote server for all the facilities that are provided by a local installation, as mentioned in the preceding sections. Here, users are working on a remote computer via the Internet. Each user is provided a limited space for working. This space can be shared with a team and users can work collaboratively. The screenshot of a Jupyter session is shown in Figure 3-5.
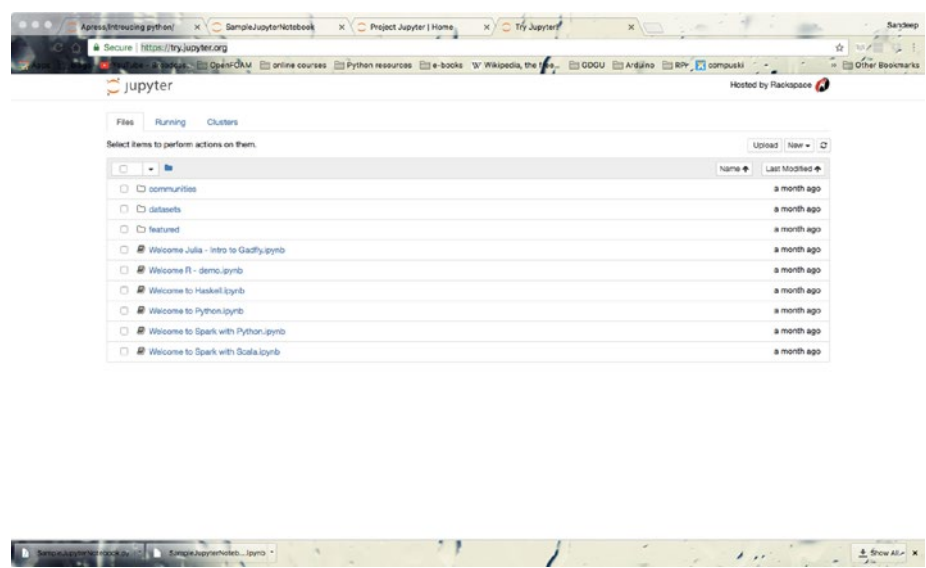
Figure 3-5. *A sample online Jupyter Notebook session*

The online session has a sample notebook for a list of languages. Users can use the sample Python Notebook to learn the basics. Users can create a directory and then use it to store all the Jupyter Notebooks in an organized fashion. These notebooks can be downloaded on a local machine and can be used in a local Jupyter Notebook session. Users can perform the opposite action too. In other words, they can load a local Jupyter Notebook on an online session using the Upload button in the top-left menu.

3.6 Summary

In this chapter, we have illustrated working in an IPython and Jupyter Notebook environment. Jupyter Notebook is increasingly being used by developers to propagate knowledge to a wider audience since users can package the code and illustrative descriptions in one single file. Thousands of Jupyter Notebooks exist on various web sites, enabling the number of Python users to multiply at an exponential rate. We suggest that users of

this book should also make a separate Jupyter Notebook for each chapter for their own understanding and release it on the Internet for the benefit of others.

3.7 Bibliography

[1] F. Pérez and B. E. Granger, "IPython: A system for interactive scientific computing," *Computing in Science and Engineering*, vol. 9, pp. 21–29, May 2007.

[2] http://blog.fperez.org/2012/01/ipython-notebook-historical.html.

[3] https://jupyter.org/.

[4] https://ipython.org/ipython-doc/3/interactive/magics.html.

[5] https://nbsphinx.readthedocs.io/en/0.2.3/code-cells.html.

[6] http://markdown-guide.readthedocs.io/en/latest/basics.html.

[7] https://try.jupyter.org/.

CHAPTER 4

Data Types

4.1 Introduction

The best way to take full advantage of this book is to begin working on an IPython prompt by feeding it code and observing the output. You can use Anaconda IDE to avoid the need to install the required packages for the book. You can use Spyder IDE during an IPython session as well as editor to write multiline files. We suggest that you write detailed descriptive comments wherever needed to enhance understanding of workflow of the given code. This scheme works uniformly across all platforms. In this chapter, we will discuss the various kinds of data in Python and their categorization into several data types.

Modern computers distinguish data in several types. Data can be numbers, characters, strings (a group of characters), and so on. In Python, you can define new data types as and when required. There are some built-in data types for handling numbers and characters. Different data types occupy different amounts of memory. It is important to understand your needs and choose a data type accordingly. Data types also determine the accuracy of the answer. We will discuss the various built-in data types in Python in the sections that follow.

4.2 Logical

This type of data stores boolean values `True` or `False` and can be operated by boolean operators such as `AND` and `OR`. Most programming languages use the values 1 or 0 for boolean values, but Python differs in this approach. Let's explore their usage with some examples.

Operator	Python Function	Symbolic Function
AND	and()	&
OR	or()	|
XOR	xor()	^
NOT	not()	

Table 4-1. *Truth Table for OR Operator*

OR	True	False
True	True	True
False	True	False

Table 4-2. *Truth Table for AND Operator*

AND	True	False
True	True	False
False	False	False

Table 4-3. *Truth Table for XOR Operator*

XOR	True	False
True	False	True
False	True	False

Table 4-4. *Truth Table for NOT Operator*

NOT	
True	False
False	True

```
 1  >>> True
 2  True
 3  >>> False
 4  False
 5  >>> not True
 6  False
 7  >>> not False
 8  True
 9  >>> True and False
10  False
11  >>> True or False
12  True
13  >>> True ^ False
14  True
15  >>> True ^ False
16  True
```

Complex logical statements can be evaluated too. For example:

```
1  >>> (True and True) or (not False)
2  True
3  >>> (True or True) or (True and False)
4  True
5  >>> not ((True or True) or (True and False))
6  False
```

It is important to see that the usage of parentheses enables the user to club together a logical expression in a meaningful way.

Logical operations are resultant of comparison operations.

For example:

```
1  >>> 2>3 # greater than operator
2  False
3  >>> 2==2 # equality operator
4  True
5  >>> 2==2.0 # equality operator compares the vakue and not
   the type
6  True
7  >>> 2==3 # equality operator
8  False
9  >>> (2+3)>(2-3) # complex expression evaluated logically
```

4.3 Numeric

Numeric: There are four types of numeric data types:

- *int*: Integers

- *float*: Floating point numbers

- *complex*: Complex numbers

4.3.1 Integer

Python has arbitrary precision for `float`, `long`, and `complex  numbers`. As a result, the limit of the length of these numbers is subject to the availability of memory. The positive side of this architecture is that users are not limited to a range of numbers, but they must always ensure that sufficient memory is available during the calculation to avoid erroneous results. Python 2 limits the size of `int` to bytes (to the same size as in the

C programming language), whereas Python 3 has merged `int` and `long` as `int`. On a 32-bit system, Python 3 stores `int` as 32 bits. On a 64-bit computer, it stores `int` as a 64-bit long number.

The built-in function `type()` presents the data type as follows:

```
1  >>> type(-1)
2  <class 'int'>
3
4  >>> type(1)
5  <class 'int'>
```

As seen in these examples, the Python interpreter allocates data type dynamically. In other words, it allocates the data type to the input based on its definition. This is quite convenient for a programmer. A 1 automatically becomes an integer and a 1.0 becomes a floating point number.

4.3.2 Floating Point Numbers

In computing, floating point notation is a scheme of representing an approximation of a mathematical real number. A real number is usually written with a decimal point. For example, 2 is an integer, whereas 2.0 is a real number. These two numbers are quite different to a computer. While 2 will be stored as `int` type, 2.0 will be stored as `float` type:

```
1  >>> type(2)
2  <class 'int'>
3  >>> type(2.0)
4  <class 'float'>
```

The mapping of a real number to a computer's storage system is a formulaic representation (called *floating point representation*) [1]. Real numbers are expressed in three parts: significand, base, and exponent.

For example, the value of π is 3.1415926535897... . Let's suppose that we have only four significant digits for a particular calculation, so the value can be rewritten as 3.1415. Now this number is represented as 31415×10^{-4} where 31415 is called *significand*, 10 is called *base*, and -4 is called *exponent*.

While assigning a number to the significand, the information about the number of significant digits is used. The significant figures of a number are digits that carry meaning contributing to its measurement resolution. In this case, we assumed only four significant digits depending on the requirements of calculations/measurements. The term *floating point* refers to the fact that a number's *radix point* (decimal point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated as the exponent component; thus, the floating point representation can be thought of as a kind of scientific notation.

4.3.3 How to Store a Floating Point Number

Computers can store numbers as floating point objects. A floating point object stores a number as follows:

$$\pm d_1 d_2 \ldots d_s \times \beta^e \tag{4.1}$$

where $d_i = 0, 1, 2 \ldots \beta - 1$ but $d_1 j = 0$ and $m \le e \le M$ where $m \in I^-$ and $M \in I^+$.

Three parts of a floating point number are the following:

- Sign ($\pm$)

- Significand (Mantisa) ($d_1 d_2 \ldots d_s$)

- Exponent (β)

Each part of a floating point number is stored at different memory locations and occupies a specified number of bits. How many bits are defined to which part? This question has been answered by IEEE standards

known as IEEE754 [1]. First, let's look at the concept of precision of a number representation:

1. **Single precision:**

   - Occupies 4 bytes = 32 bits.

2. **Double precision:**

   - Occupies 8 bytes = 64 bits.

3. **Extended double precision:**

   - Occupies 80 bits.

4. **Quadruple precision:**

   - Occupies 16 bytes = 128 bits.

This scheme can trade off between range and precision. While integers have unlimited precision, floating point numbers are usually implemented using double precision, as previously explained. The information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in sys.float_info:

```
1  >>>import sys # importing sys module
2  >>> sys.float_info #using float_into function of sys
3   sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
    max_10_exp=308,
        min=2.2250738585072014e-308, min_exp=-1021,
        min_10_exp=-307, dig=15,
        mant_dig=53, epsilon=2.220446049250313e-16, radix=2,
        rounds=1)
4  >>> sys.int_info # using int_info function of sys
5  sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

32-bit Windows OS outputs sys.int_info(bits_per_digit=15, sizeof_digit=2) when the sys.int_info command is issued at a Python prompt.

The issue with floating-point-number-based arithmetic is that the answer is an approximation of a real number since real numbers are defined with 10 as their base, whereas computers work with numbers where 2 is used as the base. For example, 0.123 is defined as

$$0.123 \rightarrow \frac{1}{10^1} + \frac{2}{10^2} + \frac{3}{10^3}$$

in the number system with base 10. Conversely, in a number system with base 2, it is represented as

$$0.123 \rightarrow \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3}$$

This calculation shows that $0.123_2 = 0.135_{10}$. If you use more bits to store the value, you get a better approximation of the real number, but you are always limited to the use of *approximated* values instead of *real* values. As `sys.float_info` informed, the biggest floating point number is $max = 1.7976931348623157e + 308$ ($1.7976931348623157 \times 10^{308}$).

To employ even more precision, you can use the `decimal` module that has the function `Decimal()`, which returns the number as stored by the computer:

```
1  >>>from decimal import Decimal
2  >>>Decimal(0.123)
3  Decimal('0.1229999999999999982236431605997
   495353221893310546875')
4  >>>Decimal(1.2345)
5  Decimal('1.23449999999999993072208326339023187
   75653839111328125')
6  >>>type(Decimal(0.123))
7  <class 'decimal.Decimal'>
```

As you can see, 0.123 is stored as 0.12299999999999999982236431605 99749535322189331054687 5, which is still an approximation of the real number 0.123. For most cases, the error is insignificant and you can ignore the fact that calculations using digital computers (running on binary system of number) have introduced some error. But, for some cases, this error is significant and you need to take proper measures to calculate this error and counter it.

4.3.4 Complex Numbers

Complex numbers are extensively used in science and engineering studies. The imaginary part of a complex number has important information about the phase of a signal. Python enables the use of complex numbers by creating an object called `complex`. The imaginary part is signified by adding the symbol j, which signifies the iota $(=\sqrt{-1}\,)$. Consequently, you can define a complex number $2+3i$ as 2+3j in Python. Another way of defining a complex number is to use the built-in function `compelx()`, which is fed two arguments: the first argument is the real part and the second argument is the complex part of the complex number. The following code will give some more information about the usage:

```
1  >>> 2+3j
2  (2+3j)
3  >>> type(2+3j)
4  <class 'complex'>
5  >>> complex(2,3)
6  (2+3j)
7  >>> type(complex(2,3))
8  <class 'complex'>
```

4.4 Sequences

Language symbols require storage on a computer and, hence, must be defined as an object. This object is known as a *character*. Every individual symbol of a language that appears on a computer screen and printed papers is considered a character in programming languages. This includes ASCII and Unicode characters. Examples of characters include letters, numeral digits, whitespaces, punctuation marks, exclamation marks, and so on. In general, all keys on a keyboard produce characters.

For the purpose of communicating between computing devices, characters are encoded in well-defined, internationally accepted formats (such as ASCII and UTF-8) that assign each character to a string of binary numbers. Two examples of usual encodings are ASCII and the UTF-8 encoding for Unicode. All programming languages must be able to decode and handle internationally accepted characters. Python also deals with characters using data type `strings`, `lists`, and `tuples`.

4.4.1 Strings

A string is merely a sequence of characters. Lowercase and uppercase characters have different encoding; thus, strings are case-sensitive:

```
1  >>> type('a')
2  <class 'str'>
3  >>> type('abba')
4  <class 'str'>
5  >>> type("a")
6  <class 'str'>
7  >>> type("abba")
8  <class 'str'>
```

Here, we have defined a string of one and four characters, respectively. It is important to remember that whitespace is also a character. As such, `Hello world!` has 12 characters, namely `h,e,l,l,o, ,w,o,r,l,d,!`. While defining strings, we enclose the characters with " or "". When a string has to span multiple lines, three sets of quotation marks are used:

```
1  >>> print("""Python can also print multiline strings which can
       span many lines instead of being printed on just one
       line""")
2  Python can also print multiline strings which can span many
   lines instead of
       being printed on just one line
```

You can use special characters to print the characters to define spaces and to print characters on a new line. This can be done using \t and \n characters, which print a tab and a new-line character, respectively:

```
1  >>> print ("' Python can also print multiline strings having
   tabs like
       \t and also prints in \n next line"')
2  Python can also print multiline strings having tabs like and
   also prints in
3      next line
```

Notice the tab being printed as four whitespaces at the place of \t and the string part after \n is printed on a new line since \n is a new-line character.

But why should we have three ways to define a string, namely using single quotes, double quotes, and triple quotes? Single quotes can be used to print strings having double quotes as characters. Double quotes can be used to print single quotes as characters. Triple quotes can be used to print

both double and single quotes as characters. This behavior can be studied
in the following code:

```
1  >>> print("'Mahatma Gandhi's words, "An eye for an eye makes
   the whole world blind", stand true even today"')
2  Mahatma Gandhi's words, "An eye for an eye makes the whole
   world blind", stand true even today
```

4.4.2 Lists and Tuples

A list is an ordered set of objects, irrespective of its data type. It can be
defined by enclosing a set of values in square brackets. For example:

```
1  >>> [1,2.0,'a',"Alpha"]
2  [1, 2.0,'a','Alpha']
3  >>> type([1,2.0,'a ',"Alpha"])
4  <class 'list'>
```

The first element of a list in Python code is an integer, the second is a
floating point number, the third is a single-character string, and the fourth
is a multi-character string.

Tuples are defined using parentheses () instead of brackets []. For
example:

```
1  >>> (1,2.0,'a',"Alpha")
2  (1,2.0,'a','Alpha')
3  >>> type((1,2.0,'a',"Alpha"))
4  <class 'tuple'>
```

In addition to being defined using different sets of brackets, the major
difference between lists and tuples is that tuples are immutable lists; their
elements, once defined, cannot be altered. Elements of a list can be altered

using their indices. More information about how this is done is discussed in Chapter 5, where operations on lists have been defined. In scientific computing, universal constants can be defined as a tuple and then can be accessed in a program using its index.

4.5 Sets and Frozensets

The set data type is the implementation of a mathematical set. It is an *unordered* collection of objects. Unlike sequence objects, such as `list` and `tuple`, where elements are ordered, sets do not have such requirements. However, sets do not permit duplicity in the occurrence of an element, that is, an element wither exist 0 or 1 times. This is not the case with `list` and `tuple` objects.

A set is defined using the `set()` function, which is supplied a `list` as its input argument. For example:

```
1  >>> set(['h','e','l','l','o',1,2.0,3+4j])
2  {1, 2.0, 'h', 'o', (3+4j), 'l', 'e'}
3  >>> type(set(['h','e','l','l','o',1,2.0,3+4j]))
4  <class 'set'>
5  >>> frozenset(['h','e','l','l','o',1,2.0,3+4j])
6  frozenset({1, 2.0, 'h', 'o', (3+4j), 'l', 'e'})
7  >>> type(frozenset(['h','e','l','l','o',1,2.0,3+4j]))
8  <class 'frozenset'>
```

Please note that since `l` occurred two times while defining the set, it was given only one membership. Set operations are discussed in detail in subsequent chapters. Just like a `tuple` object is immutable, a `frozenset` is also an immutable object and similar to a `set` object.

4.6 Mappings

Mapping is a scheme of defining data where each element is identified with a key called "hash tag." The element can be accessed by referring to the key. One of the data types in this category is a *dictionary*.

A dictionary is an *unordered* pair of values associated with keys. These values are accessed with keys instead of an index. These keys have to be hashable like integers, floating point numbers, strings, tuples, and frozensets. Lists, dictionaries, and sets other than frozensets are not hashable. Following is an example of a dictionary:

```
1  >>> {'a':1, 'b':10}
2  {'a':1, 'b':10}
3  >>> type({'a':1, 'b':10})
4  <class 'dict'>
```

In this example, we created a dictionary containing two characters, a and b, identified by two keys, 1 and 10. A variety of operators can operate on dictionaries as discussed in subsequent chapters.

4.7 Null Objects

None is a null object. Null objects in other programming languages including C, Java, and PhP are given by the keyword null, but in Python it is denoted by the keyword None. It refers to nonfunctionality (that is, no behavior for the object with which it is associated.) When it is issued at the Python prompt, nothing happens:

```
1  >>> None
2  >>> type(None)
3  <class 'NoneType'>
```

4.8 Summary

Object-oriented programming uses the fact that all computing entities are merely objects that *interact* with each other as per their defined behavior. We have discussed some built-in data types in this chapter. Some data types are defined inside the modules. Users can define their data types and define their properties. Before going to these advanced topics, it will be useful to know how operators operate on various kinds of data. This will be the subject of Chapter 5.

4.9 Bibliography

[1]   "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

CHAPTER 5

Operators

5.1. Introduction

Operators work in a similar fashion to mathematical functions. They provide a relationship between two different domains. For example, the multiplication operator makes an ordered pair of operands (data on which the operator works) and produces another data point. This can be done to any number of data points. In this way, the operator transforms data from the domain of operands to the domain of results.

The symbols for basic operators are given in Table 5-1.

Table 5-1. *Symbols for Basic Operators*

Mathematical Operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	\
Modulus	%
Exponentiation	**

These operators can be combined in a complex manner to perform an arithmetic operation. Depending on the data type, they define their functionality. For example, on numeric data, + performs numeric

addition, whereas on a string, it will perform concatenation. Similarly, the multiplication operator, when applied on strings, concatenates the string those many number of times. The division, modulus, and exponentiation operators do not have defined behavior for strings. This behavior can be verified in the following Python code:

```
 1  >>> 2+3
 2  5
 3  >>> 'a' + 'b'
 4  'ab'
 5  >>> ""hello" + " " + "world" + "!"
 6  'hello world !
 7  >>> 'a'*3
 8  'aaa'
 9  >>> 'a'/3
10  Traceback (most recent call last):
11  File "<stdin>", line 1, in <module>
12  TypeError: unsupported operand type(s) for /: 'str' and
    'int'
13  >>> 'a'%3
14  Traceback (most recent call last):
15  File "<stdin>", line 1, in <module>
16  TypeError: not all arguments converted during string
    formatting
17  >>> 'a'**3
18  Traceback (most recent call last):
19  File "<stdin>", line 1, in <module>
20  TypeError: unsupported operand type(s) for ** or pow():
    'str and 'int'
```

There are a variety of operators that can operate on data types, as discussed in Chapter 4.

The following section will discuss a variety of built-in operators including arithmetic and logical/boolean. Please note that the field of operators is not limited to the discussion in this chapter. Modules define new data types for which new operators are defined. We will confine our discussion to built-in basic operators. We will illustrate the usage of these built-in operators and their usage in defining a mathematical calculation.

5.2. Concept of Variables

When an operator acts on data, its values can change during the course of computation. This can happen multiple times. To store values temporarily (for example, during the course of computation), we use variables. Variables point to a particular value at a memory location via its address (visualized as a hexadecimal number). This whole setup can be assigned a symbolic string, which is called the variable name.

For example, we can store the value of 0.12 as a variable a and then use it in an equation like

$$1010_2 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times s^0$$

```
1  >>> a=0.12 # assigning value 0.12 to variable named 'a'
2  >>> answer = (a**2) + (10 * a) #performing calculation and
       storing in variable name 'answer'
3  >>> print(a)
4  0.12
5  >>> print(answer)
6  1.2144
7  >>> type(answer) #scanning the type for value stored in
       variable named 'answer'
8  <class 'float'>
9  >>> a=5 # reassigning different value to 'a'
10 >>> print(a) # New value is reflected
```

```
11  5
12  >>> print(answer) # old value is still stored in 'answer'
13  1.2144
14  >>> answer = (a**2) + (10 * a) # calculation using new
        values
15  >>> print(answer) # New value is printed
16  75
```

Here the numerical value 0.12 is stored at a memory location known by the name a. This is called by a subsequent equation defined using a variable name answer, which, when printed, prints the value stored in it. The value stored is a *floating point number*. The type of object can be known by the function type(), which takes the variable name as its argument.

This variable changes the value to the new value 5. However, unless the computation is performed, the variable answer stores the old evaluated value. When the new value a is used, the value of answer is also changed. Python variables don't need to be explicitly defined for their *type*. The types are guessed from the value they point to. It is important to understand that variables are merely pointers to memory locations. When values change, these pointers change to a different address. It is the memory that stores the data. Data can be of any type. In the example previously discussed, the data were a number, but they can be any of any type.

5.2.1 Rules of Naming Variables

Since Python uses certain keywords for its own uses, there are certain names that should not be used for variables. In addition, there are also some internationally accepted rules (usually common for most programming languages):

- They must begin with a letter (a–z, A–B) or underscore (_).

- The rest of the characters can be letters, numbers, or _

- Names are case-sensitive.

- There is no limit to the length of names, but it is advisable to keep them short and meaningful.

- Keywords cannot be used as variable names.

Using the module keyword, you can obtain the list of keywords using the function keyword.kwlist. The code given in keyword.py gives a list of keywords, which cannot be used as variable names. (See Listing 5-1.)

Listing 5-1. Keyword.py

```
1  import keyword
2
3  print ("Python keywords:", keyword.kwlist)
```

The result of running this code is given by:

```
1  Python keywords:  ['False', 'None', 'True', 'and', 'as',
   'assert', 'break', 'class', 'continue', 'def', 'del',
   'elif', 'else', 'except', 'finally', 'for', 'from',
   'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
   'not', 'or', 'pass', 'raise', 'return', 'try', 'while',
   'with', 'yield']
```

A handy way of checking a name among the keywords list exists for an interactive session. To check if a particular variable name is a keyword or not, you can use the function `keyword.iskeyword()`. The name can be input as a string to this function. As shown in the following code, if the input string is a keyword, then the value `True` is returned. Otherwise, `False` is returned:

```
1  >>>import keyword
2  >>>keyword.is keyword ('lambda')
3  True
4  >>>keyword.is keyword ('lamb')
5  False
```

So the name `lambda` is a keyword but the name `lamb` isn't.

5.3. Assignment Operator

The concept of variables uses the symbol `=`, which is not same as "equal to" in mathematics. Instead, it is one of the assignment operators. The symbol = *assigns* the value on the right-hand side to the variable name on the left-hand side. This essentially means that it notes down the memory location's address of the value on the left-hand side and links it to the name of the variable on the left-hand side. This behavior is demonstrated in the following Python code:

```
1  >>> a = 5 # a is assigned the value 5
2  >>> a # value of a is printed by python REPL
3  5
4  >>> print(a) # value of a is printed by print() function
5  5
6  >>> a = 10 # a is re-assigned a new value 10
7  >>> a # values of a is printed and new value is shown as answer
8  10
```

```
9  >>> b = a # a new variable b is assigned the value pointed
       by a
10 >>> b # b now stores value last stored by a i.e 10
11 10
12 >>> a = 15 # a is reassigned a new value 15
13 >>> a # REPL prints newly assigned value for a
14 15
15 >>> b # REPL prints the value of b which was last assigned
16 10
```

The value of a variable can be printed by Python REPL by simply writing the name of the variable and pressing the Enter key. It can also be printed using the print() function where the name of the variable is taken as input (not as a string without quotes). Also, the printed value corresponds to the last assigned value. When b=a was executed, a=10 was used. When a=15 was executed, the value of a changed, but the value of b remained the same. This behavior can be understood in the following way:

- a=10

  - The variable a points to the memory location storing the value 10.

- b=a

  - The variable b also points to the same memory location as a.

- a=15

  - The variable a now points to a different memory location storing the value 15.

  - The variable b still points to the memory location storing the value 10.

  - Both will keep doing so unless they are *reassigned.*

In addition to this simple assignment operator, there are more operators that can perform operations. Table 5-2 provides the information about such assignment operators.

Table 5-2. *Assignment Operators*

Operator	Example
=	v = a+b
+=	v +=a ⟹ v = v + a
-=	v -=a ⟹ v = v - a
/=	v /=a ⟹ v = v / a
//=	v //=a ⟹ v = v // a
*=	v *=a ⟹ v = v * a
**=	v **=a ⟹ v = v ** a
%=	v %=a ⟹ v = v % a

Assignment operators are the most frequently used features on all kinds of programs. Increment and decrement operators like += and -=, respectively, are used extensively where we need to proceed stepwise. The following Python code will make this clear:

```
1  >>> a = 10 # a is assigned the value 10
2  >>> a # REPL prints value of a
3  10
4  >>> b = 0 # b is assigned value 0
5  >>> b # REPl prints the value of b i.e 0
6  0
7  >>> b += a # new value of b is b+a i.e 0+10=10
8  >>> b # REPL prints value of b i.e 10
9  10
```

```
10  >>> b = 1 # b is reasisgned the value 1
11  >>> b += a # Now b is reassigned the balue b+a i.e 1+10=11
12  >>> b # REPL prints newly assigned value of b i.e b
13  11
```

Similarly, we can test the other operators:

```
 1  >>> a = 10 # a is assigned value 10
 2  >>> b = 5 # b is assigned value 5
 3  >>> a # REPL prints value of a
 4  10
 5  >>> b # REPL prints value of b
 6  5
 7  >>> b += a # b is reassigned the value b+a=10+5=15
 8  >>> a # a still points to the value 10
 9  10
10  >>> b # b points to newly reassigned value of 15
11  15
12  >>> b -=a # b is reassigned the value b-a=15-10=5
13  >>> a # a still points to the value 10
14  10
15  >>> b # b points to newly reassigned value of 5
16  5
17  >>> b /=a # b is reassigned the value b/a=0.5 i.e the
        quotient of division operation
18  >>> a # a still points to the value 10
19  10
20  >>> b # b points to newly reassigned value of 0.5
21  0.5
22  >>> b  //=a # b is reassigned the value b/a=0.0 i.e the
        quotient of floor division operation
23  >>> a # a still points to the value 10
24  10
```

```
25  >>> b # b points to newly reassigned value of 0.0
26   0.0
27  >>> a = 10 # a is reassigned the same value of 10
28  >>> b = 5 # b is reassigned a new value of 5
29  >>> a # a still points to the value 10
30  10
31  >>> b # b still points to the value 5
32  5
33  >>> a *= b # a is reassigned the value a*b=10*5=50
34  >>> a # a now points to newly reassigned value i.e 50
35  50
36  >>> b # b still points on value i.e 5
37  5
38  >>> a **= b # a is reassigned the value a**b i.e 50 raised to
        the power 5 i.e 312500000
39  >>> a # a now points to newly assigned value 312500000
40  312500000
41  >>> b # b still points to old value i.e 5
42  5
43  >>> a %= b # a is reassigned the value a%b=312500000%5 i.e
        remainder of 312500000/5=0
44  >>> a # a now pints to newly reassigned value i.e 0
45  0
46  >>> b # b still points to old value i.e 5
47  5
```

We also learned about arithmetic operators +,-,*,**,/,//, and % in the preceding Python code. They will be discussed in detail in Section 5.4. They can be used with different number types, and mathematical calculations can be performed with ease. The only fact that you need to remember is the limits of the maximum number that you can store in a particular data type and if an arithmetic operator allows usage of a particular data type as one of its arguments.

Multiple assignments within the same statement can be done using the = operator as follows:

```
1   >>> a = b = c = 10
2   >>> a
3   10
4   >>> b
5   10
6   >>> c
7   10
8   >>> c = 15
9   >>> a
10   10
11   >>> b
12   10
13   >>> c
14   15
15   >>> a = b = c = 15
16   >>> a
17   15
18   >>> b
19   15
20   >>> c
21   15
```

In the preceding Python code, first c=10 assigns value 10 to the variable name c and then b=c assigns the value stored in c (that is, 10) to the variable name b. Furthermore, a=b assigns the assigned value of b to a. Effectively, all three variables (a, b, and c) point to same memory location storing an integer 10. When c is assigned a new value 15, the variable names a and b keep pointing to the memory location storing the value 10, but c now points to the new memory location for value 15. When multiple assignment for the value 15 is again performed, a, b, and c point to the newly assigned memory location pointing to value 15.

While assigning a value, its data type doesn't need to be explicitly defined. It is judged by the Python interpreter by the data itself. For example, 4.0 will be taken as floating point number, 4 will be taken as integer, and a single character 'a' or a group of characters like 'sandeep' will be taken as a string. This is shown in the following example code:

```
1  >>> a = 4.0; type(a) # a is assigned the value 4.0 and its type
       is found to be floating point number
2  <class 'float'>
3  >>> a = 4; type(a) # a is assigned the value 4 and its type is
       found to be integer
4  <class 'int'>
5  >>> a = 4e10;type(a) # a is assigned the value 4 X 10^10 and its
       type is found to be floating point number
6  <class 'float'>
7  >>> b = 'a'; type(a) # b is assigned value same as in a i.e 4 X
       10^10 and its type is found to be floating point number
8  <class 'float'>
9  >>> b = 'sandeep'; type(b) # b is reassigned a value as string
       and its type is found to be string
10  <class 'str'>
```

It can be noted that another way of defining a floating point numbers is via engineering notation. For instance, 4e10 denotes the engineering notation for the number 4×10^{10}. Also, by using ;, printing of output at REPL can be suppressed The value of the assignment is suppressed to be printed at REPL, but output of the type() function is not suppressed and, hence, it is printed at REPL.

5.4. Arithmetic Operators

Mathematical operators such as +, -, %, /, and // (Table 5-4) work by the same logic as in mathematics. a^b is written as a**b and floor division is denoted by // symbols. Floor division (using symbols //) is when the quotient is rounded off toward $-\infty$ (in other words, toward the nearest integer on the left-hand side of a number line):

```
 1  >>> 4.2 % 2.3 #output is the remainder
 2  1.9000000000000004
 3  >>> 4.2 / 2.3 # output is the quoteint
 4  1.8260869565217392
 5  >>> 4.2 + 2.3 # output is addition
 6  6.5
 7  >>> 4.2 - 2.3 # output is subtraction
 8  1.9000000000000004
 9  >>> 4.2 ** 2 # output is exponentiation
10  17.64
11  >>> 4.2 // 2.3 # output is floor division
12  1.0
```

Some arithmetic operators work on `string` and `list` objects as well. For example, + acts as a concatenation operator on strings; it *joins* the two or more arguments together from head to tail. The same thing happens for a list data type. Also, the * operator simply performs repeated + by the number of arguments as mentioned by the integer. If the argument is a floating point number, you would encounter an error message. If you apply other arithmetic operators on strings and lists, then you also encounter error messages. The following Python code showcases this behavior:

```
 1  >>> 'hello' + ' ' + 'world' + '!'
 2  'hello world!'
 3  >>> [1,2,3] + [4,5,6]
```

```
 4  [1, 2, 3, 4, 5, 6]
 5  >>> 'hello' *3
 6  'hello hello hello'
 7  >>> [1,2,3] * 3
 8  [1, 2, 3, 1, 2, 3, 1, 2, 3]
 9  >>> 'hello' *3.1
10  Traceback (most recent call last):
11  File "<stdin>", line 1, in <module>
12  TypeError: can't multiply sequence by non-int of type 'float'
13  >>> [1,2,3] *3.1
14  Traceback (most recent call last):
15  File "<stdin>", line 1, in <module>
16  TypeError: can't multiply sequence by non-int of type
'float'
17  >>> 'hello' /3
18  Traceback (most recent call last):
19  File "<stdin>", line 1, in <module>
20  TypeError: unsupported operand type(s) for /: 'str' and 'int'
21  >>> [1,2,3] // 3
22  Traceback (most recent call last):
23  File "<stdin>", line 1, in <module>
24  TypeError: unsupported operand type(s) for //: 'list' and 'int'
25  >>> 'hello' % 3
26  Traceback (most recent call last):
27  File "<stdin>", line 1, in <module>
28  TypeError: not all arguments converted during string formatting
29  >>> [1,2,3] % 3
30  Traceback (most recent call last):
31  File "<stdin>", line 1, in <module>
32  TypeError: unsupported operand type(s) for %: 'list' and 'int'
33  >>> [1,2,3] - 3
34  Traceback (most recent call last):
```

```
35  File "<stdin>", line 1, in <module>
36  TypeError: unsupported operand type(s) for -: 'list' and 'int'
37  >>> 'hello' - 3
38  Traceback (most recent call last):
39  File "<stdin>", line 1, in <module>
40  TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

The behavior of * on strings and lists can be understood if you consider mathematical multiplication in terms of addition: $2 * 5 = 2 + 2 + 2 + 2 + 5$. Hence, multiplication with m means adding m times.

5.5. Changing and Defining Data Type

Data types of objects can be changed as per their definitions. The function int() converts an input to an integer by rounding it off to the nearest integer. Similarly, float() makes the type of answer a floating point number. All integers can be made floating point numbers as 4 can also be defined as 4.0. The str() function converts its arguments to a string object. It is worth noting that the int() and float() functions cannot convert a string input to their data type. On the other hand, giving a numeral input to the str() function converts its data type to a string. It does not remain a number anymore but behaves like a character. The following Python code demonstrates this behavior:

```
1  >>> int(4.2345) # rounder to integer 4
2  4
3  >>> int(4.7345) # rounded to integer 4
4  4
5  >>> float(4) # converted to floating point number 4.0
6  4.0
7  >>> float(0.4) #remains a floating point number 0.4
8  0.4
```

```
 9  >>> str(4) #converts to a string '4'
10  '4'
11  >>> str(4.0) # converts to a string '0.4'
12  '4.0'
13  >>> a = str(4)+str(4.0) # strings '4' and '0.4' are concatenated
14  >>> a
15  '44.0'
16  >>> float ('sandeep')
17  Traceback (most recent call last):
18  File "<stdin>", line 1, in <module>
19  ValueError: could not convert string to float: 'sandeep'
20  >>> int('sandeep')
21  Traceback (most recent call last):
22  File "<stdin>", line 1, in <module>
23  ValueError: invalid literal for int() with base 10: 'sandeep'
```

5.5.1 Order of Usage

Python follows the PEMDAS (Parentheses, Exponents, Multiplication, Division, Addition, Subtraction) order of operations. Thus, during a complex calculation involving a number of arithmetic operators, entities are calculated in the following order :

1. Parentheses

2. Exponents

3. Multiplication

4. Division

5. Addition

6. Subtraction

This can be tested with the following Python code:

```
1  >>> 5 + (6−5) ∗ (10 / (−1 / 9)∗∗2)
2  815.0
3  >>> 5 ∗ 5 + 5 − 4 ∗∗ 2
4  14
```

5.5.2 Comparison Operators

Logical operators are supremely important for comparing objects. Operators used for comparison are called logical operators. Table 5-3 illustrates the behavior of these operators.

Table 5-3. *Comparison Operators*

Operator Symbol	Operator Meaning	Example
==	equal to	1==1 is True, 1==2 is False
!=	not equal to	1!=1 is False, 1==2 is True
<>	not equal to	1==1 is False, 1==2 is True
<	less than	1<2 is True, 2<1 is False
>	greater than	1>2 is False, 2>1 is True
<=	less than equal to	1<=1 is True, 1<=2 is True
>=	greater than equal to	1>=1 is True, 1>=2 is False

The result of logical operators is either one of the two binary objects aptly named True and False. In some programming languages, binary operators are represented as 1 and 0. They can also be compared for equality:

```
1  >>> not True
2  False
3  >>> a = True
```

```
 4  >>> b = False
 5  >>> a and b
 6  False
 7  >>> a or b
 8  True
 9  >>>1 >= 2 == 2 >= 1
10  False
11  >>>1 >= 2
12  False
13  >>>2 >= 1
14  True
15  >>>False == True
16  False
17  >>>False > True
18  False
19  >>>False < True
20  True
```

5.6. Membership Operator

The membership operator checks if a value(s) of variables is a member of a specified sequence. If the member is found, it returns the boolean value True; otherwise, it returns False:

```
1  >>> 'hello' in 'hello world'
2  True
3  >>> 'name' in 'hello world'
4  False
5  >>> a=3
6  >>> b=[1,2,3,4,5]
7  >>> a in b
```

```
 8  True
 9  >>> 10 in b
10  False
```

The operator in is used extensively in checking conditions for loops. It is one of the most convenient ways to run a loop. First, you construct a list of as per a defined condition/formula and then you run a loop until the condition is satisfied. This approach will become clearer in Chapter.

5.7. Identity Operator

To check if two values point to the same type of object, an identify operator is used. It returns a boolean value True if objects on either of its sides are the same and returns False otherwise:

```
 1  >>> 1 is 1 # both are integer type
 2  True
 3  >>> 1 is 1.0 # 1 is int and 1.0 is float type
 4  False
 5  >>> 1 is 2 # both are integer type
 6  True
 7  >>> '1' is 1 # '1' is str type and other is int type
 8  False
 9  >>> '1' is 1.0 # '1' is str type and other is float type
10  False
```

is not operator is negation of result with is operator.

```
1  >>> 1 is not 1.0
2  True
3  >>> 1 is not 1
4  False
```

5.8. Bitwise Operators

Data are stored as bits in computers. If we can operate directly on bits, we will have great flexibility and fast computation. However, it is difficult for humans to comprehend this concept since we are used to numerals defined in decimal format rather than binary format. Table 5-4 presents a list of bitwise operators.

Table 5-4. *Bitwise Operators*

Bitwise Operator	Description
>>	Bitwise left shift
<<	Bitwise right shift
&	Bitwise AND
|	Bitwise OR
~	Bitwise not

The binary composition of an `int` object can be shown using the `bin()` function as follows:

```
1  >>> bin(1)
2  '0b1'
3  >>> bin(10)
4  '0b1010'
5  >>> bin(100)
6  '0b1100100'
7  >>> bin(1000)
8  '0b1111101000'
```

When printing a binary representation of an integer number, 0b signifies that it is a binary representation. Now, it's easy to understand the following:

$$1010_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times ss^0 \tag{5.1}$$

$$= 8 + 0 + 2 + 0 = 10_{10} \tag{5.2}$$

Thus, $1010_2 = 10_{10}$. The decimal integer 10 is stored as the binary number 1010. Bitwise operators operate on this number at the bit level. So, the bitwise left shift operator will shift the value of bits one place to left. Likewise, the bitwise right shift operator will shift the bit value one step to right. This will result in a new binary representation and will be equivalent to a new decimal number. For example:

```
1   >>> bin(100)
2   '0b1100100'
3   >>> 100 << 1
4   200
5   >>> bin(200)
6   '0b11001000'
7   >>> 100>>1
8   50
9   >>> bin(50)
10  '0b110010'
11  >>> 100 << 2
12  400
13  >>> bin(400)
14  '0b110010000'
15  >>> 100 >> 2
16  25
17  >>> bin(25)
18  '0b11001'
```

The << shifts all bits of 100_{100} (1100100_2) to the left and fills the gap with a new 0 and, thus, results in 11001000_2, which is equivalent to 200_{10}. The >> shifts all bits of 100_{100} (1100100_2) to the right and results in 110010_2, which is equivalent to 50_{10}. Next, the bits are shifted two places.

The logical operators defined in Chapter 4 can be used here too. Python's boolean value True is equivalent to 1, and 0 is equivalent to Python's boolean value False. As a result, Tables 4-1, 4-2, 4-3, and 4-4 can be redesigned as table

- AND is 1 only if both of its inputs are 1; otherwise, it's 0.

- OR is 1 if one or both of its inputs are 1; otherwise, it's 0.

- XOR is 1 only if exactly one of its inputs is 1; otherwise it's 0.

- NOT is 1 only if its input is 0; otherwise, it's 0.

Truth tables are useful in understanding their operations.

By using Table 5-5, you can understand the following Python code:

```
1  >>> 20 and 200
2  200
3  >>> 20 or 200
4  20
5  >>> bin(20)
6  '0b10100'
7  >>> bin(200)
8  '0b11001000'
```

Table 5-5. *Truth Table for AND, OR,*
NOT, and XOR Operators

AND	0	1
0	0	0
1	0	1
OR	0	1
0	0	1
1	1	1
XOR	0	1
0	0	1
1	1	0
NOT	0	1
	1	0

When we write 20 or 200 and 20 and 200, their binary representations are acted upon. The results are also binary numbers that are converted back into integers and reported at REPL.

5.8.1 Using Bitwise Operations

Bitwise operations find their use while dealing with hardware registers in embedded systems. Every processor uses one or more registers (usually a specific memory address) that control whether an interrupt is enabled or disabled. When an interrupt is enabled, signals can be communicated. Interrupts are enabled by setting the enable bit for that particular interrupt and, most importantly, not modifying any of the other bits in the register. When an interrupt communicates with a data stream, it typically sets a bit in a status register so that a single service routine can determine the precise reason for the interrupt. Testing the individual bits allows for a

fast decode of the interrupt source. This is where bit operations come in handy. Shift operators are used to shift the bits as per a formula, whereas AND and OR operations are used to check the status of bits at a specific location. The same concept is used to alter the system file permission. In the Linux file system, each file has a number called its *mode*, which indicates the permission about accessing the file. This integer can be retrieved in a program to know the status of permissions for the file. For example, `if ((mode & 128) != 0) {<do this>}` will check the mode by checking if an appropriate bit is 0 in a 128-bit system. Bitwise operations are also preferred for their speed of operation since they directly operate on bits in the memory.

5.9. Summary

Operators play a very important role in computing as they provide the backbone of defining pathways for computing. All mathematical functions are expressed either by individual operators or by a combination of them. For a programming language that caters to a variety of fields such as science, engineering, business, and the arts, a lot of different kinds of operators are needed. Python is now being applied in various dimensions of life and maturing with a rich library of built-in as well as module-wise operators.

CHAPTER 6

Arrays

6.1 Introduction

Most often during scientific computation, a series of numbers needs to be operated upon together. The `list` data type stores a *sequence* of values. All elements of `list` can be accessed by their index, but individual `list` elements can belong to any data type. Hence, a new kind of object needs to be defined, similar to `list` but that stores only numeric values. This data type is called an `array`. This data type is not built in the Python interpreter, but it is within the module `numpy`.

The `numpy` module carries a unique object class called `array`. It carries member elements of only one data type. The concept of using arrays to store numerals gave rise to a powerful idea of array-based computing. The origins of this method can be traced back to matrix algebra. A matrix is also a collection of numbers. Similar to matrices, arrays can be multidimensional and can be operated on by operators defined the same way as those for mathematical matrices. Using matrices, problems involving a system of equations can be solved (which can even be coupled to each other) in one instance. Using the method of indexing of elements, particular elements can be accessed for operations. Using the concept of slicing, array dimensions can be altered as per requirements. Using operators acting on this object, mathematical formulations can be implemented. In this chapter, we will discuss the use of arrays for mathematical computations.

6.2 numpy

The numpy package contains various items that can be used for numerical computation—hence, the name *numerical Python*. NumPy originated from Numeric, which was originally created by Jim Hugunin with contributions from several other developers. Travis Oliphant created NumPy in 2005 by incorporating features of the competing Numarray into Numeric, with extensive modifications. numpy is released under an open source license. This chapter's code has been tested for version 1.12.1.

Just like any other Python module, numpy can be installed on Ubuntu with a simple pip program:

```
pip install numpy
```

numpy is pre-installed in Anaconda IDE, so windows and Mac OS X users are advised to work using Anaconda IDE.

To use, numpy can be imported and the version number can be checked:

```
1  >>> import numpy
2  >>> print (numpy.version.version)
3  1.12.1
```

Line 1 imports the whole module named numpy for our use. Line 2 uses the function version, which further uses the function version to find the installed version of numpy on the system. Users are encouraged to check their version of Python.

It is also important to note that importing a module is necessary for its usage, but once imported in a session, it doesn't need to be imported each time it is used. However, if the session is restarted, it must be freshly imported. For this chapter, it can be imported once and then codes can be written, but if a user shuts down the Python session and comes back to start it again, it must be imported again.

6.3 ndarray

ndarray is the main object of numpy, which is homogeneous (containing only one data type for all its member elements). The elements are created using the array() function of numpy. This functions needs a list as input. Elements are indexed by a tuple of positive integers:

```
 1  >>> a = [1,2,3]
 2  >>> a
 3  [1, 2, 3]
 4  >>> type(a)
 5  <class 'list'>
 6  >>> import numpy
 7  >>> b = numpy.array([1,2,3])
 8  >>> b
 9  array([1, 2, 3])
10  >>> type(b)
11  <class 'numpy.ndarray'>
12  >>> b.dtype
13  dtype('int64')
14  >>> c = numpy.array([1.0, 2.0, 3.0])
15  >>> c
16  array([ 1.,  2.,  3.])
17  >>> c.dtype
18  dtype('float64')
```

This example shows how lists and arrays in numpy are created differently. The numpy object array takes a list as input. The data type of elements can be assigned by the Python interpreter at the time of interpretation in a dynamic fashion. Alternatively, the data type can be defined at the time of creation too using the second argument of the array() function:

```
 1  >>>a1 = numpy.array([1,2,3], dtype=float)
 2  >>>a1
 3  array([ 1.,  2.,  3.])
 4  >>>a1.dtype
 5  dtype('float64')
 6  >>>a2 = numpy.array([1,2,3], dtype=complex)
 7  >>>a2
 8  array([ 1.+0.j,  2.+0.j,  3.+0.j])
 9  >>>a2.dtype
10  dtype('complex128')
```

ndarray is also known by its alias array. Apart from knowing the data type by using dtype, there are a variety of methods to get information about various attributes of ndarray, as shown in Table 6-1.

Table 6-1. *Various Methods to Probe the Property of an Array*

ndarray.dtype	Data type of elements
ndarray.ndim	Dimension of array
ndarray.shape	Shape of array, (n, m) for (n, m) array
ndarray.size	Size of array, $n \times m$
ndarray.itemsize	Size in bytes of each element
ndarray.data	Buffer data containing actual element
ndarray.reshape	Reshapes keeping $n \times m$ constant

Table 6-1 can be understood using the following Python code. We first define a 3 array named a3 and then probe the property of this object using the methods listed:

```
1  >>> a3 = numpy.array( [ (1,2,3), (4,5,6), (2,7,8) ] ) #
   defining array
2  >>> a3 # REPL prints contents of a3
3  array([[1, 2, 3],
4  [4, 5, 6],
5  [2, 7, 8]])
6  >>> a3.ndim # number if dimesnions of a3-2 i.e it can be
   described in terms of  rows and columns
7  2
8  >>> a3.size # total number of elements
9  9
10 >>> a3.shape # number of elements in each dimesnion as a tuple
11 (3, 3)
12 >>> a3.dtype # data type of member elements
13 dtype('int64')
14 >>> a3.data # address of memory location whre array is stored
15 <memory at 0x11070aa68>
16 >>> a3.itemsize # Size in bytes of each element
17 8
18 >>> a3.reshape(1,9) # reshaping 3 X 3 array as 1 X 9 array
   i.e 1 row and 9 columns
19 array([[1, 2, 3, 4, 5, 6, 2, 7, 8]])
20 >>> a3.reshape(9,1) # reshaping 3 X 3 array as 9 X 1 array
   i.e 9 row and 1 columns
21 array([[1],
22 [2],
23 [3],
24 [4],
25 [5],
26 [6],
27 [2],
28 [7],
```

```
29  [8]])
30  >>>a3.reshape(9,1) is a3.reshape(1,9)
31  False
32  # result is false because both arrays have different shapes
```

6.4 Automatic Creation of Arrays

Various functions exist to automatically create an array of the desired dimensions and shape. This comes in handy during mathematical calculations where creating arrays by hand is a tiresome task.

6.4.1 zeros()

To create an array where all elements are 0, we use the zeros() function:

```
1   >>> numpy.zeros((3,4)) # arrays of float tyoe by default
2   array([[ 0.,  0.,  0.,  0.],
3   [ 0.,  0.,  0.,  0.],
4   [ 0.,  0.,  0.,  0.]])
5   >>> numpy.zeros((3,4),dtype=int) # array of int data type
6   array([[0, 0, 0, 0],
7   [0, 0, 0, 0],
8   [0, 0, 0, 0]])
9   >>> numpy.zeros((3,4),dtype=complex) # array of complex
    data type
10  array([[ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
11  [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
12  [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j]])
13  >>> numpy.zeros((3,4),dtype=str) # array of empty strings
14  array([['', '', '', ''],
15  ['', '', '', ''],
16  ['', '', '', '']],
17  dtype='<U1')
```

During the initialization to zero values for matrix computations, the `zeros()` function is extensively used.

6.4.2 ones()

To create an array where all elements are 1, we use the `ones()` function:

```
 1  >>> numpy.ones((3,4)) # array of float values
 2  array([[ 1.,  1.,  1.,  1.],
 3  [ 1.,  1.,  1.,  1.],
 4  [ 1.,  1.,  1.,  1.]])
 5  >>> numpy.ones((3,4),dtype=int) # array of int values
 6  array([[1, 1, 1, 1],
 7  [1, 1, 1, 1],
 8  [1, 1, 1, 1]])
 9  >>> numpy.ones((3,4),dtype=complex) # array of complex
    number values
10  array([[ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j],
11  [ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j],
12  [ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j]])
13  >>> numpy.ones((3,4),dtype=str) # array of strings
14  array([['1', '1', '1', '1'],
15  ['1', '1', '1', '1'],
16  ['1', '1', '1', '1']],
17  dtype='<U1')
```

It is interesting to observe that the complex number 1+0.j is equivalent to 1.0 mathematically, but the data type determines the ways in which the data will be stored in computer memory as well as how it will be operated upon using operators. So, it should be used judiciously.

6.4.3 ones_like()

Taking its cue from an existing array, `ones_like()` creates a `ones` array of similar shape and type:

```
 1  >>> a = numpy.array([[1.1, 2.2, 4.1],[2.5,5.2,6.4]])
 2  >>> a
 3  array([[ 1.1,  2.2,  4.1],
 4  [ 2.5,  5.2,  6.4]])
 5  >>> numpy.ones_like(a)
 6  array([[ 1.,  1.,  1.],
 7  [ 1.,  1.,  1.]])
 8  >>> numpy.ones_like(a,dtype=int)
 9  array([[1, 1, 1],
10  [1, 1, 1]])
11  >>> numpy.ones_like(a,dtype=complex)
12  array([[ 1.+0.j,  1.+0.j,  1.+0.j],
13  [ 1.+0.j,  1.+0.j,  1.+0.j]])
14  >>> numpy.ones_like(a,dtype=str)
15  array([['1', '1', '1'],
16  ['1', '1', '1']],
17  dtype='<U1')
```

6.4.4 empty()

`empty()` returns an array of uninitialized (arbitrary) data of the given shape, dtype, and order. Object arrays will be initialized to None. Unlike `zeros`, it does not always set the array values to 0 value:

```
 1  >>> numpy.empty((2,2))
 2  array([[ 0.,  0.],
 3  [ 0.,  0.]])
 4  >>> numpy.empty((2,2),dtype=int)
```

```
 5  array([[0, 0],
 6  [0, 0]])
 7  >>> numpy.empty((2,2),dtype=complex)
 8  array([[  3.10503618e+231 +3.10503618e+231j,
 9  1.23516411e-322 +0.00000000e+000j],
10  [  0.00000000e+000 +2.21170104e-314j,
11  2.21184108e-314 +2.25920578e-314j]])
12  >>> numpy.empty((2,2),dtype=str)
13  array([['', ''],
14  ['', '']],
15  dtype='<U1')
```

6.4.5 empty_like()

Taking its cue from an existing array, `empty_like()` creates an `empty` array of similar shape and type:

```
1  >>> a = numpy.array([[1.1, 2.2, 4.1],[2.5,5.2,6.4]])
2  >>> a
3  array([[ 1.1,  2.2,  4.1],
4  [ 2.5,  5.2,  6.4]])
5  >>> numpy.empty_like(a)
6  array([[ 0.,  0.,  0.],
7  [ 0.,  0.,  0.]])
```

6.4.6 eye()

Similar to an identity matrix, `eye()` returns a two-dimensional array where diagonal elements are valued equal to 1. The function takes the second argument as the index of the diagonal. 0 indicates the central diagonal. A positive number means moving a designated number of steps in a vertical upward direction. A negative number means moving a designated number of steps in a vertical downward direction.

```
 1  >>> numpy.eye(3, k=0)
 2  array([[ 1.,  0.,  0.],
 3  [ 0.,  1.,  0.],
 4  [ 0.,  0.,  1.]])
 5  >>> numpy.eye(3, k=1)
 6  array([[ 0.,  1.,  0.],
 7  [ 0.,  0.,  1.],
 8  [ 0.,  0.,  0.]])
 9  >>> numpy.eye(3, k=-1)
10  array([[ 0.,  0.,  0.],
11  [ 1.,  0.,  0.],
12  [ 0.,  1.,  0.]])
13  >>> numpy.eye(3, k=-2)
14  array([[ 0.,  0.,  0.],
15  [ 0.,  0.,  0.],
16  [ 1.,  0.,  0.]])
17  >>> numpy.eye(3, k=2)
18  array([[ 0.,  0.,  1.],
19  [ 0.,  0.,  0.],
20  [ 0.,  0.,  0.]])
21  >>> numpy.eye(3, k=1,dtype=int)
22  array([[0, 1, 0],
23  [0, 0, 1],
24  [0, 0, 0]])
25  >>> numpy.eye(3, k=1,dtype=complex)
26  array([[ 0.+0.j,  1.+0.j,  0.+0.j],
27  [ 0.+0.j,  0.+0.j,  1.+0.j],
28  [ 0.+0.j,  0.+0.j,  0.+0.j]])
29  >>> numpy.eye(3, k=1,dtype=str)
30  array([['', '1', ''],
31  ['', '', '1'],
32  ['', '', '']],
33  dtype='<U1')
```

6.4.7 identity()

The identity() function generates a two-dimensional identity array with 1 as diagonal values. An equivalent matrix will be the one with determinant as 1—hence, the name *identity*.

```
 1  >>> numpy.identity(4)
 2  array([[ 1.,  0.,  0.,  0.],
 3  [ 0.,  1.,  0.,  0.],
 4  [ 0.,  0.,  1.,  0.],
 5  [ 0.,  0.,  0.,  1.]])
 6  >>> numpy.identity(4,dtype=int)
 7  array([[1, 0, 0, 0],
 8  [0, 1, 0, 0],
 9  [0, 0, 1, 0],
10  [0, 0, 0, 1]])
11  >>> numpy.identity(4,dtype=complex)
12  array([[ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
13  [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
14  [ 0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
15  [ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j]])
16  >>> numpy.identity(4,dtype=str)
17  array([['1', '', '', ''],
18  ['', '1', '', ''],
19  ['', '', '1', ''],
20  ['', '', '', '1']],
21  dtype='<U1')
```

6.4.8 full()

full fills up particular data into all elemental positions:

```
1  >>>>>> numpy.full((3,2),10.5,dtype=int)
2  array([[10, 10],
3  [10, 10],
4  [10, 10]])
5  >>> numpy.full((3,2),10.5)
6  array([[ 10.5,  10.5],
7  [ 10.5,  10.5],
8  [ 10.5,  10.5]])
9  >>> numpy.full((3,2),10.5,dtype=complex)
10  array([[ 10.5+0.j,  10.5+0.j],
11  [ 10.5+0.j,  10.5+0.j],
12  [ 10.5+0.j,  10.5+0.j]])
13  >>> numpy.full((3,2),10.5+2j,dtype=complex)
14  array([[ 10.5+2.j,  10.5+2.j],
15  [ 10.5+2.j,  10.5+2.j],
16  [ 10.5+2.j,  10.5+2.j]])
17  >>> numpy.full((3,2),'a',dtype=str)
18  array([['a', 'a'],
19  ['a', 'a'],
20  ['a', 'a']],
21  dtype='<U1')
```

6.4.9 full_like()

Just like empty_like and ones_like, full_like creates a new matrix taking shape and data types from an existing array:

```
1  >>> numpy.full_like(a,5.0)
2  array([[ 5.,  5.,  5.],
```

```
 3  [ 5.,   5.,   5.]])
 4  >>> numpy.full_like(a,5.0,dtype=int)
 5  array([[5, 5, 5],
 6  [5, 5, 5]])
 7  >>> numpy.full_like(a,5.0+2.3j,dtype=complex)
 8  array([[ 5.+2.3j,  5.+2.3j,  5.+2.3j],
 9  [ 5.+2.3j,  5.+2.3j,  5.+2.3j]])
10  >>> numpy.full_like(a,'a',dtype=str)
11  array([['a', 'a', 'a'],
12  ['a', 'a', 'a']],
13  dtype='<U1')
```

6.4.10 Random Numbers

To create a random array (filled up with random numbers), we use the
random function as follows:

```
1  >>> numpy.random.rand(4)
2  array([ 0.60855254,  0.81713983,  0.01249653,  0.55668541])
3  >>> numpy.random.rand(4,4)
4  array([[ 0.56770136,  0.45094011,  0.50791373,  0.29436197],
5  [ 0.664363  ,  0.73385773,  0.55709437,  0.84207538],
6  [ 0.15050127,  0.56031549,  0.07850941,  0.0052116 ],
7  [ 0.54952592,  0.66695696,  0.07617147,  0.00457323]])
```

Note that the function rand() comes inside the subpackage random
within numpy. To get complete details of this wonderful package, we
encourage users to explore it using help(numpy.random) after issuing the
command import numpy.

The documentation describes a rich library of functions to create
random numbers as per choice. This makes numpy a good choice for
libraries used in simulation work. The choice of distribution depends

on the properties of the system under study. Normalized distribution of random numbers must be used for systems following Gaussian statistics.

Random Integers

Random integers can be generated by using the `numpy.random.random_integers()` and `numpy.random.randint()` functions. The latter needs the first input as starting of the limit and the second as ending the limit for random numbers.

The thrd argument can be a list describing the shape of the matrix:

```
 1  >>> numpy.random.random_integers(10) # a random integer
    uptill 10
 2  5
 3  >>> numpy.random.random_integers(10) # a random integer
    uptill 10
 4  8
 5  >>> numpy.random.randint(0,5,3) # a random integer array:
    form 0 to 5 with 3 elements
 6  array([1, 2, 4])
 7  >>> numpy.random.randint(2,5,3) # a random integer array:
    form 2 to 5 with 3 elements
 8  array([2, 4, 3])
 9  >>> numpy.random.randint(2,50,3) # a random integer array:
    form 2 to 50 with 3 elements
10  array([36, 43, 29])
11  >>> numpy.random.randint(2,50,[2,3]) # a random integer
    array form 2 to 50 with shape of 2 X 3 i.e 2 rows and 3
    columns
12  array([[ 5, 22, 23],
13  [36, 13, 11]])
14  >>> numpy.random.randint(2,50,[5,3]) # a random integer array
    form 2 to 50 with shape of 5 X 3 i.e 5 rows and 3 columns
```

```
15  array([[10, 30, 34],
16  [20, 26, 44],
17  [30, 42, 37],
18  [49, 36, 20],
19  [43, 23, 37]])
```

Random Floating Point Numbers

Random floating point numbers can be generated by the numpy.random.
ranf() function where the input argument can be a list. As per members
of the list, the dimension and size of the array is determined. Also the
elements belong to the interval [0.0, 1.0):

```
 1  >>> numpy.random.ranf([2,3]) # array of random numbers with
    size 2 X 3 i.e 2 rows and 3 columns
 2  array([[ 0.94222024,  0.75429361,  0.52951222],
 3  [ 0.87956205,  0.12400001,  0.78250773]])
 4  >>> numpy.random.ranf([5,3]) # array of random numbers with
    size 5 X 3 i.e 5 rows and 3 columns
 5  array([[ 0.45427483,  0.78061704,  0.38300021],
 6  [ 0.40070286,  0.2259602 ,  0.37309902],
 7  [ 0.18188091,  0.02052539,  0.63328754],
 8  [ 0.82842309,  0.95054644,  0.30800412],
 9  [ 0.15268755,  0.6668035 ,  0.3047649 ]])
10  >>> numpy.random.ranf(20) # array of 20 random numbers,
    size is decided to be 1 X 20
11  array([ 0.18816171,  0.8807835 ,  0.03181879,  0.83308999,
    0.80802698,
12  0.98561721,  0.40781223,  0.21523024,  0.15926338,  0.72142345,
13  0.14311937,  0.50017394,  0.67017638,  0.69552528,  0.46469783,
14  0.83841762,  0.81342313,  0.13859541,  0.77675513,  0.71401225])
15  >>> numpy.shape(numpy.random.ranf(20))
16  (20,)
```

101

Random Choice

Given one array, another array can be generated with its elements using the `numpy.random.choice()` function:

```
 1  >>> a = numpy.array([1,2,3,4,5,6]) # define a target array
 2  >>> numpy.random.choice(a,3) # from target array, define an
    array of 3 X 1 shape
 3  array([3, 4, 1])
 4  >>> numpy.random.choice(a,3) # from target array, define an
    array of 3 X 1 shape
 5  array([4, 2, 2])
 6  >>> numpy.random.choice(a,3) # from target array, define an
    array of 3 X 1 shape
 7  array([1, 2, 3])
 8  >>> numpy.random.choice(a,[2,3]) # from target array,
    define an array of 2 X 3 shape
 9  array([[4, 3, 4],
10  [6, 1, 5]])
11  >>> numpy.random.choice(a,[2,3]) # from target array,
    define an array of 2 X 3 shape
12  array([[5, 5, 2],
13  [6, 4, 4]])
```

Another function, `numpy.random.shuffle()`, shuffles the contents of an array. It is important to note that only rows are shuffled:

```
 1  >>> a = numpy.random.randint(0,10,[10,10]) # an array of
    random integers from 0 to 10 in shape 10 X 10
 2  >>> a
 3  array([[1, 0, 5, 7, 6, 7, 6, 7, 3, 3],
 4  [0, 8, 3, 4, 8, 1, 9, 0, 6, 2],
 5  [5, 7, 6, 7, 1, 1, 0, 9, 6, 8],
```

```
 6  [3, 3, 1, 9, 4, 4, 1, 2, 8, 6],
 7  [5, 6, 8, 4, 3, 6, 8, 9, 3, 4],
 8  [3, 7, 0, 4, 4, 2, 2, 9, 6, 8],
 9  [1, 6, 4, 8, 8, 6, 9, 2, 3, 2],
10  [8, 4, 0, 2, 4, 0, 4, 5, 2, 9],
11  [5, 8, 2, 1, 4, 8, 8, 3, 2, 6],
12  [8, 4, 0, 1, 9, 1, 7, 0, 9, 9]])
13  >>> numpy.random.shuffle(a) # shuffle content of a
14  >>> a # contents of a have been shuffled indeed
15  array([[1, 6, 4, 8, 8, 6, 9, 2, 3, 2],
16  [1, 0, 5, 7, 6, 7, 6, 7, 3, 3],
17  [3, 3, 1, 9, 4, 4, 1, 2, 8, 6],
18  [3, 7, 0, 4, 4, 2, 2, 9, 6, 8],
19  [5, 6, 8, 4, 3, 6, 8, 9, 3, 4],
20  [0, 8, 3, 4, 8, 1, 9, 0, 6, 2],
21  [8, 4, 0, 2, 4, 0, 4, 5, 2, 9],
22  [8, 4, 0, 1, 9, 1, 7, 0, 9, 9],
23  [5, 7, 6, 7, 1, 1, 0, 9, 6, 8],
24  [5, 8, 2, 1, 4, 8, 8, 3, 2, 6]])
```

The numpy.random.permutation() function randomly permutes a sequence:

```
1  >>> a = numpy.random.randint(0,10,[10,10])
2  >>> a
3  array([[9, 6, 5, 0, 2, 5, 2, 6, 4, 5],
4  [6, 2, 0, 6, 9, 7, 5, 1, 6, 4],
5  [9, 3, 2, 6, 6, 7, 3, 1, 8, 8],
6  [9, 6, 4, 8, 2, 8, 7, 7, 4, 7],
7  [2, 6, 1, 1, 3, 0, 8, 5, 3, 5],
8  [8, 8, 7, 3, 4, 3, 5, 5, 0, 2],
9  [3, 5, 6, 8, 6, 6, 0, 7, 9, 2],
```

```
10  [8, 9, 6, 9, 4, 4, 5, 8, 1, 9],
11  [6, 0, 1, 8, 8, 7, 2, 7, 2, 1],
12  [1, 3, 4, 8, 9, 6, 4, 1, 9, 4]])
13  >>> numpy.random.permutation(a)
14  array([[8, 8, 7, 3, 4, 3, 5, 5, 0, 2],
15  [9, 6, 4, 8, 2, 8, 7, 7, 4, 7],
16  [3, 5, 6, 8, 6, 6, 0, 7, 9, 2],
17  [2, 6, 1, 1, 3, 0, 8, 5, 3, 5],
18  [9, 3, 2, 6, 6, 7, 3, 1, 8, 8],
19  [1, 3, 4, 8, 9, 6, 4, 1, 9, 4],
20  [6, 0, 1, 8, 8, 7, 2, 7, 2, 1],
21  [6, 2, 0, 6, 9, 7, 5, 1, 6, 4],
22  [8, 9, 6, 9, 4, 4, 5, 8, 1, 9],
23  [9, 6, 5, 0, 2, 5, 2, 6, 4, 5]])
```

Beta Distribution

The function numpy.random.beta() returns an array that follows beta distribution, which is often encountered in Bayesian inference and order statistics:

$$f(x;a,b) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1} (1-x)^{\beta-1} \tag{6.1}$$

$$B(\alpha,\beta) = \int_0^1 t^{\alpha-1} (1-t)^{\beta-1} \, dt \tag{6.2}$$

```
1  >>> a = numpy.random.beta(2,10,10)
2  >>> a
3  array([ 0.06604658,  0.21840079,  0.04368817,  0.08380568,
   0.12769291,
4  0.06450275,  0.04227899,  0.04637705,  0.19960681,  0.24094982])
5  >>> a = numpy.random.beta(2,10,[3,3])
```

```
6  >>> a
7  array([[ 0.03500157,  0.15174638,  0.19104822],
8  [ 0.09989465,  0.0410233 ,  0.10818969],
9  [ 0.05473703,  0.53414904,  0.09791452]])
```

Binomial Distribution

The function `numpy.random.binomial(n,p,size=)` returns an array that follows binomial distribution [1]. Samples are drawn from a binomial distribution with specified parameters, n trials, and p probability of success where n is an integer >= 0 and p is in the interval [0, 1]. Let's consider the case where we have 50 trials and the probability of success is 80%:

```
1  >>> n, p = 50, .8
2  >>> a = numpy.random.binomial(n,p,10)
3  >>> a
4  array([41, 35, 42, 42, 34, 38, 38, 42, 40, 43])
5  >>> a = numpy.random.binomial(n,p,[3,3])
6  >>> a
7  array([[39, 36, 41],
8  [42, 38, 38],
9  [40, 43, 40]])
```

Normal Distribution

A normal (Gaussian) distribution of random numbers with a fixed mean (a) and standard deviation (b) can be generated by using `numpy.random.random(a,b,size=)`:

```
1  >>> numpy.random.normal(2,1,20) # mean=2, SD=1, 20 numbers
2  array([ 1.8625696 ,  0.80712566,  2.24495879,  3.14961488,
   1.38803327,
3  2.59744647,  0.99560626,  2.79150258,  2.87579258,  2.01958405,
4  2.33145923,  2.81541595,  2.25550088,  2.13514339,  0.89957839,
```

```
5  0.95178737,  1.16911591,  4.32423549,  3.3398163 ,  1.35927406])
6  >>> numpy.random.normal(2,1,[2,3]) # mean=2, SD=1, 2 X 3 array
7  array([[ 4.6907596 ,  0.47341209,  2.09379377],
8  [ 3.44982049,  1.1475949 ,  3.03557547]])
```

Other Distributions

A lot of other distributions can also be used in the numpy.random
subpackage [2]. The choice of a particular distribution depends on the
simulation under study. Under a certain set of conditions, physical systems
follow a particular set of distributions.

6.5 Numerical Ranges

Creating a sequence of numbers is an integral part of a numerical
computation. A variety of functions exists to create a desired sequence of
numbers automatically, as per a defined rule.

6.5.1 A Range of Numbers

The syntax for automatically generating a range of numbers from a starting
point to a stop point with a step size is given by the following:

```
numpy.arange([start, ]stop, [step, ]dtype=)
 1  >>> numpy.arange(1,10,0.5) # start=1,stop=10,step=0.5
 2  array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
    5.5,  6. ,
 3  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])
 4  >>> numpy.arange(1,10,1) # start=1,stop=10,step=1
 5  array([1, 2, 3, 4, 5, 6, 7, 8, 9])
 6  >>> numpy.arange(10) # start=0 (defualt),stop=10,step=1
    (default)
```

```
 7  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
 8  >>> numpy.arange(1+2j,10+3j,0.1,dtype=complex) #
    start=1+2j, stop=10+3j, step=0.1
 9  array([ 1.0+2.j,  1.1+2.j,  1.2+2.j,  1.3+2.j,  1.4+2.j,
    1.5+2.j,
10  1.6+2.j,  1.7+2.j,  1.8+2.j,  1.9+2.j])
```

6.5.2 Linearly Spaced Numbers

Whereas arange() gives you good control over step size, you cannot specify the number of elements in the array. To solve this issue, the linspace() function is defined with the following syntax:

linspace(start, stop, num, endpoint, dtype)

```
1  >>> numpy.linspace(1,10,5) # start=1,stop=10,number of
   points=5
2  array([  1.  ,    3.25,    5.5 ,    7.75,   10.  ])
3  >>> numpy.linspace(1,10,10) # start=1,stop=10,number of
   points=10
4  array([  1.,    2.,    3.,    4.,    5.,    6.,    7.,    8.,
   9.,   10.])
5  >>> numpy.linspace(1,10,10,endpoint=False) #
   start=1,stop=10,number of points=10 without including end
   point
6 array([ 1. ,  1.9,  2.8,  3.7,  4.6,  5.5,  6.4,  7.3,  8.2,  9.1])
7  >>> numpy.linspace(1,10,10,endpoint=True) #
   start=1,stop=10,number of points=10 including the end point
8  array([  1.,    2.,    3.,    4.,    5.,    6.,    7.,    8.,
   9.,   10.])
```

6.5.3 Logarithmically Spaced Numbers

Just as linearly spaced points are generated by `linspace()`, `logspace()` generates linearly spaced points on a logarithmic axis:

```
logspace(start, stop, num, endpoint=, base=, dtype=)
```

```
1  >>> numpy.logspace(1,10, num=3) # 3 logarithmically spaced
   points between 1 and 10
2  array([  1.00000000e+01,    3.16227766e+05,
   1.00000000e+10])
3  >>> numpy.logspace(1,10, num=5) # 5 logarithmically spaced
   points between 1 and 10
4  array([  1.00000000e+01,    1.77827941e+03,    3.16227766e+05,
5  5.62341325e+07,    1.00000000e+10])
6  >>> numpy.logspace(1,10, num=5,endpoint=False) # 3
   logarithmically spaced points between 1 and 10 without
   inclduing the end point
7  array([  1.00000000e+01,    6.30957344e+02,    3.98107171e+04,
8  2.51188643e+06,    1.58489319e+08])
```

6.5.4 meshgrid()

The `meshgrid` is modeled after the MATLAB `meshgrid` command. To understand the working of `meshgrid`, it's best to use it once as follows:

```
1  >>> xx = numpy.linspace(1,3,3)
2  >>> xx
3  array([ 1.,  2.,  3.])
4  >>> yy = numpy.linspace(2,4,3)
5  >>> yy
6  array([ 2.,  3.,  4.])
7  >>> (a,b) = numpy.meshgrid(xx,yy)
```

```
 8  >>> a
 9  array([[ 1.,  2.,  3.],
10  [ 1.,  2.,  3.],
11  [ 1.,  2.,  3.]])
12  >>> b
13  array([[ 2.,  2.,  2.],
14  [ 3.,  3.,  3.],
15  [ 4.,  4.,  4.]])
```

meshgrid() makes a two-dimensional coordinate system where the x-axis is given by the first argument (here xx) and the y-axis is given by the second argument (here yy). This function is used while plotting three-dimensional plots or defining a function defined on two variables.

6.5.5 mgrid() and ogrid()

mgrid and ogrid are used to created mesh directly (that is, without using linspace, arange, and so on). A simple statement like

mgrid[a:b , c:d]

constructs a grid where the x-axis has points from a to b and the y-axis has points from c to d.

mgrid constructs a multidimensional meshgrid. The following example demonstrates its use:

```
1  >>> (a,b) = numpy.mgrid[1:10 , 2:5]
2  >>> a
3  array([[1, 1, 1],
4  [2, 2, 2],
5  [3, 3, 3],
6  [4, 4, 4],
7  [5, 5, 5],
```

```
 8  [6, 6, 6],
 9  [7, 7, 7],
10  [8, 8, 8],
11  [9, 9, 9]])
12  >>> b
13  array([[2, 3, 4],
14  [2, 3, 4],
15  [2, 3, 4],
16  [2, 3, 4],
17  [2, 3, 4],
18  [2, 3, 4],
19  [2, 3, 4],
20  [2, 3, 4],
21  [2, 3, 4]])
22
23  >>> x, y = numpy.ogrid[0:5, 0:5]
24  >>> x
25  array([[0],
26  [1],
27  [2],
28  [3],
29  [4]])
30  >>> y
31  array([[0, 1, 2, 3, 4]])
```

6.6 tile()

From a target array of a smaller size, a bigger array can be constructed using the `tile()` function. `tile()` makes a copy of an existing array by the defined number of times needed to make a new array:

```
 1  >>> a = numpy.array([1,2,3])
 2  >>> a
 3  array([1, 2, 3])
 4  >>> b = numpy.tile(a,3) # array a is repeated 3 times to
    make a new array b
 5  >>> b
 6  array([1, 2, 3, 1, 2, 3, 1, 2, 3])
 7  >>># Another example to do the same for two dimesnional
    array
 8  >>> a1 = numpy.eye(4)
 9  >>> a1
10  array([[ 1.,  0.,  0.,  0.],
11  [ 0.,  1.,  0.,  0.],
12  [ 0.,  0.,  1.,  0.],
13  [ 0.,  0.,  0.,  1.]])
14  >>> a2=numpy.tile(a1,2) # repeat a1 in first dimesnion only
    i.e row-wise
15  >>> a2
16  array([[ 1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
17  [ 0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.],
18  [ 0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.],
19  [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.]])
20  >>> a2 = numpy.tile(a1,(2,2)) # repeat a1 in both
    dimesnions 2 times
21  >>> a2
22  array([[ 1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
```

```
23  [ 0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.],
24  [ 0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.],
25  [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.],
26  [ 1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
27  [ 0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.],
28  [ 0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.],
29  [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.]])
```

6.7 Broadcasting

Basic operations on numpy arrays are elementwise. This means that the dimensions of the arrays should be compatible for the desired operation. For example, 2 arrays and 2 arrays would be incompatible as the first array has one fewer column than the second one.

```
 1  >>>a = numpy.eye(4)
 2  >>>b = numpy.array([1,2,3,4])
 3  >>>c = a + b
 4  >>>c
 5
 6  array([[ 2.,  2.,  3.,  4.],
 7        [ 1.,  3.,  3.,  4.],
 8        [ 1.,  2.,  4.,  4.],
 9        [ 1.,  2.,  3.,  5.]])
10
11  >>>a.shape
12  (4L, 4L)
13
14  >>>b.shape
15  (4L,)
16
```

```
17  >>>c.shape
18  (4L, 4L)
19
20  # bradcasting enables array a ( 4 X 4) to be added to b (4
    X 1) to produce an array c (4 X 4)
21
22  # Another example
23  >>>a = numpy.eye(4)
24  >>>a
25
26  array([[ 1.,  0.,  0.,  0.],
27        [ 0.,  1.,  0.,  0.],
28        [ 0.,  0.,  1.,  0.],
29        [ 0.,  0.,  0.,  1.]])
30
31  >>>b = numpy.array([10, 10, 10, 10])
32  >>>c = a + b
33  >>>c
34
35  array([[ 11.,  10.,  10.,  10.],
36        [ 10.,  11.,  10.,  10.],
37        [ 10.,  10.,  11.,  10.],
38        [ 10.,  10.,  10.,  11.]])
39
40  # A 4 X 4 matrix can be operated with a 4 X 1 matrix by
    making the 'invisible' elemets zero.
```

6.8 Extracting Diagonal

The diagonal elements of a given array/matrix can be obtained using the diag() function. You can choose the diagonal using the second argument. The default value is 0, which indicates the main diagonal. A positive value indicates moving in an upward direction. A negative value indicates moving in a downward direction.

```
 1  >>> a = numpy.eye(5) # making an identity matrix
 2  >>> a
 3  array([[ 1.,  0.,  0.,  0.,  0.],
 4  [ 0.,  1.,  0.,  0.,  0.],
 5  [ 0.,  0.,  1.,  0.,  0.],
 6  [ 0.,  0.,  0.,  1.,  0.],
 7  [ 0.,  0.,  0.,  0.,  1.]])
 8  >>> numpy.diag(a) #extracting element of main diaginal
 9  array([ 1.,  1.,  1.,  1.,  1.])
10  >>> numpy.diag(a,1) # extractiong diagonal elements in
    upwards direction
11  array([ 0.,  0.,  0.,  0.])
12  >>> numpy.diag(a,-1) # extractiong diagonal elements in
    downwards direction
13  array([ 0.,  0.,  0.,  0.])
```

6.9 Indexing

The elements of an array or list start with 0 in Python; the first element is indexed 0. All elements can be accessed using their indexes:

```
 1  >>>a =[1,2,3,4,5,6] # creating a list and storing it in
    variable name 'a'
 2  >>> type(a) # a stores a 'list' object
 3  <class 'list'>
```

```
4  >>> b = numpy.array(a) # An array is referenced by 'b'
   using list referenced by 'a'
5  >>> b
6  array([1, 2, 3, 4, 5, 6])
7  >>> type(b)
8  <class 'numpy.ndarray'>
9  >>>a[1] # accessing second element from left hand side for
   the list 'a'
10 2
11 >>>b[0] # accessing first element from left hand side for
   the array 'b'
12 1
13 >>>a[-1] # accessing the first element from the right hand
   side for list 'a'
14 6
15 >>>b[-2] # accessing the second element from the right hand
   side for array 'b'
16 5
```

These examples make it clear that arrays are similar to lists and follow the same rules of indexing. Multidimensional arrays also follow the same pattern of indexing. For two-dimensional arrays, the first number indicates the row and the second number indicates the columns:

```
1  >>>a1 = numpy.array([[1,2,3],[3,2,1]]) # 2D array created
   using 2 sets of list
2  >>>a1
3  array([[1, 2, 3],
4      [3, 2, 1]])
5  >>>a1[1,2] # choosing elemnet whose row is indexed 1 and
   coloumn is indexed 2 i.e second row and thierd coloumn i.e
   down-right last element
```

```
 6   1
 7   >>>a1[1,1] # chooisng an element whose row is indexed 1 and
     clolumn is indexed 1 i.e. second row and second coloumn
 8   2
 9   >>>a1[1] # chooisng row with index 1 i.e second row
10   array([3, 2, 1])
```

Indexes can also be used to assign a particular value of the element. Here you use of the assignment operator discussed earlier:

```
1   >>>a1 = numpy.array([[1,2,3],[3,2,1]]) # 2D array created
    using 2 sets of list
2   >>>a1
3   array([[1, 2, 3],
4        [3, 2, 1]])
5   >>>a1[1,1] = 0 # elemnet with row index=1, column index=1 is
    set to be 0
6   >>>a1 # second row and second coloumns element i.e 2 is
    changed to 0
7   array([[1, 2, 3],
8        [3, 0, 1]])
```

6.10 Slicing

Among the first operations to be applied on arrays is *slicing*. Slicing employs the operator :, which is used to separate the data on the row:

```
1   >>>a1 = numpy.arange(10) # an array from 1 to 10
2   >>>a1
3   array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
4   >>>a1[0:5] # [0:5] selecets 'from' index 0 'untill' index 5
    i.e. excluding index 5
```

```
 5  array([0, 1, 2, 3, 4])
 6  >>>a1[:5] # [:5] selecets 'from' starting 'untill' index 5
    i.e. excluding index 5
 7  array([0, 1, 2, 3, 4])
 8  >>>a1[2:5] # [2:5] selecets 'from' index 2 'untill' index 5
    i.e. excluding index 5
 9  array([2, 3, 4])
10  >>>a1[2:-2] # [2:-2] selecets 'from' index 2 'untill' index -2
    (counting from right starts from -1) i.e. exclusing index -2
11  array([2, 3, 4, 5, 6, 7])
12  >>>a1[2:] # [2:] selecets 'from' index 2 'untill' last
    index
13  array([2, 3, 4, 5, 6, 7, 8, 9])
14  >>>a1[0:5:2] # [start:stop:step] =[0:5:2] hence it takes a
    tep of 2 while choosing indices from 0 to 5
15  array([0, 2, 4])
```

Slicing operations can also be accomplished for multidimensional arrays. Here a tuple should be passed to define slicing for each desired dimension. The following Python code will demonstrate the same:

```
 1  >>>a = [1,2,3,4,5] # 'a' refernece to a list
 2  >>>b = [5,6,7,8,9] # 'b' references to a list
 3  a1 = numpy.array([a,b]) # 'a' and 'b' are used to create a
    2D array of shape 2 X 5 array
 4  >>>a1
 5  array([[1, 2, 3, 4, 5],
 6      [5, 6, 7, 8, 9]])
 7  >>>a1.ndim # the dimension of array a1
 8  2
 9  >>>a1[0:2,0:2] # Start collecting elements from row indexed
    0 and coloumn indexed 2
```

```
10  array([[1, 2],
11      [5, 6]])
12  >>>a1[0:2,0:4] # First slice indicates to collect only
    first two elements by second slice indicates to collect
    first four elements, hence using broadcasting the result is
    implemented
13  array([[1, 2, 3, 4],
14      [5, 6, 7, 8]])
15  >>>a1[1:2,-1:] # because second slice indicates to collect
    the last element
16  array([[9]])
```

Similar logic can be applied to any dimensional array. Slicing becomes an extremely important tool for data filtering. In some cases, we would like to work with only specific rows and/or columns of data. In that case, the data can be sliced as per your needs.

6.11 Copies and Views

From the memory usage point of view, the slicing operation creates just a view of the original array. Using numpy.may_share_memory(), we can verify this claim:

```
1  >>>a = numpy.arange(10)
2  >>>a
3  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
4  >>>b = a[2:5]
5  >>>b
6  array([2, 3, 4])
7  >>>numpy.may_share_memory(a,b) # array 'a' and 'b' share
    same memory space
8  True
```

```
 9  >>># Now if we change first element of 'b', array 'a'
    element will also change since 'b' is just a "view" of 'a'
10  >>> b[0]=10 # first element of 'b' is set to value 10
11  >>> b # change of value is reflected here
12  array([10,  3,  4])
13  >>> a # change of elemnet value is reflected in 'a' too
14  array([ 0,  1, 10,  3,  4,  5,  6,  7,  8,  9])
```

Since it is only a view, if an element of a slice is modified, the original array is modified as well. Although this facility can be desirable, it can create a nuisance in certain problems. Consequently, the function copy() provides a way out by copying the original array instead of providing a view:

```
 1  >>>a = numpy.arange(10) # defining array 'a'
 2  >>>c = a[2:5].copy() # array 'c' is created using a slice
    of 'a'
 3  >>> a # checking elements of array 'a'
 4  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
 5  >>>c # checking elements of array 'c'
 6  array([2, 3, 4])
 7  >>>c[0] = 10 # changing first element to 10
 8  >>>c # checking for change
 9  array([10,  3,  4])
10  >>>a # array 'a' remains unchanged
11  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
12  >>>numpy.may_share_memory(a,c) # 'a' remains unchanged
    because 'a' and 'c' don't share thier memories
13  False
```

6.12 Masking

Arrays can be indexed using the method of *masking*. Masking is a way to define the indexes as a separate object and then generate a new array from the original array using the mask as a rule. There are two ways that arrays can be masked: fancy indexing and indexing using boolean values. It is important to note that masking methods generate copies instead of views. The two methods are discussed in the following subsections.

6.12.1 Fancy Indexing

numpy offers quite unique indexing facilities. One of them is fancy indexing where an array of indexes can be used to generate an array of elements:

```
1  >>>a = numpy.arange(1000)**3# Generated cubes of first 1000
   cubes
2  >>>i = numpy.array(numpy.arange(10)) # Generated an array
   of first 10 numbers starting from 0 upto 9
3  >>> i
4  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
5  >>>a[i] # those elements of array 'a' which has indices of
   elemental values if array 'i'
6  array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
7  >>>j = numpy.array(numpy.arange(0,50,10)) # j is an array
   of numbers from 0 to 50 with steps of 10
8  >>>j
9  array([ 0, 10, 20, 30, 40])
10 >>>a[j] # a[j] is the array of cubes indexed with array j
11 array([    0,  1000,  8000, 27000, 64000])
12 >>>k = numpy.array( [ [ 1, 2], [ 11, 12 ] ] ) # k is a two
   dimensional array of indexes 1,2,11,12
```

```
13  >>>a[k] # a[k] is array made up of elements placed at
    indexes given by k
14  array([[   1,    8],
15       [1331, 1728]])
```

6.12.2 Indexing with Boolean Arrays

Indexing using integers specifies the position of the element. By using fancy indexing, we can pick up those particular elements. This is done with a different philosophy when using boolean data type for indexing. Here the boolean value True means that an array should become part of the final array and the value False indicates that the element should not become part of the array:

```
 1  >>>a = numpy.arange(100).reshape(10,10) # a is a 10 X 10
    matrix of first hundred numbers. Our aim is to make an
    array of even numbers and make a 5 X 10 matrix
 2  >>> a
 3  array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
 4  [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
 5  [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 6  [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
 7  [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
 8  [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
 9  [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
10  [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
11  [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
12  [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
13  >>>b = (a % 2 ==0) # b now stores a matrix of boolean
    values where the value is 'True' when the element of a is
    divisible by two and value is 'False' when elemnet of a is
    not divisible by two.
```

```
14  >>>a[b].reshape(5,10)
15  array([[ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18],
16  [20, 22, 24, 26, 28, 30, 32, 34, 36, 38],
17  [40, 42, 44, 46, 48, 50, 52, 54, 56, 58],
18  [60, 62, 64, 66, 68, 70, 72, 74, 76, 78],
19  [80, 82, 84, 86, 88, 90, 92, 94, 96, 98]])
```

6.13 Arrays Are Not Matrices

Even though the Python object `ndarray` (or simply `arrays`) looks like defining matrices, they are not the same. Arrays perform elementwise operators well, but they do not reflect the same behavior as that of matrix operations defined in linear algebra.

Matrix objects can be defined using `numpy.matrix`. A `numpy.matrix` is a specialized 2-D array that retains its 2-D nature through operations. Certain special operators, such as * (matrix multiplication) and ** (matrix power) are defined for them. The following Python code will demonstrate this behavior:

```
1  >>>a = numpy.arange(10).reshape(2,5) # defining a 2 X 5
   array made of numbers from first ten numbers
2  >>>a
3  array([[0, 1, 2, 3, 4],
4      [5, 6, 7, 8, 9]])
5  >>>type(a) # 'a' is a numpy.ndarray
6  <class 'numpy.ndarray'>
7  >>># Now we shall create a matrix using this array
8  >>>b = numpy.matrix(a) # 'b' references a matrix object
9  >>> type(b) # 'b' is a matrix unlike 'a', which is an array
10  <class 'numpy.matrixlib.defmatrix.matrix'>
```

```
11  >>>b
12  matrix([[0, 1, 2, 3, 4],
13      [5, 6, 7, 8, 9]])
```

Mathematical operations like scalar multiplication, matrix multiplication (dot and cross product), and matrix power are defined for this data type:

```
 1  >>>a_array = numpy.arange(12).reshape(3,4)
 2  >>>a_array # 'a_array' stores a 3 X 4 array of numbers
 3  array([[ 0,  1,  2,  3],
 4      [ 4,  5,  6,  7],
 5      [ 8,  9, 10, 11]])
 6  >>>a_array_1 = a_array.copy # 'a_array_1' is a copy of
    'a_array'
 7  >>>sum_array = a_array + a_array # sum of two arrays
    produces a new array where elementwise operation (addition
    here) is performed
 8  >>>sum_array
 9  array([[ 0,  2,  4,  6],
10      [ 8, 10, 12, 14],
11      [16, 18, 20, 22]])
12  >>>scalar_product = 3 * a_array # scalar product of array
    with a number is simply elementwise multiplication
13  >>>scalar_product
14  array([[ 0,  3,  6,  9],
15      [12, 15, 18, 21],
16      [24, 27, 30, 33]])
17  >>>a_matrix = numpy.matrix (a_array) # A matrix 'a_matrix'
    is created using an array 'a_array'
18  >>>a_matrix
19  matrix([[ 0,  1,  2,  3],
```

```
20      [ 4,  5,  6,  7],
21      [ 8,  9, 10, 11]])
22  >>>sum_matrix = a_matrix + a_matrix
23  >>>sum_matrix
24  matrix([[ 0,  2,  4,  6],
25      [ 8, 10, 12, 14],
26      [16, 18, 20, 22]])
27  >>>scalar_mul_matrix = 3 * a
28  >>>scalar_mul_matrix = 3 * a_matrix
29  >>>scalar_mul_matrix
30  matrix([[ 0,  3,  6,  9],
31      [12, 15, 18, 21],
32      [24, 27, 30, 33]])
33  >>># Checking for transpose
34  >>>a_array_T = a_array.T # Transpose of array i.e rows and
    copy are exchanged
35  >>>a_array_T
36  array([[ 0,  4,  8],
37      [ 1,  5,  9],
38      [ 2,  6, 10],
39      [ 3,  7, 11]])
40  >>>a_matrix_T = a_matrix.T # Transpose of matrix object
41  >>>a_matrix_T
42  matrix([[ 0,  4,  8],
43      [ 1,  5,  9],
44      [ 2,  6, 10],
45      [ 3,  7, 11]])
46  >>># checking for dot product of arays and matrices
47  >>>dot_array = numpy.dot(a_array, a_array_T)
48  >>>dot_array
49  array([[ 14,  38,  62],
```

```
50      [ 38, 126, 214],
51      [ 62, 214, 366]])
52  >>>dot_matrix = numpy.dot(a_matrix, a_matrix_T)
53  >>>dot_matrix
54  matrix([[ 14,  38,  62],
55      [ 38, 126, 214],
56      [ 62, 214, 366]])
```

Until now, the matrix has behaved exactly the same as an array. A common question that arises in the minds of programmers is that if you had an array object, what was the need of matrix ?

The answer is quite complex. While array serves most of the general purposes for matrix algebra, matrix is written to facilitate linear algebra functionalities. Linear algebra is performed using a submodule of numpy accessed as numpy.linalg. Issuing the command help(numpy.linalg) gives an idea about the purpose of this module. Following are some of the useful functions from the matrix algebra point of view:

- solve(): to solve system of linear equations

- norm(): to find norm of matrix

- inv(): to find matrix inverse of a square matrix

- pinv(): to find pseudo-inverse of any matrix (nonsquare matrices too)

- matrix_power(): to perform an integer power of a square matrix

To perform linear algebra calculations using matrices, it is suggested that the matrix object is used to avoid errors.

More information about the matrix object can be found by issuing the command help(numpy.matrix) or visiting reference [3].

6.14 Some Basic Operations

array allows some basic built-in operations that come in quite handy while performing calculations. These functions have been written to optimize time spent on running the code and minimizing error. As a result, users can concentrate on using them for computation rather than writing their own and then optimizing them. Some of them are discussed in the following subsections.

6.14.1 sum

sum() calculates the sum of all elements in the array if it is one-dimensional or else it calculates the sum of elements of a column:

```
 1  >>>a = numpy.arange(25) # created an array 'a' consisting
    of first 25 numbers
 2  >>>a
 3  array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,
    13, 14, 15, 16,
 4  17, 18, 19, 20, 21, 22, 23, 24])
 5  >>>sum(a) # sum of all elements
 6  300
 7  >>> b = a.reshape(5,5)
 8  >>> b
 9  array([[ 0,  1,  2,  3,  4],
10         [ 5,  6,  7,  8,  9],
11         [10, 11, 12, 13, 14],
12         [15, 16, 17, 18, 19],
13         [20, 21, 22, 23, 24]])
14  >>> sum(b)
15  array([50, 55, 60, 65, 70])
```

6.14.2 Minimum and Maximum

`min()` and `max()` gives the minimum and maximum value among the element values:

```
 1  >>>a = numpy.arange(10).reshape(2,5)
 2  >>>a
 3  array([[0, 1, 2, 3, 4],
 4      [5, 6, 7, 8, 9]])
 5  >>>a.min()
 6  0
 7  >>>a.max()
 8  9
 9  >>>a.max(axis=0) # maximum in each coloumn
10  array([5, 6, 7, 8, 9])
11  >>>a.max(axis=1) # maximum in each row
12  array([4, 9])
```

6.14.3 Statistics: Mean, Median, and Standard Deviation

`mean()`, `median()`, and `std()` find the mean, median, and standard deviation for the data stored in the array:

```
 1  >>>a = numpy.arange(10).reshape(2,5)
 2  >>>a
 3  array([[0, 1, 2, 3, 4],
 4      [5, 6, 7, 8, 9]])
 5  >>>numpy.mean(a) # mean of all values
 6  4.5
 7  >>>numpy.median(a) # median of ala value
 8  4.5
```

```
 9  >>>numpy.std(a) # standard deviation for all values
10  2.8722813232690143
```

6.14.4 sort()

sort() sorts the array values from maximum to minimum:

```
 1  >>> a = numpy.random.rand(3,4)
 2  >>> a
 3  array([[ 0.51389165,  0.65372872,  0.22264919,  0.41722141],
 4  [ 0.19754059,  0.78218154,  0.0830597 ,  0.44351731],
 5  [ 0.54915984,  0.49219332,  0.24111783,  0.5021453 ]])
 6  >>> numpy.sort(a)
 7  array([[ 0.22264919,  0.41722141,  0.51389165,  0.65372872],
 8  [ 0.0830597 ,  0.19754059,  0.44351731,  0.78218154],
 9  [ 0.24111783,  0.49219332,  0.5021453 ,  0.54915984]])
10  >>> numpy.sort(a, axis=1) # sorting along columns
11  array([[ 0.22264919,  0.41722141,  0.51389165,  0.65372872],
12  [ 0.0830597 ,  0.19754059,  0.44351731,  0.78218154],
13  [ 0.24111783,  0.49219332,  0.5021453 ,  0.54915984]])
14  >>> numpy.sort(a, axis=0) # sorting along rows
15  array([[ 0.19754059,  0.49219332,  0.0830597 ,  0.41722141],
16  [ 0.51389165,  0.65372872,  0.22264919,  0.44351731],
17  [ 0.54915984,  0.78218154,  0.24111783,  0.5021453 ]])
```

A variety of sorting algorithms exists. The choice of the algorithm can depend on the requirements for average speed, worst-case scenario, workspace size, and stability. numpy documentation at [4] lists three choices as follows:

Sorting an array of complex numbers is accomplished by sort_complex(). (See Table 6-2.)

Table 6-2. *Sorting Algorithms*

Kind	Speed	Worst Case	Workspace	Stable
'quicksort'	1	$O(n^2)$	0	no
'mergesort'	2	$O(n \times log(n))$	$\dfrac{n}{2}$	yes
'heapsort'	3	$O(n \times log(n))$	0	no

```
1  >>> a = numpy.array([4-3j, 4+5j, 3-8j])
2  >>> a
3  array([ 4.-3.j,  4.+5.j,  3.-8.j])
4  >>> numpy.sort_complex(a)
5  array([ 3.-8.j,  4.-3.j,  4.+5.j])
```

6.14.5 Rounding Off

Rounding off numbers is performed by the function around() with the same logic as in mathematics. (If a numeral is 5 or more, the preceding numeral is incremented by 1.)

```
1  >>> a = numpy.random.rand(10)
2  >>> a
3  array([ 0.15342238,  0.90475845,  0.90006375,  0.68768342,
   0.14153903,
4  0.07962712,  0.59819738,  0.76087737,  0.97013725,  0.1560855 ])
5  >>> numpy.around(a)
6  array([ 0.,  1.,  1.,  1.,  0.,  0.,  1.,  1.,  1.,  0.])
7  >>> numpy.around(a).astype(int)
8  array([0, 1, 1, 1, 0, 0, 1, 1, 1, 0])
```

6.15 asarray() and asmatrix()

A variety of variables is not defined as arrays, but if at a certain point of time during computation they need to be considered as an array or as matrix, asarray() and asmatrix() can be used:

```
 1  >>> (a,b,c,d) = (1,2,3,4)
 2  >>> array1 = numpy.asarray([a,b,c,d]) # defined as array
 3  >>> array1
 4  array([1, 2, 3, 4])
 5  >>> matrix1 = numpy.asmatrix([a,b,c,d]) # defined as a
    matrix
 6  >>> matrix1
 7  matrix([[1, 2, 3, 4]])
 8  >>> string = 'Hello world'
 9  >>> string
10  'Hello world'
11  >>> array2 = numpy.asarray(string) # defined as array
12  >>> array2 # indicates that data type is strong with 11
    characters
13  array('Hello world',
14  dtype='<U11')
```

6.16 Summary

Array-based computing is used as a primary force to solve equations and systems of equations. Using slicing and indexing operations, it provides powerful tools to manipulate data using a program. Since this book is an interactive text on Python, discussion about all functions for indexing and slicing is out of its scope.

Discussing all the facilities of array manipulations that are present in numpy is beyond the scope of any textbook. Moreover, new functionalities are added with each new version. However, some quite important functions have been discussed in this chapter.

Python's ability to flexibly create a variety of arrays and to compute using various mathematical functions makes it one of the most preferred languages in the field of computational physics and mathematics. Another fact that makes it a preferable programming language among members of the scientific community is its ability to plot publication-quality graphs with relative ease, which we will discuss in Chapter 7.

6.17 Bibliography

[1] https://en.wikipedia.org/wiki/Binomial_distribution.

[2] https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.random.html.

[3] http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html.

[4] http://docs.scipy.org/doc/numpy/reference/generated/numpy.sort.html.

CHAPTER 7

Plotting

7.1 Introduction

Plotting data is one of the most essential parts of numerical computation. In pre-processing, during computation, and in post-processing, the ability to plot data on different kinds of graphs is essential. Visualization of data in a convenient format allows for better understanding of the process. Visual clues generate a lot of information about the process that generated that particular data. You can easily look for errors and derive simple as well as complex interpretations. A good programming language must incorporate facilities to plot data easily. Plotting two-dimensional (2-D) and three-dimensional (3-D) graphs is vital for creating a good visualization product. Python users have a number of choices in this regard.

In this chapter, we will discuss some of the choices. Essential requirements while choosing a plotting library depends on requirements of the data, such as the following:

- Plotting on 2-D or 3-D graphs

- Plotting live data or static data

- Plotting large data quickly

- Saving plots in a variety of formats

- Plotting data with a chosen resolution to keep a check on file size

The plotting library used for this chapter is called `matplotlib`. Just like any other module, it they can be installed using `pip` on an Ubuntu system. For Windows and Mac OS X users who use Anaconda IDE, this library is pre-installed in the system.

7.2 matplotlib

John Hunter, the creator of `matplotlib` [1], rightly stated that **"`matplotlib` tries to make easy things easy and hard things possible [2]."**

In some cases, with just one line of code, you can generate a high-uality, publication-ready graphics visualization of the problem at hand. Before Python, `gnuplot` was used to plot the data passed by a Python script. With `matplotlib` available, this action has become very flexible. `matplotlib` was modeled after the graphic capabilities of MATLAB, which came as a boon for programmers who were already well versed with MATLAB. Some of the major advantages of using `matplotlib` over other plotting libraries are as follows:

- It is integrated with LaTeX markup.

- It is cross-platform and portable.

- It is open sourced so users don't have to worry about license fees.

- Being part of Python, it is programmable.

`matplotlib` stands for *mathematical plotting library*. It is one of the most popular plotting libraries among programmers owing to its simple and intuitive commands as well as its ability to produce high-quality plots that can be saved in variety of formats. It supports both interactive and noninteractive modes of plotting and can save images in a variety of formats including JPEG, PS, PDF, and PNG. It can utilize a variety of window toolkits like GTK+, wxWidgets, and QT. The most attractive feature

is that it has a variety of plotting styles such as line, scatter, and bar charts as well as histograms and many more. It can also be used interactively with IPython. `numpy` is necessary for working with `matplotlib`. Hence, it must be installed on the system before you can work with `matplotlib`.

`matplotlib` depends on a certain set of programs for proper executions. These dependencies must be first installed [3]. This installation is automatically done when `pip` is used, but users who wish to build it from source must perform the same. This book is intended for beginners, so this topic is not illustrated here.

7.2.1 pylab vs. pyplot

Within `matplotlib`, `pyplot` and `pylab` are the two most discussed modules inside `matplotlib` that provide almost the same functionalities and, thus, cause some confusion about their usage. As a result, it is important to differentiate between them at this point.

Within the package `matplotlib`, two subpackages, namely `matplotlib.pylab` and `matplotlib.pyplot`, exist. Since plotting can start as a simple exercise and then become a quite complicated one, `matplotlib` is designed in a hierarchical pattern where, by default, simple functions are implemented in the `matplotlib.pylab` environment. As the complexity increases, a more complex environment like `matplotlib.pyplot` is implemented.

A more object-oriented approach is `matplotlib.pyplot`, where functions like `figures()`, `axes()`, and `axes()` are defined as objects to keep track of their properties dynamically. For even more complex tasks like making graphic user interfaces (GUI), exclusive `pyplot` usage can be dropped altogether and an object-oriented approach can be used to fabricate plots. Also, when plots are mostly noninteractive, the `pyplot` environment can be used. `pylab` is used for interactive studies.

7.3 Plotting Basic Plots

We will first explore the working environment offered by `matplotlib.pyplot`. Functions inside `pyplot` control a particular feature of the plot like putting up a tile, mentioning labels on the x-axis and y-axis, putting mathematical equations on the body of plot at a desired position, defining tick labels, defining types of markers to plot a graph, and so on. `pyplot` is stateful; it keeps updating the changes in the state of figure once defined. This makes it easier to modify a graph until the desired level is reached before including the code in the program. The `plot()` function is used for plotting simple 2-D graphs. It takes a number of arguments that fill data and other feature information to plot a graph. In its simplest form, it can plot a list of numbers. (See Listing 7-1.)

Listing 7-1. sqPlot0.py

```
1  # Python code for plotting numbers
2  import numpy as np
3  import matplotlib.pyplot as plt
4  a = np.arange(10)
5  plt.plot(a)
6  plt.show()
```

It is important to note that libraries/modules `numpy` and `matplotlib.pyplot` are used with an alias: `np` and `plt`. This is usually preferred by programmers to avoid writing the name of the library each time they need to use its functions. We will use this convention in this chapter.
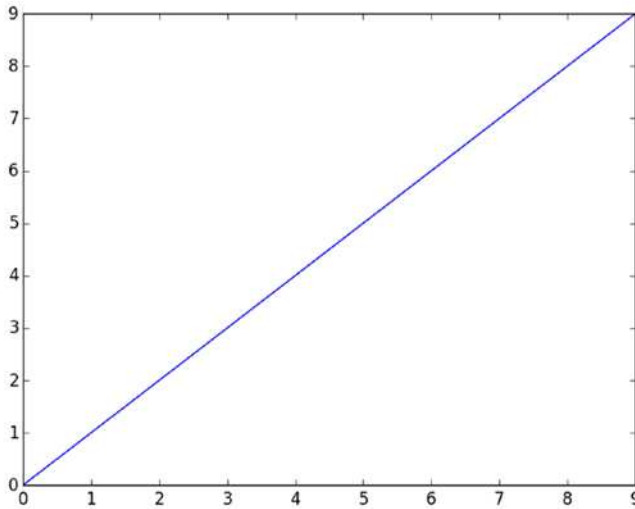
Figure 7-1. *Plotting the first 10 integers using plot() function*

The result can be seen in Figure 7-1. A plot needs two axes, which are usually termed as the x-axis and y-axis. When the plot() command is supplied with a single list or array, it assumes it to be the values for the y-axis and automatically generates corresponding x values, taking its cue from the length of list. We had 10 numbers. Consequently, the x-axis had 10 numbers from 0 to 9. plot() can take both axes as input vectors to produce a plot as shown in the code in Listing 7-2. This code uses the pylab library instead of pyplot.

Listing 7-2. sqPlot1.py

```
1  # Plotting number using pylab
2  import numpy as np
3  from matplotlib import pylab as pl
4  x = np.linspace(0,100)
5  y = x ** 2
6  pl.plot(x,y)
7  pl.show()
```

The result can be seen in Figure 7-2. A very basic plot could be plotted by just a few lines of code where you first import relevant libraries (line 1 and line 2), then define the x- and y-axes, and finally use the `plot` command, which gives the parameters for the x- and y-axes. These commands can be issued one-by-one at the Python command prompt, or they can be saved as a Python file. (Use a text editor, write the code, and save with `sqPlot1.py`.)
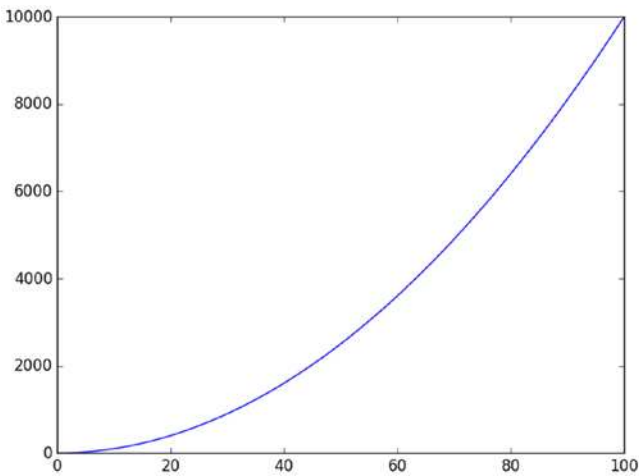
Figure 7-2. *Plotting the first 10 integers using plot() function*

The axes are defined as a `numpy` array. They can be generated by all the methods available including creating them by hand, using a formula (like in code `sqPlot1.py`—array named `y` is generated by elementwise squaring of array `x`), using a data file, taking data live from a remote/local server using the Internet or LAN, and so on. Subsequent chapters will deal with file input/output facilities. In this chapter, only an array generated by self or using formulas will be used.

It is worth mentioning the role of a seemingly simple but powerful function: show(). A simple search at the command prompt (write matplotlib.pyplot.show) declares its purpose. It displays figure(s) on a computer terminal having graphics capability, which most modern computers do. In the noninteractive mode in an IPython console, it first displays all figures and blocks the console until the figures have been closed. Conversely, in the interactive mode, it does not block the console. Both modes have their own merits. The interactive mode is used for checking the change in features. The noninteractive mode is used when the focus is more upon the code generation that produces the graphs. Usually, programmers work with the interactive mode and optimize a view plots and then work with the noninteractive mode taking the same settings that were generated during experiments with the interactive mode.

7.3.1 Plotting More than One Graph on Same Axes

More than one plot on the same axes can be plotted in the same figure by simply issuing two plot commands, as shown in Listing 7-3.

Listing 7-3. sqPlot2.py

```
1  import numpy as np
2  from matplotlib import pylab as pl
3  x = np.linspace(0,100)
4  y1 = x ** 2 # y is square of x
5  y2 = x ** 2.2 # y is x raised to power 2.2
6  pl.plot(x,y1)
7  pl.plot(x,y2)
8  pl.show()
```
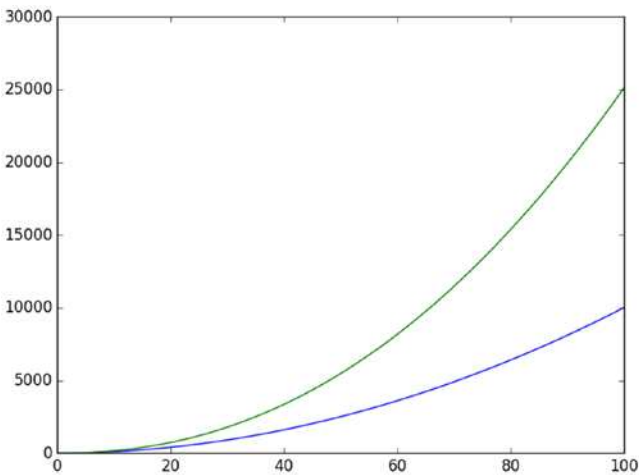
Figure 7-3. *Plotting more than one graph on the same axes*

7.3.2 Various Features of a Plot

A variety of features exists for a graph. Following is a list of features of a graph:

- **Title**: The title gives a short introduction for the purpose of the graph.

  `title()` object sets the title of the current axes, positioned above axes and in the center. It takes a string as an input.

- **Labels for axes**: Labels marks the purpose of graph axes.

  `xlabel()` and `ylabel()` objects set the label of the x- and y-axis, respectively. The text string, which it takes as input, is positioned above the axis in the center.

- **Ticks**: Ticks on axes show the division of data points on an axis and help judge information about a data point on graph.

  `xticks` and `yticks` set the ticking frequency and location. For example,

  `xticks( arange(5), ('a', 'b', 'c', 'd', 'e') )`

shows that five ticks named a, b, c, d, and e are placed equidistant. `linspace` and `logspace` can also be used for the same.

- **Markers**: Markers are the symbols drawn at each data point. The size and type of markers can be differentiated for showing the data points belonging to two or more different data sets.

  In the `plot()` function, for every pair *x, y,* there is an optional third argument as a format string that indicates the color and line type of the plot. A list of markers is given at http://matplotlib.org/api/markers_api. html#module-matplotlib.markers. For example: `plot(x,y,'r+')` means that red plus signs (+) will be placed for each data point. They are tabulated in Table 7-1.

- **Line width**: Line width defines the width of markers. `linewidth=n` where n can be set as an integer, setting the marker size to a desired dimension.

- **Line style**: Line style defines the style of lines that connect the markers. They can be set off when data points don't need to be connected.

  `linestyle = '.'` sets the line style as a connecting dot between two data points. Similarly, a number of other line styles also exists as tabulated in Table 7-2.

- **Color**: The color of markers can also be used for distinguishing data points belonging to two or more different data sets, but this method cannot be used where data needs to be published in a black-and-white color scheme.

Table 7-1. *Plot Features*

Marker Abbreviation	Marker Style
.	Point
,	Pixel
o	Circle
v	Triangle down
<	Triangle left
>	Triangle right
1	Tripod down
2	Tripod up
3	Tripod left
4	Tripod right
s	Square
p	Pentagon
*	Star
h	Hexagon
H	Rotated hexagon
+	Plus
x	Cross
D	Diamond
d	Thin diamond
-	Horizontal line

Table 7-2. *Plot Line Styles*

Style Abbreviation	Style
-	Solid line
–	Dashed line
-.	Dash dot line
:	Dotted line

The following command sets the line style as -- and markers as + in green:

```
plot(arange(10,100,1), linestyle='--', marker='+', color='g')
```

A shortcut command would have been the following:

```
plot(range(10), '--g+')
```

Table 7-3 lists the codes for choosing a particular color.

Table 7-3. *Colors for Plotting*

Color Abbreviation	Color Name
b	Blue
c	Cyan
g	Green
k	Black
m	Magenta
r	Red
w	White
y	Yellow

In addition to using the predefined symbols for colors, you can also use a hexadecimal string such as #FF00FF or a RGBA tuple like (1,0,1,1). You can set grayscale intensity as a string like '0.6'.

- **Grid**: Grids can be turned off or an for a graph using the following syntax:

  ```
  grid(True)
  ```

- **Legends**: Legends are used to differentiate between types of data points from multiple graphs in the same figure by showing symbols for the data type and printing text for the same.

  Their usage is illustrated at

  http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.legend

  By default, legend() takes input as the string provided within the plot() function under the flag label=". The location is set to be top-right corner by default. It can be changed as required by setting loc= argument.

A variety of options [4] are presented for 2-D line styles for setting the line properties for a 2-D plot() functions. Code sqPlot3.py shows a plotting of a formatted figure using a variety of formatter arguments, as shown in Listing 7-4 and Figure 7-4.

Listing 7-4. sqPlot3.py

```
1  # import pylab and numpy
2  from matplotlib import pylab as pl
3  import numpy as np
4
5  # Create a figure of size 9x7 inches, 100 dots per inch
6  pl.figure(figsize=(9, 7), dpi=100)
```

```
7
8  # Create a new subplot from a grid of 1x1
9  pl.subplot(1, 1, 1)
10
11 # We wish to plot sin(x) and sin(2x)
12
13 # first we define x- axis in terms of pi units
14
15 X = np.linspace(-np.pi * 2, np.pi *2, 10e4, endpoint=True)
16
17 ''' x-axis is defined from -2pi to 2pi with 10000 points
18 where last point is also included'''
19
20 S, S2 = np.sin(X), np.sin(2*X)
21
22 # Plot sin(x) with a blue continuous line of width 1
   (pixels)
23 # labelled as sin(x)
24 pl.plot(X, S, color="blue", linewidth=1.0, linestyle="-",
   label = "$sin(x)$")
25
26 # Plot sine(2x) with a red continuous line of width 1
   (pixels)
27 # labelled as sin(2x)
28 pl.plot(X, S2, color="red", linewidth=1.0, linestyle="-",
   label = "$sin(2x)$")
29
30 # Set x limits from -2pi to 2.5*pi
31 pl.xlim(-2* np.pi, 2.5* np.pi)
32
33 # Set x ticks
34 pl.xticks(np.linspace(-2.5 * np.pi, 2.5 * np.pi, 9,
   endpoint =True))
```

```
35
36  # Set y limits from 1.2 to -1.2
37  pl.ylim(-1.2, 1.2)
38
39  # Set y ticks
40  pl.yticks(np.linspace(-1, 1, 5, endpoint=True))
41
42  # Set the tile as 'Sine waves'
43  pl.title('$sin(x)$ and $sin(2x)$ waves')
44
45  # Setting label on x-axis and y-axis
46
47  pl.ylabel('$sin(x)$ and $sin(2x)$')
48  pl.xlabel('$x$')
49
50  # Setting the grid to be ON
51  pl.grid(True)
52
53  # To show a legend at one corner for differentiating two
    curves
54  pl.legend()
55
56  # Show result on screen
57  pl.show()
```
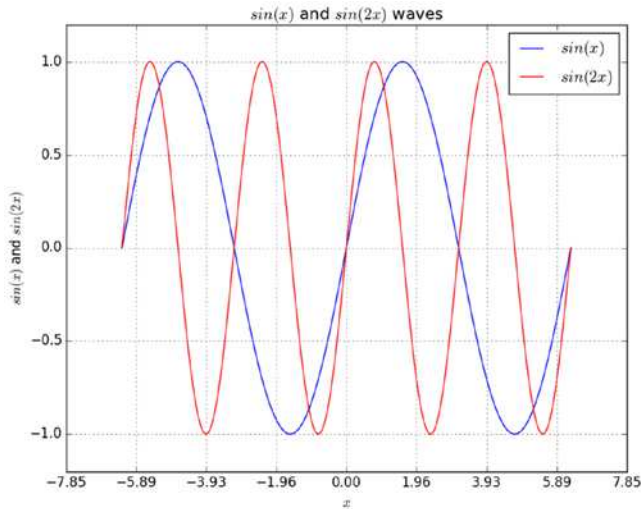
Figure 7-4. *Plotting sin(x) and sin(2x)*

7.4 Setting Up to Properties

The setup() and getup() objects allow us to set and get properties of objects. They work well with matplotlib objects. For the object plot(), setup() can be used to set the properties. For now, the beginner can ignore this.

7.5 Histograms

Histograms use vertical bars to plot events occurring with a particular range of frequency (called bins). They can be simply plotted using the hist() function, as shown in Listing 7-5.

Listing 7-5. plotHistogram.py

```
1  import matplotlib.pyplot as pt
2  import numpy as np
3
4  a = np.random.rand(50)
5  pt.hist(a)
6  pt.show()
```

The result is shown in Figure 7-5. It is important to note that since we used random numbers as input to the `hist()` function, a different plot will be produced each time.
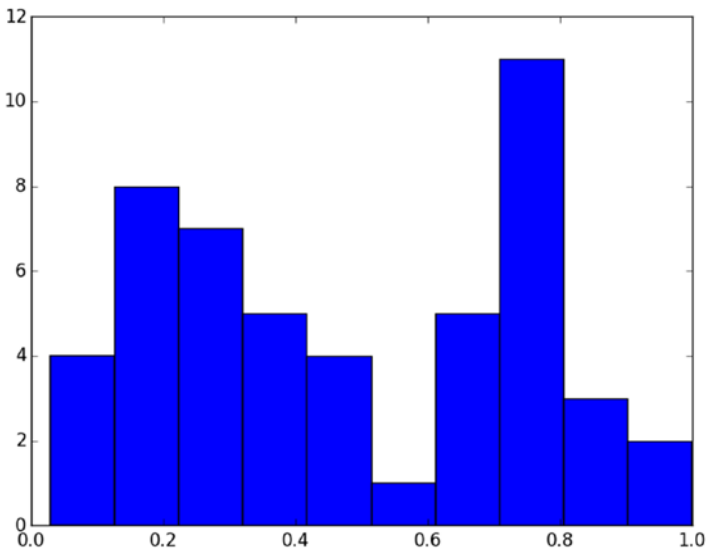
Figure 7-5. *Histogram plot for 50 random numbers*

The umber of bins can be set to a number along with input variable, as shown in Listing 7-6.

Listing 7-6. plotHistogramBins.py

```
1  import matplotlib.pyplot as pt
2  import numpy as np
3
4  a = np.random.rand(50)
5  pt.hist(a,25) # setting number of bins to 25
6  pt.show()
```

The result is shown in Figure 7-6.
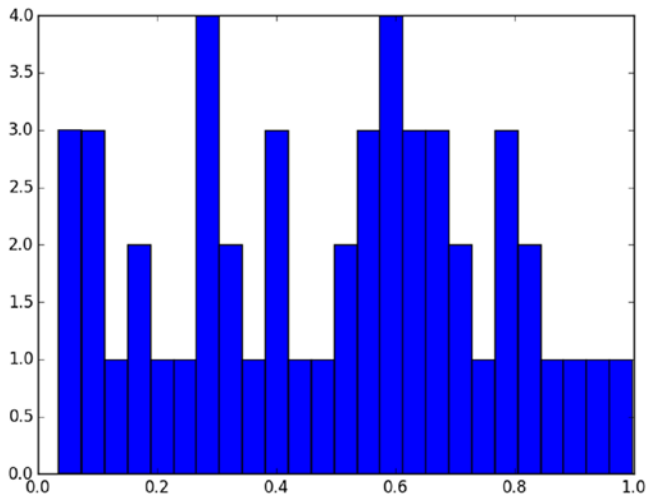
Figure 7-6. *Histogram plot for 50 random numbers*

7.6 Bar Charts

One of the simplest plots employs rectangular bars (either horizontally or vertically) where the height of the rectangle is proportional to the data value. This kind of graph is called a bar chart. Bar graphs are generated by the bar() function, which takes two inputs for defining the x-axis and y-axis, as opposed to the hist() function, which takes only one input. A sample code is presented in Listing 7-7, where the x and y arrays are defined.

Listing 7-7. bar.py

```
1  import matplotlib.pyplot as pl
2  import numpy as np
3  x = np.array([1,2,3,4,5,6,7,8,9,0])
4  y = np.array([1,4,2,3,4,5,7,6,8,7])
5  pl.bar(x, y)
6  pl.title('Vertical Bar chart')
7  pl.xlabel('$x$')
8  pl.ylabel('$y$')
9  pl.show()
```

The result is shown in Figure 7-7.
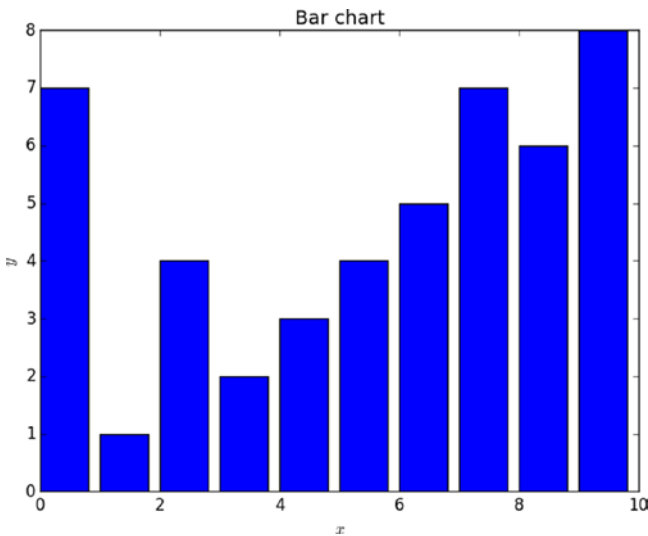
Figure 7-7. *Vertical bar chart*

Bar charts and histograms look very similar. The difference lies in the way you define them. Whereas `bar()` requires both x-axis and y-axis arguments, `hist()` requires only an y-axis argument. `barh()` function plots horizontal bars. (See Listing 7-8.)

Listing 7-8. barh.py

```
1  import matplotlib.pyplot as pl
2  import numpy as np
3  x = np.array([1,2,3,4,5,6,7,8,9,0])
4  y = np.array([1,4,2,3,4,5,7,6,8,7])
5  pl.barh(x, y)
6  pl.title('Horizontal Bar chart')
7  pl.xlabel('$x$')
8  pl.ylabel('$y$')
9  pl.show()
```

The result is shown in Figure 7-8.
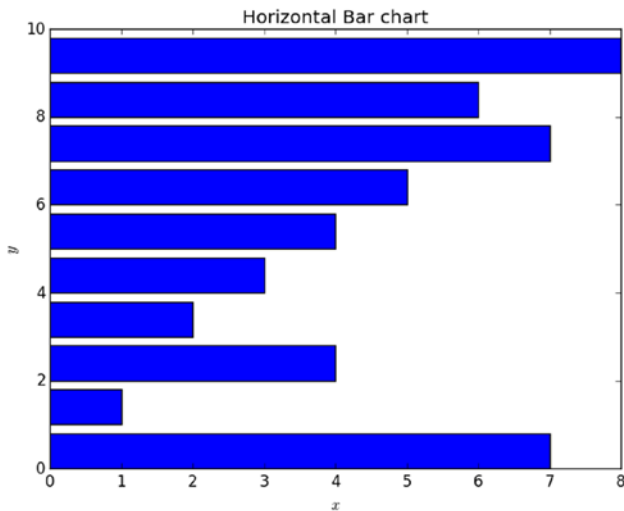
Figure 7-8. *Horizontal bar chart*

151

7.7 Error Bar Charts

All of experimental scientific work involves errors, and it is important to plot errors along with the data for many reasons. Errors must be plotted so that you can judge data quality and understand the regions of data where error is huge or minuscule. In `matplotlib`, the `errorbar()` function enables you to create such graphs called *error bar charts*. The representation of the distribution of data values is done by plotting a single data point, (commonly) the mean value of the dataset, and an error bar to represent the overall distribution of data. To accomplish this, the code `ploterror.py` shown in Listing 7-9 defined an array for the x-axis and then defined an array for the y-axis using the formula $y = x2$. Error is stored in the third array saved with the variable named `err`. These variable names are passed as arguments to the `errorbar()` function; hence, it takes three variables in a sequence as x, y, and `err`. The file produces the graph shown in Figure 7-9.

Listing 7-9. ploterror.py

```
 1  import matplotlib.pyplot as pl
 2  import numpy as np
 3  x = np.arange(0, 4, 0.2) # generated data point from 0 to 4
    with step of 0.2
 4  y = x*2 # y = e^(-x)
 5  err = np.array([0,.1,.1,.2,.1,.5,.9,.2,.9,.2,.2,.2,.3,.2,
    .3,.1,.2,.2,.3,.4])
 6  pl.errorbar(x, y, yerr=err, ecolor='r')
 7  pl.title('Error bar chart with symmetrical error')
 8  pl.xlabel('$x$')
 9  pl.ylabel('$y$')
10  pl.show()
```
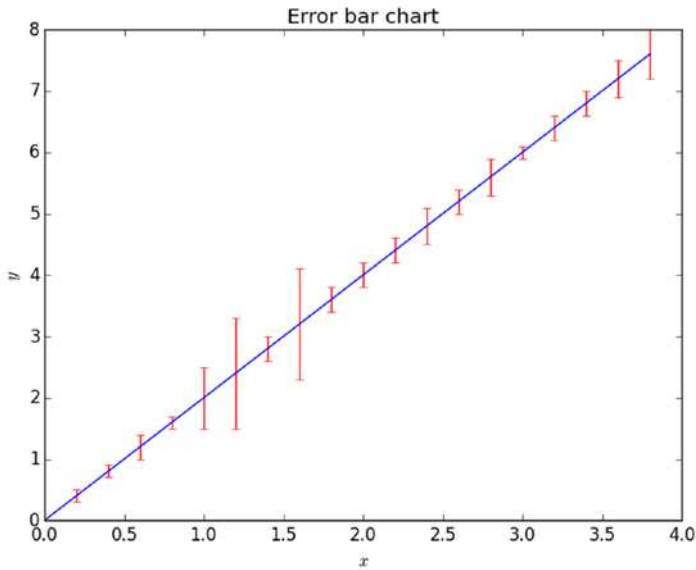
Figure 7-9. *Symmetrical error bar chart*

ecolor sets the color for error bars. Just as you can the value for yerr keyword, you can also set xerr to produce error bars on the x-axis.

In Listing 7-10, error bars are symmetrical—that is, positive and negative errors are equal. For plotting asymmetrical error bars, errorbar() would incorporate two arrays for error definition.

Listing 7-10. ploterror1.py

```
1  import matplotlib.pyplot as pl
2  import numpy as np
3  x = np.arange(0, 4, 0.2) # generated data point from 0 to 4
   with step of 0.2
4  y = x*2 # y = e^(-x)
5  err_positive = np.array([0.5,.1,.1,.2,.1,.5,.9,.2,.9,.2,
   .2,.2,.3,.2,.3,.1,.2,.2,.3,.4])
6  err_negative = np.array([0.2,.4,.3,.1,.4,.3,.1,.9,
   .1,.3,.5,.0,.5,.1,.2,.6,.3,.4,.1,.1])
```

```
 7  pl.errorbar(x, y, yerr=[err_positive, err_negative],
    ecolor='r')
 8  pl.title('Error bar chart with Asyymetric error')
 9  pl.xlabel('$x$')
10  pl.ylabel('$y$')
11  pl.show()
```

The file produces the graph shown in Figure 7-10.
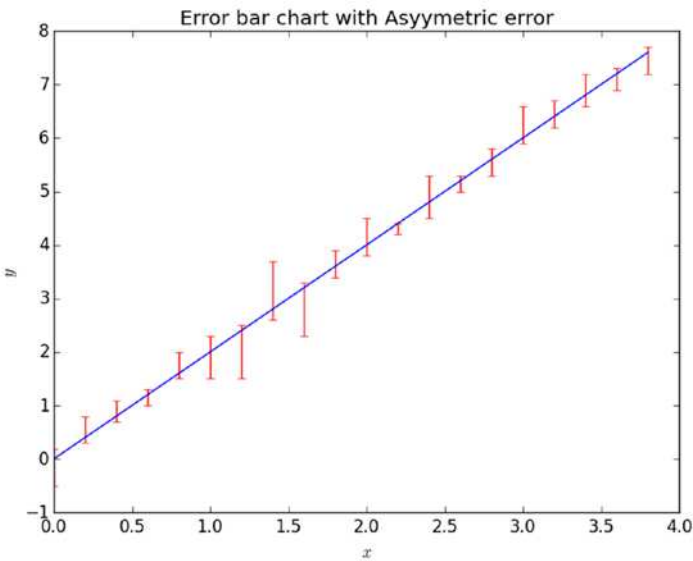
Figure 7-10. *Asymmetrical error bar chart*

7.8 Scatter Plots

Scatter plots consist of points plotted on a 2-D mesh (made of two axes, say *x* and *y*). The data aren't connected with lines; thus, they look scattered and unconnected!

These plots are achieved by the scatter() function, which takes two arrays as arguments. Scatter plots are used to get a correlation between two variables. When plotted, the clusters show a strong correlation

between particular data ranges. This is one of the key actions required by regression analysis. (See Listing 7-11.)

Listing 7-11. scatter.py

```
1  import matplotlib.pyplot as pl
2  import numpy as np
3  x = np.random.rand(1000)
4  y = np.random.rand(1000)
5  pl.scatter(x,y)
6  pl.title('Scatter Chart')
7  pl.xlabel('$x$')
8  pl.ylabel('$y$')
9  pl.show()
```

The file produces the graph shown in Figure 7-11.
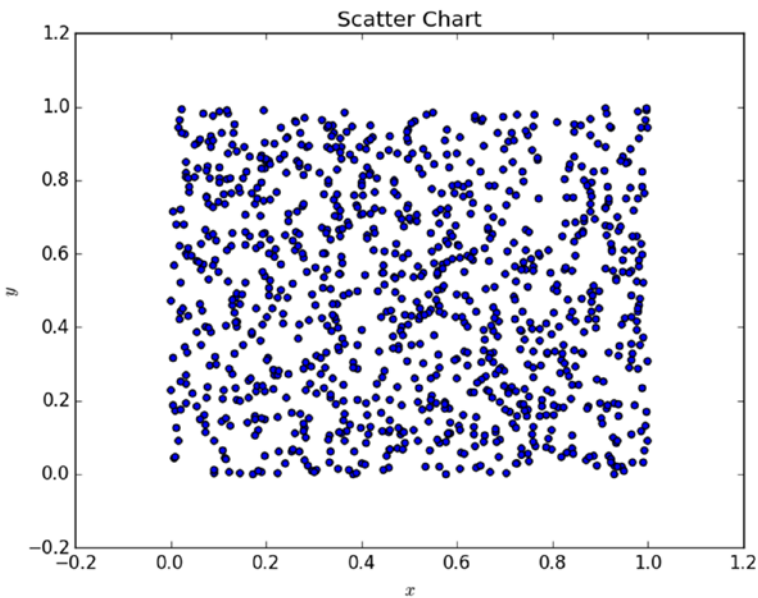
Figure 7-11. *Scatter plot*

7.9 Pie Charts

When data need to be categorized into sectors for a number of events in a particular range, pie charts come in handy. Pie charts are circular shapes where sectors/wedges are carved out for different data ranges. The size of a wedge is proportional to the data value. The `pie()` function works in this regard, as shown in Listing 7-12.

Listing 7-12. pie.py

```
1  import matplotlib.pyplot as pl
2  import numpy as np
3  x = np.array([1,2,3,4,5,6,7,8,9,0])
4  label = ['a','b','c','d','e','f','g','h','i','j']
5  explode = [0.2, 0.1, 0.5, 0, 0, 0.3, 0.3, 0.2, 0.1,0]
6  pl.pie(x, labels=label, explode = explode, shadow=True,
   autopct ='%2.2f%%')
7  pl.title('Pie Chart')
8  pl.show()
```

The file produces the graph shown in Figure 7-12.
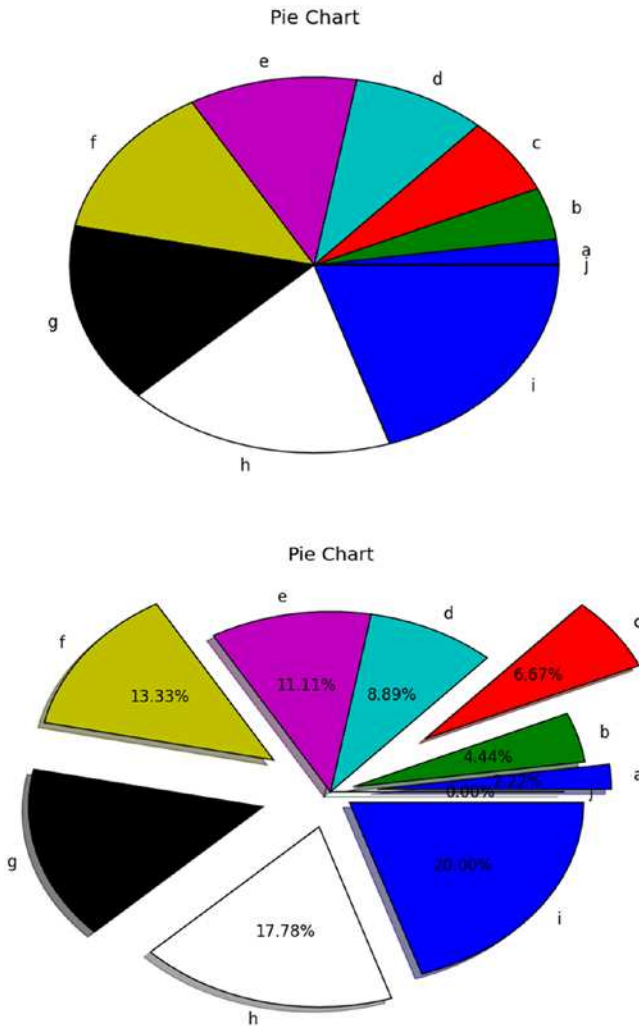
Figure 7-12. *Unexploded and exploded pie chart*

When the explode label is not defined, you get an unexploded pie chart. autopct sets the percentage of weight for a particular weight, which can be set by format specifier. %2.2f%% sets the display of percentage weights up to two decimal places with two significant digits. shadow provides a shadow below the wedge so that it looks like a real pie!

7.10 Polar Plots

Until now, all the plots we have discussed have been dealing with data defined in a Cartesian system. For data defined in a polar system—(r, θ) instead of (x, y)—polar plots are obtained by the plot() function, as shown in Listing 7-13.

Listing 7-13. polar.py

```
1  import matplotlib.pyplot as pl
2  import numpy as np
3
4  r = np.arange(0, 10.0, 0.1)
5  theta = 2* np.pi * r
6
7  pl.polar(theta, r, color ='g')
8  pl.show()
```

The file produces the graph shown in Figure 7-13.
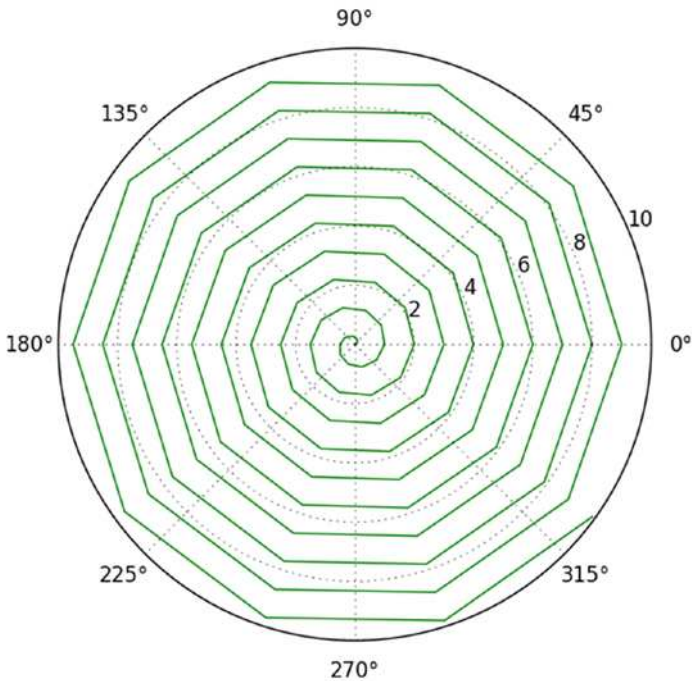
Figure 7-13. *Polar plot*

7.11 Decorating Plots with Text, Arrows, and Annotations

Sometimes, you are required to put text at a specific place on the plot (say coordinated (*x, y*)). The text() function is used to place text, as shown in Listing 7-14.

Listing 7-14.

```
1  import matplotlib.pyplot as pl
2  import numpy as np
3  x = np.arange(0, 2*np.pi, .01)
4  y = np.sin(x)
5  pl.plot(x, y, color = 'r');
```

159

```
 6  pl.text(0.1, -0.04, '$sin(0) = 0$')
 7  pl.text(1.5, 0.9, '$sin(90) = 1$')
 8  pl.text(2.0, 0, '$sin(180) = 0$')
 9  pl.text(4.5, -0.9, '$sin(270) = -1$')
10  pl.text(6.0, 0.0, '$sin(360) = 1$')
11  pl.annotate('$sin(theta)=0$', xy=(3, 0.1), xytext=(5, 0.7),
        arrowprops=dict(facecolor='green', shrink=0.05))
12  pl.title('Inserting text and annotation in plots')
13  pl.xlabel('$theta$')
14  pl.ylabel('$y = sin( theta)$')
15  pl.show()
```

The file produces the graph shown in Figure 7-14.
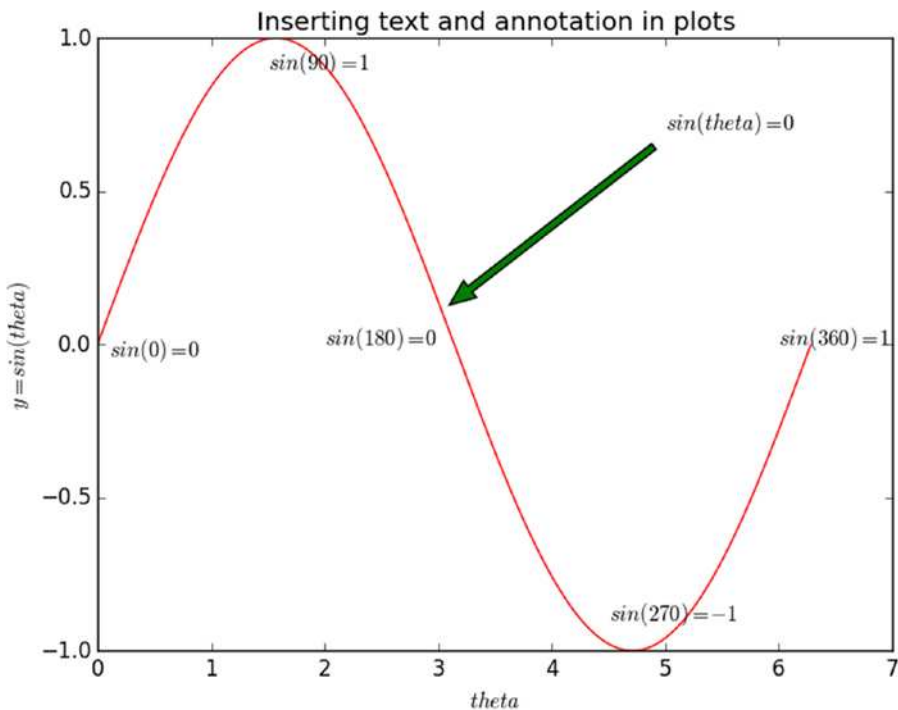
Figure 7-14. *Inserting text in the plot*

As seen in Listing 7-14, the text, arrow, and annotations can be placed at appropriate places by defining coordinate axis points for them. A convenient method of identifying appropriate coordinates is to roll over the mouse on the body of the plot and look for the down-left corner of the figure where present mouse coordinates are shown. `help(matplotlib.pyplot.annotate)` gives useful inputs to use this function.

7.12 Subplots

Multiple plots can be plotted using the subplot option where different plots are considered to be a matrix of graphs. Just like a regular matrix, elements are identified by index. As seen in Listing 7-15, subplots are located using indices like (222), which essentially means that the matrix is a 2 × 2 and one is accessing the second position to place the `scatter()` function-based plot. Similarly (221) uses a `plot()` function at the first position, (223) at the third position plots a histogram using the `hist()` function, and (224) is the fourth plot using `barh()`, giving a horizontal bar graph.

Listing 7-15. subplot1.py

```
 1  import matplotlib.pyplot as pl
 2  import numpy as np
 3
 4  x = np.arange(10)
 5  y1 = np.random.rand(10)
 6  y2 = np.random.rand(10)
 7
 8  fig = pl.figure()
 9
10
11  ax1 = fig.add_subplot(221)
12  ax1.plot(x,y1)
```

```
13
14  ax2 = fig.add_subplot(222)
15  ax2.scatter(x,y2)
16
17  ax3 = fig.add_subplot(223)
18  ax3.hist(y1)
19
20  ax4 = fig.add_subplot(224)
21  ax4.barh(x,y2)
22
23  pl.show()
```

The file produces the graph shown in Figure 7-15.
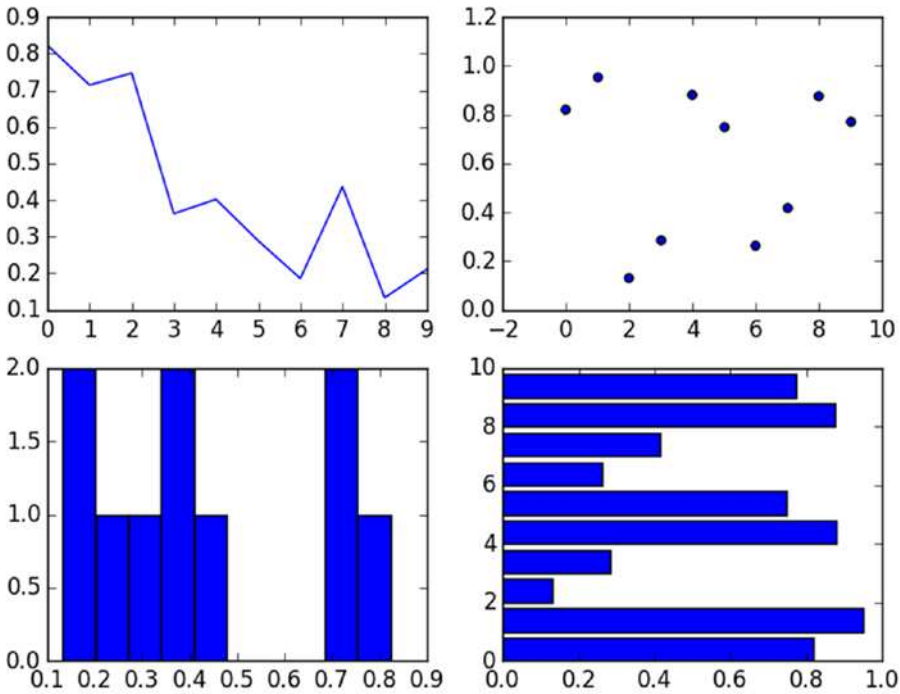
Figure 7-15. *Subplots*

7.13 Saving a Plot to a File

Most of the time, a graph plotted using `matplotlib` needs to be saved for future reference and use. In these cases, the `savefig()` function is used to specify the file name, permission, file type, and so on, for the figure. As an example, Listing 7-16 results a file named `plot1.png` in the working folder. (Those who are working with IPython can find out about the present working directory by typing `pwd`). If you provide a proper path as the string argument of the `savefig()` function, a file is created at that path provided you have proper privileges to create the same.

Listing 7-16. sqPlot4.py

```
1   import matplotlib.pyplot as pt
2   import numpy as np
3   pt.plot (np.arange (10))
4   pt.savefig ('plot1.png')
```

The file produces the graph shown in Figure 7-16.
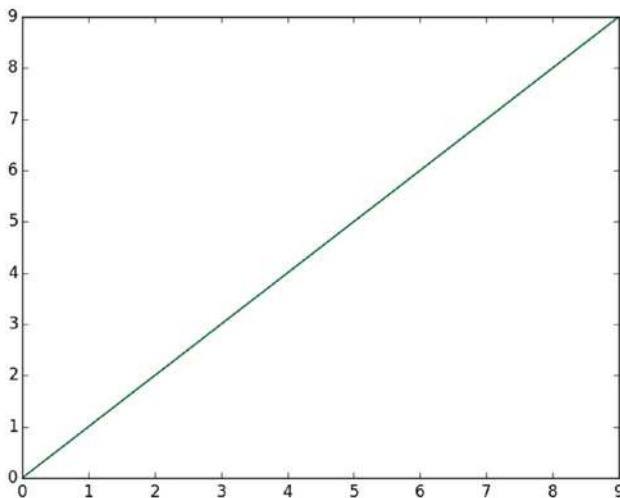
Figure 7-16. *Plotting sin(x) and sin(2x)*

163

It is important to know the size of the file and the resolution of the figure for the purpose of publication at both offline/print and online media. When none of the arguments for setting the resolution of the figure and size of the file are set within the program, default values are used. These default values can be known using the following code:

```
1  >>>import matplotlib as ml
2  >>>ml.rcParams['figure.figsize']
3  [8.0, 6.0]
4  # Default figure size is 8 X 6 inches
5  >>>ml.rcParams['savefig.dpi']
6  100.0
7  # Default figure resolution is 100 dpi
```

Since a 8 × 6 inch figure is created with 100 dpi (dots/pixels per inch), a 800 × 600 pixels image is saved using `savefig()` by default. When this file is directed toward a computer graphic terminal for displaying, length units are ignored and pixels are displayed. If the file is directed toward a printing media like a printer or plotter, length parameters and DPI determine figure size and resolution.

7.14 Displaying Plots on Web Application Servers

In the era of connecting using the Internet, many problems require plotting interactive graphs on web pages. These require different types of plots to be plotted on web pages written in different kinds of languages working under different environments. It is important to note that `matplotlib` requires graphic user interface requiring a X11 connection. Hence, it is important to turn on this faculty on a web application server before updating the plots dynamically, generated by `matplotlib`. There are two aspects to plotting graphs on computers in general. Coding using a set

of commands to make a script is known as a *frontend* task, which requires a *backend* effort. Backend does all the hard work of interacting with graphical capabilities of the system to produce a graph in a desired plot. Plots can be plotted interactively using backends like pygtk, wxpython, tkinter, qt4, and macosx, or they can be plotted noninteractively (permanently saved as files on a computer) using backends like PNG, SVG, PS, and PDF. The latter are also known as *hardcopy* backends.

There are two routes to configure a backend:

1. matplotlibrc file in the installation directory can be edited to set the backend parameter to a particular value such as the following example:

    ```
    1  backend : WXAgg   # use wxpython with antigrain
    2  (agg) rendering
    ```

    matplotlibrc is present at /etc/matplotlibrc on a LINUX system.

2. Use use() to set a particular backend temporarily:

    ```
    1  >>>import matplotlib
    2  >>>matplotlib.use('PDF')   # generate PDF file as
    3  output by default
    ```

Choosing one of the ways to set backend depends on the application of the program. If the program aims to save plots as noninteractive PDF files temporarily, method number 2 can be used, which works until another program sets the backend differently. Setting the backend must be done before import matplotlib.pyplot or import matplotlib.pylab. More information on using various backends for variety of application can be found at http://matplotlib.org/faq/usage_faq.html#what-is-a-backend. For a web application server, setting backend as WXAgg, GTKAgg, QT4Agg, and TkAgg will work.

One way to save transparent figures as opposed to white-colored ones by default is to set `transparent=TRUE`. This is particularly important in the case when figures need to be embedded on a web page with predefined background color/image.

Another way of turning interactive mode to `ON` or `OFF` is by including commands `matplotlib.pyplot.ion()` and `matplotlib.pyplot.ioff()`, respectively.

7.14.1 IPython and Jupyter Notebook

While working with `IPython`, if you wish to work with plots to dynamically change by issuing commands at the `IPython` command prompt, you simply issue a command at the UNIX terminal:

```
1  $ ipython —pylab
```

This enables a special `matplotlib` support mode in `IPython` that looks for a configuration file looking for the backend, activating the proper GUI threading model if required. It also sets the `Matplotlib` interactive mode so that the `show()` commands does not block the interactive `IPython` shell.

When working in the `Jupyter  Notebook` environment, issue the following command:

```
1  $ jupyter notebook ——pylab=inline
```

This command produces the graphs inline, that is, within the body of the code in between the command lines where the plot is called. Otherwise, the plots pop out in a separate window. Inline mode is useful while designing teaching materials. However, before sharing with concerned persons, it should be ensured that encoded backends and dependencies are installed on the user's computer.

Alternatively, you can write `%matplotlib inline` as one of the magic commands in the first cells so that all plots are directed to be shown within the framework of the web browser interface. This is the preferred approach in most cases, especially for teaching Python-based plotting.

7.15 Working with matplotlib in Object Mode

The Pythonic way of using `matplotlib` is to use it in object mode where explicit definition of an object allows ultimate customization. For this purpose, you must define each element of a graph as an object and use the properties to customize the same. The hierarchy of three basic objects used for the purpose is as follows:

1. `FigureCanvas`: Container class for `Figure` instance

2. `Figure`: Container class for axes instance

3. `Axes`: Axes are the rectangular areas to hold various plot features such as lines, ticks, curves and text.

When working with `matplotlib` in object-oriented mode, specify a `FigureCanvas`, which holds `figure(s)` that in turn holds `axes` where a variety of plotting features can be implemented. The whole process allows customization to any extent that can be imagined. Listing 7-17 explains this concept.

Listing 7-17. objPlot.py

```
1  import matplotlib.pyplot as plt
2  fig = plt.figure()
3  # variable fig stores "figure" instance
4  ax1 = fig.add_subplot(221)
5  # variable ax1 stores the subplot of figure at 1st place in
   2 x 1 matrix
```

167

```
 6  ax1.plot([-1, 1, 4], [-2, -3, 4]);
 7  # ax1 iscalled and plot fucntion is given to it.
 8  # plot function carries two lists giving x and y axis
    points for graph
 9  ax2 = fig.add_subplot(222)
10  ax2.plot([1, -2, 2], [0, 0, 2]);
11  # same logic as for ax1
12  ax3 = fig.add_subplot(223)
13  ax3.plot([10, 20, 30], [10, 20, 30]);
14  ax4 = fig.add_subplot(224)
15  ax4.plot([-1, -2, -3], [-10, -20, -30]);
16  plt.show()
17  # show the figure on computer terminal
```

The resulting plot is given by Figure 7-17.
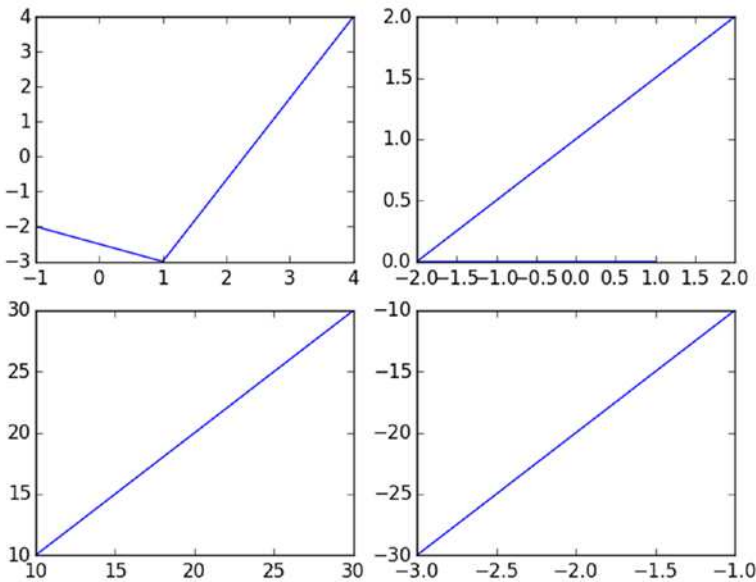
Figure 7-17. *Plotting using object mode capabilities*

7.16 Logarithmic Plots

A variety of engineering data use logarithmic scales, particularly those where changes result in an order of magnitude change in values of observed variable. Python provides the facility to plot logarithmic plots, as shown in Listing 7-18.

Listing 7-18. log.py

```
 1  import matplotlib.pyplot as plt
 2  import numpy as np
 3
 4  x = np.arange(0., 10, 0.01)
 5
 6  fig = plt.figure()
 7
 8  ax1 = fig.add_subplot(221)
 9  y1 = np.log(x)
10  ax1.plot(x, y1);
11  ax1.grid(True)
12  ax1.set_ylabel('$y = log(x)$');
13  ax1.set_title('y-axis in log scale')
14
15  ax2 = fig.add_subplot(222)
16  y2 = np.sin(np.pi*x/2.)
17  ax2.semilogx(x, y2, basex = 3);
18  ax2.grid(True)
19  ax2.set_title('x-axis in log scale')
20
21  ax3 = fig.add_subplot(223)
22  y3 = np.sin(np.pi*x/3.)
23  ax3.loglog(x, y3, basex=2);
24  ax3.grid(True)
```

```
25  ax3.set_ylabel('both axes in log');
26
27  ax4 = fig.add_subplot(224)
28  y4 = np.cos(2*x)
29  ax4.loglog(x, y3, basex=10);
30  ax4.grid(True)
31
32  plt.show()
```

The following axes instances are defined:

1. ax1 uses $y = log(x)$.

2. ax2 uses $y = sin\left(\dfrac{\pi x}{2}\right)$.

3. ax3 uses $y = sin\left(\dfrac{\pi x}{3}\right)$.

4. ax4 uses $y = cos(2x)$.

The resulting graph is given by Figure 7-18.
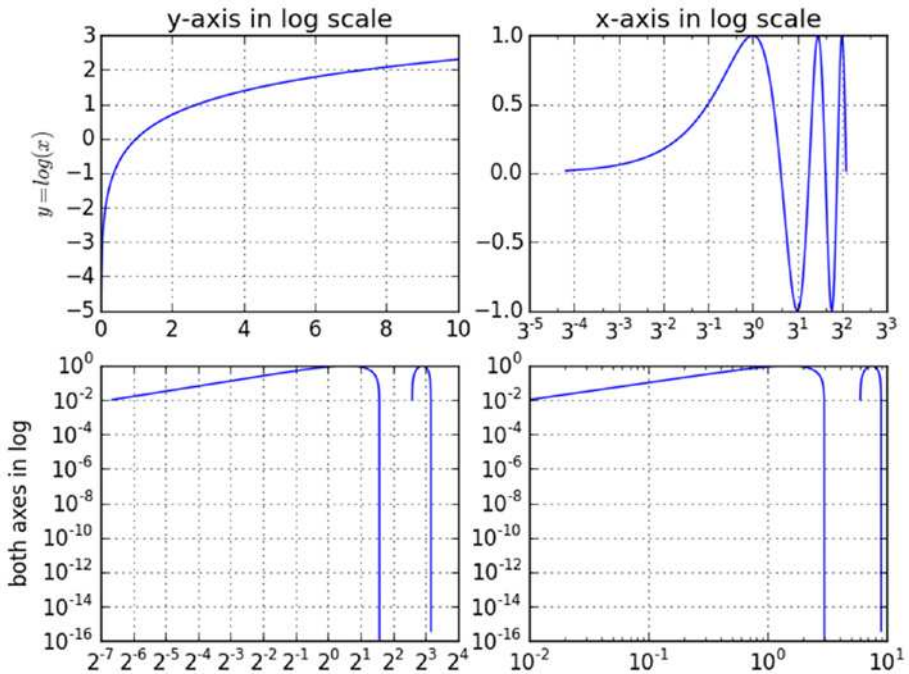
Figure 7-18. *Plotting using object mode capabilities*

As seen in the program log.py, logarithmic values can be directly passed to the plot() function. When a particular axis needs to be plotted in logarithmic values, semilog() and semilogy() can be used where a base index can also be defined. The value of that particular axis is converted into a logarithmic scale and plotted subsequently. When logarithmic values need to be plotted on both axes, the loglog() function needs to be invoked.

Logarithmic plots find their use in a variety of fields like signal processing and thermodynamics. Essentially, whenever data changes by an order of a magnitude, it's easier to observe it using a logarithmic plot. A logarithmic scale is a nonlinear scale. The ability to change the base of a logarithmic function provides a powerful tool at the hands of developers to derive the most meaningful conclusion.

7.17 Two Plots on the Same Figure with at least One Axis Different

Using functions `twinx()` and `twiny()`, you can use two *x*- or *y*-axes on the same figure to plot two sets of data points. An example to use `twinx()` is used in Listing 7-19 where the *x*-axis is *twinned* to produce plots of two data sets sharing the same x-axis data points.

Listing 7-19. twinx.py

```
 1  import matplotlib.pyplot as plt
 2  import numpy as np
 3
 4  x = np.arange(0., 100, 1);
 5  y1 = x**2;
 6  # y1 is defined as square of x values
 7  y2 = np.sqrt(x);
 8  # y2 is defined as square root of x values
 9
10  fig = plt.figure()
11  ax1 = fig.add_subplot(111)
12  ax1.plot(x, y1, 'bo');
13  ax1.set_ylabel('$x^{2}$');
14  ax2 = ax1.twinx() # twinx() function is used to show
    twinned x axes
15  ax2.plot(x, y2, 'k+');
16  ax2.set_ylabel('$\sqrt{x}$');
17  ax2.set_title('Same x axis for both y values');
18  plt.show()
```

It is worth noting that two different axes instances, namely `ax1` and `ax2`, are *superimposed* on each other where data from `y1` being allotted to axes instance `ax1` and data from `y2` being allotted to axes instance `ax2`. This also illustrates the power of defining a plot in object mode. The corresponding plot is given in Figure 7-19.
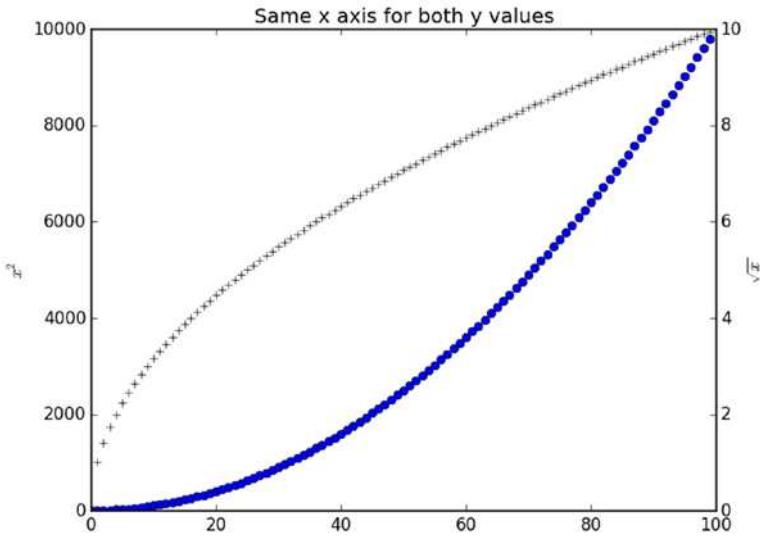
Figure 7-19. *Using twinx() variable*

7.18 Contour Plots

In some engineering applications, contour plots become an essential part of interpretation because they can define segregation of data into regions based on certain similarities. For example, if you imagine a mountain viewed from the top, you can define regions of a similar height being shown with a closed loop. Thus, a mountain will be a series of loops. Similarly, a 2-D map of a nonuniformly heated region can be viewed as contours depicting regions of the same temperature. A region of rainfall can be viewed as a contour showing regions of a dissimilar size of droplets.

Hence, contour lines are also called *level lines* or *isolines*. The term *iso* is attached to data points having constant value and the regions of these data points are separated by contours.

For a contour plot, you need *x-*, *y-* and *z*-axes where the *z*-axis defines the height. The data with the same height is clubbed together within isolines. (See Listing .)

Listing 7-20. contour.py

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # defining data for x, y, z axes
5  x = np.linspace(0,1,100)
6  y = np.linspace(1,2,100)
7  (X,Y) = np.meshgrid(x,y)
8  z = np.sin(X)-np.sin(Y)
9
10 # plotting contour
11 fig = plt.figure()
12
13 ax1 = fig.add_subplot(211)
14 c1 = ax1.contour(x,y,z)
15 l1 = ax1.clabel(c1)
16 lx1 = ax1.set_xlabel("x")
17 ly1 = ax1.set_ylabel("y")
18
19
20 # plotting filled contour
21 ax2 = fig.add_subplot(212)
22 c2 = ax2.contourf(x,y,z)
23 l2 = ax2.clabel(c2)
```

```
24  lx2 = ax2.set_xlabel("x")
25  ly2 = ax2.set_ylabel("y")
26
27  plt.show()
28
29  # plotting filled contour
```

The corresponding plot is given in Figure 7-20.
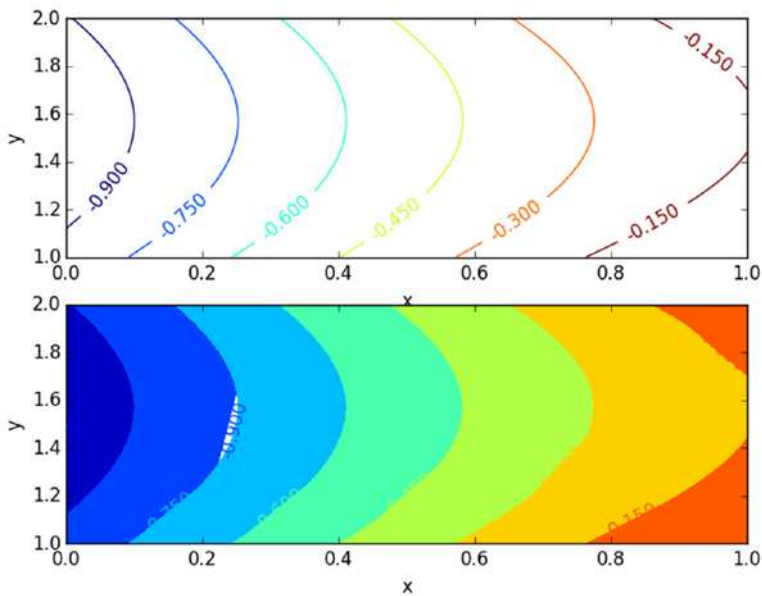
Figure 7-20. *Contour plots*

contour() and contourf() functions can be used to plot unfilled and filled contour. A simple command help(contour) gives extensive information about setting various parameters for a contour plot.

7.19 3D Plotting in matplotlib

With advanced computing technologies at both the software and hardware's end, it has become easier to produce interactive 3-D plots on graphic terminals. `matplotlib` provides a decent number of options in this regard, which are discussed in following subsections.

7.19.1 Line and Scatter Plots

`matplotlib`'s `toolkits` have the class `mplot3D`, which provides an `Axes3D` object. Using the `projection='3D'` keyword, an `Axes3D` object is created, which provides the screen area to show a 3-D plot. `Axes3D` can then be passed on to a `figure` object to show it as a figure. A line plot can be simply created by passing three arguments to the `plot()` function as seen in Listing 7-21.

Listing 7-21. 3Dline.py

```
1   import matplotlib as mpl
2   from mpl_toolkits.mplot3d import Axes3D
3   import numpy as np
4   import matplotlib.pyplot as plt
5
6   fig = plt.figure()
7   ax = fig.gca(projection='3d')
8
9   x = np.linspace(-10*(np.pi),10*(np.pi),10e4)
10  y = np.sin(x)
11  z = np.cos(x)
12
13  ax.plot(x, y, z, label='$y=sin(x)$ and $z = cos(x)$')
14  ax.legend()
15  ax.set_title('3D line curve')
16  ax.set_xlabel('$x$')
```

```
17  ax.set_ylabel('$y = sin(x)$')
18  ax.set_zlabel('$z = cos(x)$')
19  plt.show()
```

The corresponding graph is shown in Figure 7-21.
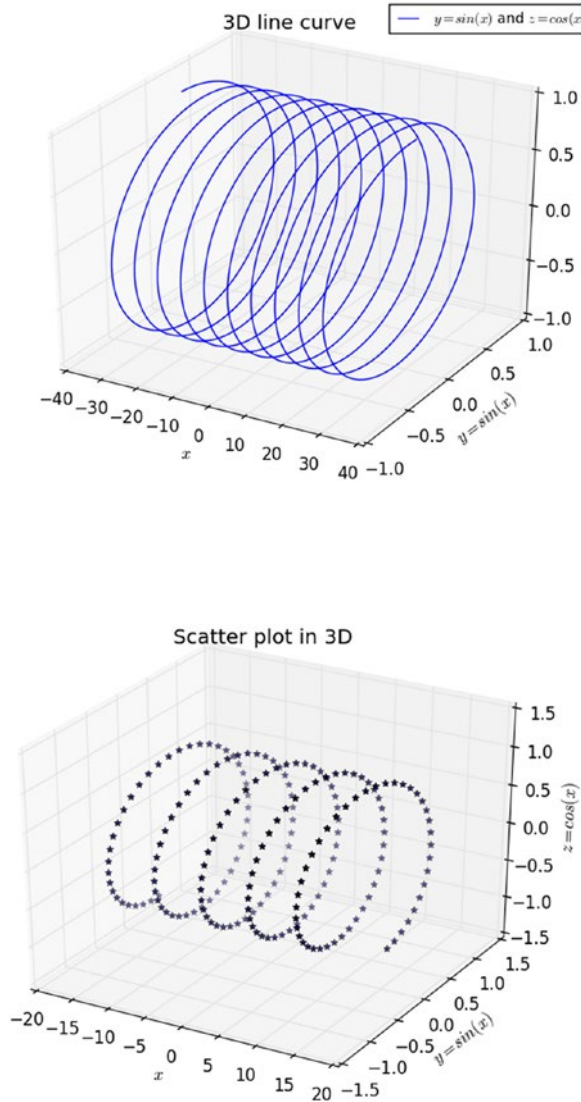
Figure 7-21. *Line and scatter plot in 3D*

Scatter plots are plotted in 3-D in a similar way to line plots. This is illustrated in the Python script Listing 7-22 and the corresponding graph is shown in Figure 7-21.

Listing 7-22. 3Dscatter.py

```
1  import numpy as np
2  from mpl_toolkits.mplot3d import Axes3D
3  import matplotlib.pyplot as plt
4
5  fig = plt.figure()
6  ax = fig.add_subplot(111, projection='3d')
7
8  x = np.linspace(-5*(np.pi), 5*(np.pi),200)
9  y =np.sin(x)
10 z =np.cos(x)
11
12 ax.scatter(x, y, z, marker='*')
13
14 ax.set_xlabel('$x$')
15 ax.set_ylabel('$y = sin(x)$')
16 ax.set_zlabel('$z = cos(x)$')
17 ax.set_title('Scatter plot in 3D')
18
19 plt.show()
```

7.19.2 Wiremesh and Surface Plots

During computation with discrete values, it is sometimes useful to plot a `wiremesh` plot as seen in Figure 7-22. As shown in Listing 7-23, the `meshgrid` function is used to generate a grid of points using x and y values. Function is generated on top of this grid and plotted using the `wiremesh` function. `rstride` and `cstride` define the row and column step size:

$$z = \sqrt{x^2 + y^2}$$

Listing 7-23. 3Dwiremesh.py

```
1   from mpl_toolkits.mplot3d import axes3d
2   import matplotlib.pyplot as plt
3   import numpy as np
4
5   fig = plt.figure()
6   ax = fig.add_subplot(111, projection='3d')
7
8   a = np.arange(-5, 5, 0.25)
9   b = np.arange(-5, 5, 0.25)
10  x, y = np.meshgrid(a, b)
11  z = np.sqrt(x**2 + y**2)
12
13  ax.plot_wireframe(x, y, z, rstride=2, cstride=2)
14
15  ax.set_xlabel('$x$')
16  ax.set_ylabel('$y$')
17  ax.set_zlabel('$z = \sqrt{x^{2}+y^{2}}$')
18  ax.set_title('Wiremesh type of 3D plot')
19
20  plt.show()
```

Surface plots are similar to `wiremesh` plots except for the fact that it's continuously filled up. Hence, instead of the `wiremesh()` function, you use the `surface()` function. (See Listing 7-24.)

Listing 7-24. 3Dsurface.py

```
1   from mpl_toolkits.mplot3d import axes3d
2   import matplotlib.pyplot as plt
3   import numpy as np
4
5   fig = plt.figure()
6   ax = fig.add_subplot(111, projection='3d')
7
8   a = np.arange(-5, 5, 0.25)
9   b = np.arange(-5, 5, 0.25)
10  x, y = np.meshgrid(a, b)
11  z = np.sqrt(x**2 + y**2)
12
13  ax.plot_surface(x, y, z, rstride=2, cstride=2)
14
15  ax.set_xlabel('x')
16  ax.set_ylabel('$y$')
17  ax.set_zlabel('$z = \sqrt{x^{2}+y^{2}}$')
18  ax.set_title('Surface type of 3D plot')
19
20  plt.show()
```

The corresponding graph is shown in Figure 7-22.
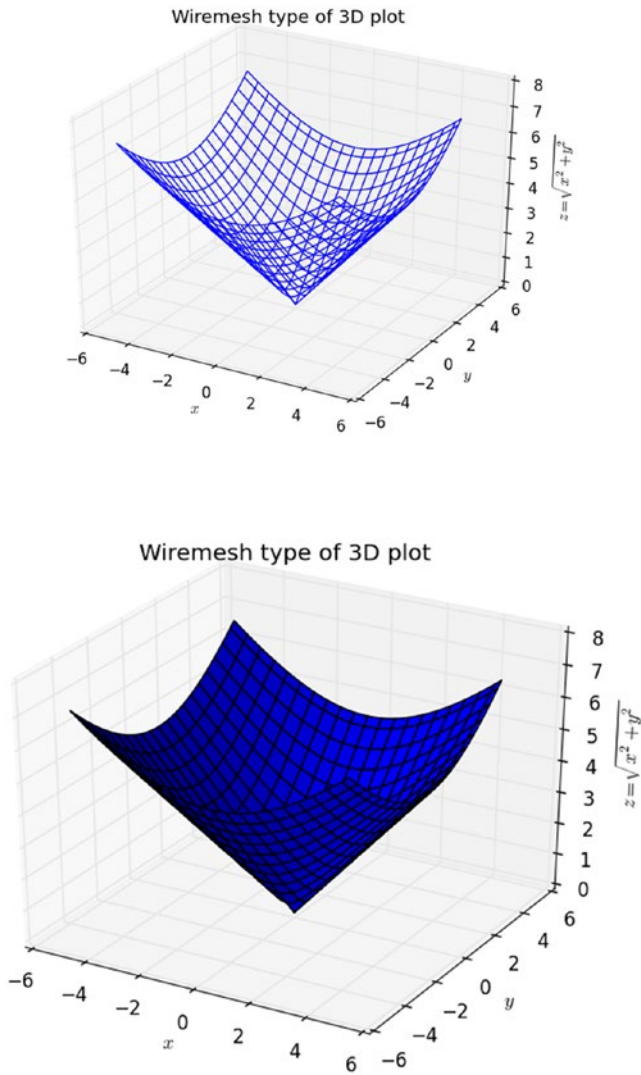
Figure 7-22. *Wiremesh and surface plot*

7.19.3 Contour plots in 3D

Just as in 2-D contours, 3-D contour plots employ *isosurfaces* (surfaces having equal height). Using contour() and contourf() functions, you can plot unfilled and filled contour plots, as shown in Listing 7-25.

Listing 7-25. 3Dcontour.py

```
1   from mpl_toolkits.mplot3d import axes3d
2   import matplotlib.pyplot as plt
3   from matplotlib import cm
4   import numpy as np
5
6   fig = plt.figure()
7   ax1 = fig.add_subplot(121, projection='3d')
8   x = np.linspace(2*np.pi,-2*(np.pi),1000)
9   y = np.linspace(2*np.pi,-2*(np.pi),1000)
10  X,Y = np.meshgrid(x,y)
11  Z = np.sin(X) + np.sin(Y)
12
13  cont = ax1.contour(X, Y, Z)
14  ax1.clabel(cont, fontsize=9, inline=1)
15  ax1.set_xlabel('$x$')
16  ax1.set_ylabel('$y$')
17  ax1.set_title('Contour for $z=sin(x)+sin(y)$')
18
19  ax2 = fig.add_subplot(122, projection='3d')
20  Z = np.sin(X) + np.sin(Y)
21  cont = ax2.contourf(X, Y, Z)
22  ax2.clabel(cont, fontsize=9, inline=1)
23  ax2.set_xlabel('$x$')
24  ax2.set_ylabel('$y$')
25  ax2.set_title('Filled Contour for $z=sin(x)+sin(y)$')
```

```
26
27
28
29  plt.show()
```

The corresponding graph is shown in Figure 7-23.
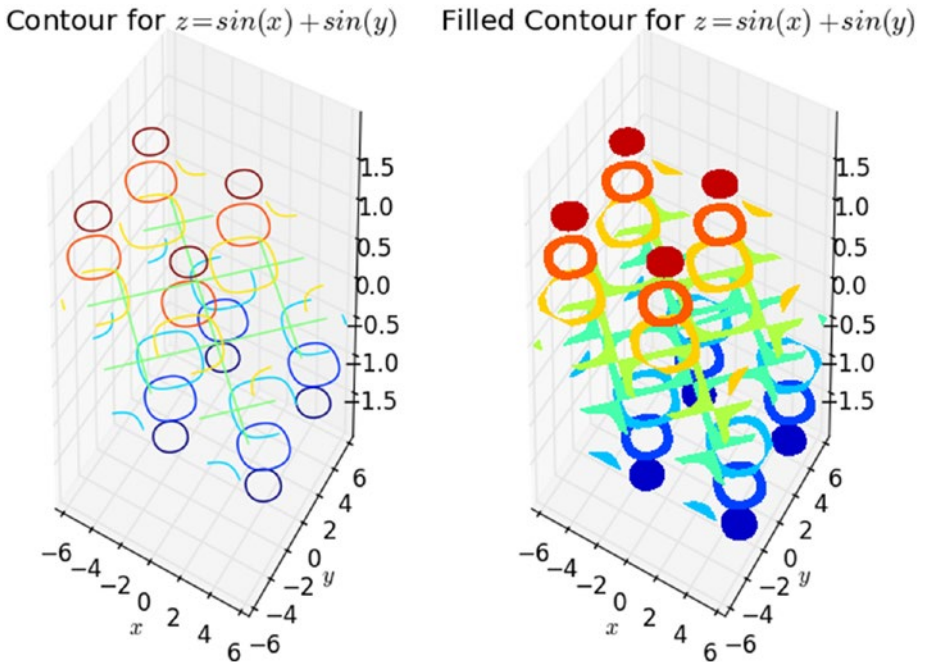
Figure 7-23. *Contour plots using contour and contour functions plot*

7.19.4 Quiver Plots

Listing 7-26. 3Dquiver.py

```
1  from mpl_toolkits.mplot3d import axes3d
2  import matplotlib.pyplot as plt
3  import numpy as np
```

```
 4
 5  x = np.linspace(np.pi,-(np.pi),10)
 6  y = np.linspace(np.pi,-(np.pi),10)
 7  (X,Y) = np.meshgrid(x,y)
 8  u = -15*X
 9  v = 5*Y
10  q = plt.quiver(X,Y,u,v,angles='xy',scale=1000,color='b')
11  #p = plt.quiverkey(q,1,16.5,50,"50 m/s",coordinates='data',
    color='r')
12  xl = plt.xlabel("x (km)")
13  yl = plt.ylabel("y (km)")
14  plt.show()
```

The corresponding graph is shown in Figure 7-24.
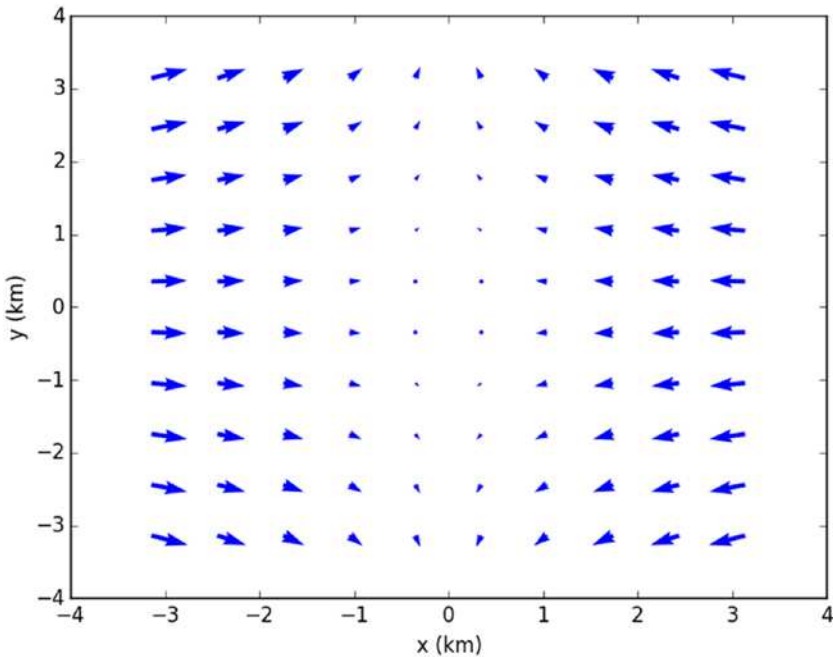
Figure 7-24. *Quiver plots*

7.20 Other Libraries for Plotting Data

matplotlib has been in use to such an extent that new developers do not realize other options to plot data. There are a variety of other ways to plot data in other modules that might have more powerful plotting capabilities depending on context. Among them are plotly, mavavi, and gnuplot (to name a few). A brief discussing follows about using plotly, which interestingly plots data on the Web. This is particularly interesting for those engineering applications where the sensor data needs to be plotted on the Web in real time.

7.20.1 plotly

plotly is an online analytics and data-visualization tool. In addition to Python, plotly can plot data used in Perl, Julia, Arduino, R, and MATLAB. Weblink [5] gives a pretty good sense of the capabilities of plotly. Essentially, plotly allows plotting and publishing graphs online to encourage collaboration. Hence, you must be connected to the Internet before working with the plotly library.

First, you need to make a user account at the plotly web site [5]. The account will provide a username and API key, which needs to be used in the program. Then, you can write a code either as one command at a time on a Python terminal or as a Python script. (See Listing 7-27.)

Listing 7-27. plotly.py

```
1  import numpy as np
2  import plotly.plotly as py
3  from plotly.graph_objs import *
4  py.sign_in('username','APIkey')
5  data = Data([Scatter(x=np.arange(100),y=np.random.
   randn(100), name='trace 0')])
6  fig = Figure(data=data)
7  plot_url = py.plot(fig)
```

You need to provide the *username* and *API key* in the script in Listing 7-27. The script results in generating a scatter plot online at a workspace in the user account. These graphs can be plotted quite interactively using the functions provided by `plotly`. When data are steamed from a device connected to a live sensor, live data is plotted in real time. This can further be embedded on a web site.

7.21 Summary

In this chapter, we have discussed various plotting options available while working with Python. The ease of plotting data is one of the most attractive features of Python. Just a few lines of code provide visualization of the data in a variety of ways. Visualization is the backbone of data presentation and analysis since it makes understanding of the data clearer. In addition to simple visualization, `matplotlib` provides rich features to decorate the graph with useful information in a desired manner.

7.22 Bibliography

[1]   J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[2]   http://matplotlib.org/.

[3]   https://matplotlib.org/users/installing.html.

[4]   http://matplotlib.org/api/lines_api. html#matplotlib.lines.Line2D.

[5]   https://plot.ly/.

Functions and Loops

8.1 Introduction

A function is a part of a computer program where a number of programming statements is clubbed as a block. It can be called when desired. This enables a modular approach to programming tasks and has become very popular among programmers nowadays since modules can be edited with ease. Functions receive *input parameters* and return *output parameters*. When a function is being used, the *function name* is called along with values for input parameters. After execution, a set of output parameters is returned. Python functions can be defined at any place in the program, regardless of the place from where they are being called. They can even be defined as a separate file individually or in a combined manner. Also, they can be called any number of times or they may not be called at all, as per the user's requirements.

8.2 Defining Functions

Just like any other language, Python has its own way of defining functions. Following is the structure of a Python function:

```
1  def function_name(parameter_1, parameter_2, ...):
2      """Descriptive string"""
3      # Comment about statements below
4      statements
5      return return_parameters
```

As you can see, a Python function consists of three parts:

1. **Header:** Begins with a keyword (`def`) and ends with a colon (`:`)

2. **Descriptive String**: A string that describes the purpose of a character and can be accessed using the `help()` function

3. **Body:** An indented (four whitespaces in general) set of Python statements below the header

8.2.1 Function Name

Function names follow the same rules as variable names in Python. It is a good idea to give a function a name that is relevant to its description and to keep it short.

8.2.2 Descriptive String

An essential part of a defining a function is to define its inner working details as a string. When `help()` is used, this descriptive string is displayed to the user so the user can understand the function and its usage. The usage of a descriptive string is not a compulsory feature, but it is recommended as good programming practice. Since the description should be as detailed as possible, it constitutes a multiline string. Multiline strings can be written under triple quotes. Even if the description is only one line long, it might need to print single or double quotes for emphasizing a word or phrase. As a result, the usage of triple quotes has been mandated in the definition of a descriptive string.

8.2.3 Indented Block of Statements

To define the block of statements that is part of a function, indentation is used as a marker. Statements that are indented after the first statement to define a function are part of the body of the statement that the function will execute. When a statement is written without indentation, the function is exited.

The rules of defining a function name are the same as those for defining variable names. It is important to name them with functionally relevant names so that they are easy to recall when needed.

For example, fn-hello.py, as shown in Listing 8-1, does not take any input parameters and simply executes the statements of printing the string Hello world.

Listing 8-1. fn–hello.py

```
1  def greet():
2      '''
3      greet() just greets with the string
4      "Hello World!" each time it is called
5      '''
6      print("Hello world!")
7
8  greet()
```

Running the file fn-hello.py produces the following output:

```
1  >>>python fn—hello.py
2  >>>Hello World!
```

8.2.4 Return Statement

When a function returns a parameters, it doesn't always need to print the same. Returning can be many other kinds of actions apart from simply printing on a screen. Returning can include passing the variables or their values to another function and/or variables, generating a file of code/data, and generating graphs and/or storing the function as a file in a graphic format. The Python file fn-sq.py prints the square of the first 10 integers, as shown in Listing 8-2.

Listing 8-2. fn–sq.py

```
1  def square(x):
2      '''
3      square() squares the input value
4      One must only used numeric data types
5      to avoid getting error
6      '''
7      return x*x
8
9  for i in range(10):
10     squared_i= square(i)
11     print(i, squared_i)
```

Following is the result of running the program:

```
1  >>>0 0
2  1 1
3  2 4
4  3 9
5  4 16
6  5 25
```

```
 7   6 36
 8   7 49
 9   8 64
10   9 81
```

Here the function square is called inside a "for loop," which increments the value of the variable i from 1 to 10 (generated by the built-in function range()). The for loop structure will be discussed in Section 8.6.

8.3 Multi-input and Multi-output Functions

A function can input and return any number of parameters, as shown in Listing 8-3.

Listing 8-3. def–multi.py

```
1  def sum(a,b):
2      '''
3      sum() takes two inputs and produces
4      a tuple of inputs and thier sum
5      '''
6      c = a+b
7      return a,b,c
8
9   results = sum(100,102)
10  print(results)
```

The execution results in:

```
1  >>>(100, 102, 202)
```

8.4 Namespaces

When we call a function within a program, Python creates a namespace to work within this function. The namespaces are containers that store information about mapping names to objects. Since multiple functions can be defined within a main program, multiple namespaces can exit at the same time independently. They can contain the same variable names. Hence, a hierarchy level must be defined for conflict resolution. This is defined by scope rules.

8.4.1 Scope Rules

Scope determines the range of workability for an object—in other words, where the object can and cannot be accessed for computations. As it is clear by now, the variables defined within a function are defined to be local variables, whereas variables defined in the main program are global variables that can be accessed anywhere. The Python code `namespace.py` gives a better view of this concept, as shown in Listing 8-4.

Listing 8-4. namespace.py

```
1   # Python code to demonstrate the concept
2   # of namespace and scope
3   i = 1
4
5   def my_function1():
6       i = 2
7       print("value of i in my_function()1 scope is", i )
8
9   def my_function2():
10      i = 3
11      print("value of i in my_function()2 scope is", i )
12
```

```
13  print('value of i in global scope is', i )
14  my_function1()
15  print('value of i in global scope is', i )
16  my_function2()
17  print('value of i in global scope is', i )
```

The result is as follows:

```
1  value of i in global scope is', 1
2  value of i in my function()1 scope is', 2
3  value of i in global scope is', 1
4  value of i in my function()2 scope is', 3
5  value of i in global scope is', 1
6
```

In the code namespace.py, the variable named i assigned three values:

1. i=1 within the main program

   • **scope** of this variable is **global**. (It remains the same all through the program even if other functions are called.)

2. i=2 within the function named my_function1

   • **scope** of this variable is local, that is, within the function my_function1. Outside the function, it loses its definition.

3. i=3 within the function named my_function2

   • **scope** of this variable is local, that is, within the function my_function2. Outside the function, it loses its definition.

The hierarchical structure of **scope** follows LEGB rules. *LEGB* stands for *Local→Enclosed→Global→Built-in*.

This essentially dictates in which order the Python interpreter searches for different levels of namespaces before it finds the name-to-object mapping. Following are the four levels of namesspaces:

- **Local**: This is within a user-defined function or class method.

- **Enclosed**: This is within an enclosing function, that is, in a case when a function is wrapped inside another function.

- **Global**: This signifies the uppermost level, that is, the main Python program that is being executed by the interpreter.

- **Built-in**: These are special names that the Python interpreter reserves for itself.

If a name cannot found in any of the namespaces, a `NameError` will be raised.

8.5 Concept of Loops

The main advantage of using computers for calculations when performing repetitive tasks is that they can compute faster than humans. The term *loop* is associated with repetitive calculations because a user-defined variable names the values and the computer repetitively shuffles the variable values in a specified sequence generated by a condition. For example, a user might like to find the square root of the first 10 integers. To perform this calculation, the user needs to run the function `sqrt()` on a list of the first 10 integers (which can be generated by `range()`) function.) The list of integers can be stored in an integer, and this variable can be put into a loop to perform the operation of finding the square root of each member of the list one-by-one.

8.6 for Loop

When the same operation has to be carried out on a given set of data, for loop is a good choice. For example, suppose we simply want to print the individual members of the list. The Python code in Listing 8-5 can be employed.

Listing 8-5. ListMembers.py

```
1   #Python program to illustrate usage
2   #of for loop
3   a = ['a',1,3.14,'name']
4
5   for item in a:
6       print("The current item is: ",item)
```

The output is printed on the terminal as the following:

```
1   The current item is: a
2   The current item is: 1
3   The current item is: 3.14
4   The current item is: name
```

It is worth noting that most of the programming languages employ a logical statement defining the initialization, condition, and increment for running the code. Python programs employ a different strategy where an array is employing a condition and the loop simply iterates on each member of the array. This is important for Python programs since Python is an interpretive language and is inherently slower in operation. Spending time checking a condition each time the loop wishes to take a step is a computationally costly affair. Hence, Python devises the computation in such a way that once a list/array is formed as per the defined condition, the loop can then run a list/array members. In this way, the overall computation time can be reduced.

As an example to understand the usage of for loop for numerical computation, let's suppose we want to find the even Pythagorean numbers such that

$$a^2 + b^2 = c^2$$

This can be accomplished using for loop as given in the Python code pytha.py shown in Listing 8-6. In this Python code, the user inputs a number denoting the maximum number for which this calculation will be performed. This ensures that the calculation has a proper *end* condition (without explicit definition).

Listing 8-6. pytha.py

```
1   # Program to generate even Pythagorean numbers
2   # Pythagorean numbers are those numbers for which
    pythagorus equation stands true
3
4   from numpy import sqrt
5   n = input("Please input a maximum number:") # Asks user to
    input a number
6   n = int(n)+1 # converts the values stored in variable n to
       integer data type and adds 1 so that computation can be done
       if user feeds 0
7
8   # Two loops to define arrays for a and b for which c shall
    be computed
9   for a in range(1,n):
10      for b in range(a,n):
11          c_square = a**2 + b**2
12          c = int(sqrt(c_square)) # c is converted to an integer
```

```
13        if ((c_square - c**2) == 0): # if square of a and
          square of b is equal to square of c then the result
          will be zero
14            if (c%2 ==0): # checking if c is an even number
15                print(a, b, c)
```

```
 1 Please input a maximum number:200
 2 6 8 10
 3 10 24 26
 4 12 16 20
 5 14 48 50
 6 16 30 34
 7 18 24 30
 8 18 80 82
 9 20 48 52
10 22 120 122
11 24 32 40
12 24 70 74
13 26 168 170
14 28 96 100
15 30 40 50
16 30 72 78
17 32 60 68
18 32 126 130
19 36 48 60
20 36 160 164
21 40 42 58
22 40 96 104
23 40 198 202
24 42 56 70
25 42 144 150
26 48 64 80
27 48 90 102
```

28   48 140 148
29   50 120 130
30   54 72 90
31   56 90 106
32   56 192 200
33   60 80 100
34   60 144 156
35   64 120 136
36   66 88 110
37   66 112 130
38   70 168 182
39   72 96 120
40   72 154 170
41   78 104 130
42   78 160 178
43   80 84 116
44   80 150 170
45   80 192 208
46   84 112 140
47   90 120 150
48   96 110 146
49   96 128 160
50   96 180 204
51   102 136 170
52   108 144 180
53   112 180 212
54   114 152 190
55   120 126 174
56   120 160 200
57   120 182 218
58   126 168 210
59   130 144 194

```
60  132 176 220
61  138 184 230
62  144 192 240
63  150 200 250
64  160 168 232
```

As an exercise, you can write a few more lines of code to check if there are any prime number triplets as Pythagorean numbers.

8.7 if-else Loop

In the example Python code `pytha.py`, an `if` statement has already been used. It simply checks a condition, runs the loop if the condition results `True` boolean data, and exits the loop if the condition results `False` boolean data. When multiple conditions need to be checked in a sequence, `if-else` loops are employed where the `if` condition results `True` boolean data and the statement is executed. Otherwise, the next condition is checked and a similar operation is performed recursively. This action is performed unless all conditions result in returning the `False` boolean data. (See Listing 8-7.)

Listing 8-7. ifelif.py

```
1  # Program to calculate shipping cost based on money spent
   and location
2
3  total = int(input('What is the total amount for your online
   shopping?\n'))
4  area = input('''Type "I" if you are shopping within
   India...
5  and "O" if you are shopping outside India\n''')
6
```

```
 7  if area == "I":
 8      if total <= 500:
 9          print("Shipping Costs INR 20.00")
10      elif total <= 1000:
11              print("Shipping Costs INR 100.00")
12      elif total <= 1500:
13              print("Shipping Costs INR 250.00")
14      else:
15          print("FREE")
16
17  if area == "O":
18      if total <= 500:
19          print("Shipping Costs INR 75.00")
20      elif total <= 1000:
21          print("Shipping Costs INR 200.00")
22      elif total <= 1500:
23          print("Shipping Costs INR 500.00")
24      else:
25          print("FREE")
```

```
 1  >>>What is the total amount for your online shopping?
 2  2001
 3
 4  Type "I" if you are shopping within India...
 5  and "O" if you are shopping outside India
 6  I
 7  FREE
 8
 9  >>>What is the total amount for your online shopping?
10  300
11
12  Type "I" if you are shopping within India...
```

```
13  and "O" if you are shopping outside India
14  I
15  Shipping Costs INR 20.00
```

8.8 while Loop

A while loop has the following syntax:

```
1  while expression:
2  statement(s)
```

Note that the statement(s) are indented for grouping. The statement(s) can be single or multiple actions. The condition is a logical expression. The loop iterated until the value of the logical expression is True. As soon as it becomes False, the program control is passed to the next line. while loop plays an important role in cases where looping must be skipped if the condition is not satisfied since none of the statement is executed if the logical expression has False value.

The program while.py gives an example code demonstrating the working of while loop. (See Listing 8-8.) Here another module, namely time, is used to time taken by two lines of codes for their execution. Writing help(time) gives important documentation regarding its usage. The function time.clock() returns a floating point number that represents CPU time since the start of the process or the time when this function is called first. By subtracting the two, you get a number depicting the number of seconds taken to execute statements.

Listing 8-8. while.py

```
1  # Program demonstrating usage of while loop
2
3  # Program to count number of steps and time taken for thier
     execution
4
```

```
5  import time #This module is used for timing lines of codes
6  import numpy as np
7
8  i = 0 # initializing the counter
9  while(i<10): # counter condition
10       start = time.clock() # defining the variable which stores
         time.clock(value)
11       print("Square root of %d  = %3.2f:"%(i,np.sqrt(i)))
         # printing the number and its squareroot
12       i=i+1 # incrementing the counter
13       timing = time.clock() - start # prints time taken to
         execute two lines of code above
14       print("Time taken for execution = %e seconds \n" %
         timing)
15
16  print("The end") # signifies exiting the loop after
    condition is satisfied
```

The result is shown as follows:

```
1  >>>Square root of 0  = 0.00:
2  Time taken for execution = 0.000158652234404 seconds
3
4  Square root of 1  = 1.00:
5  Time taken for execution = 3.97699673158e-05 seconds
6
7  Square root of 2  = 1.41:
8  Time taken for execution = 4.19081375185e-05 seconds
9
10  Square root of 3  = 1.73:
11  Time taken for execution = 2.77962135442e-05 seconds
12
13  Square root of 4  = 2.00:
```

```
14  Time taken for execution = 2.22369708354e-05 seconds
15
16  Square root of 5  = 2.24:
17  Time taken for execution = 2.18093368858e-05 seconds
18
19  Square root of 6  = 2.45:
20  Time taken for execution = 2.13817029362e-05 seconds
21
22  Square root of 7  = 2.65:
23  Time taken for execution = 2.1809336431e-05 seconds
24
25  Square root of 8  = 2.83:
26  Time taken for execution = 2.22369708354e-05 seconds
27
28  Square root of 9  = 3.00:
29  Time taken for execution = 2.43751414928e-05 seconds
30
31  The end
```

Note that the output time might be different for each execution even on same the computer since time taken to process a line of code is functional of the state of CPU at that particular moment of time.

8.9 Infinite Loops

If the logical expression always outputs the boolean value True, the program never stops. These loops are termed as *infinite* **loops** since they will take an infinite amount of time for execution. One of the simplest examples is given in the Python code infinite.py shown in Listing 8-9.

Listing 8-9. infinite–loop.py

```
1  # Program to define an infinite loop
2  # Press Ctrl+C to interrupt execution of python REPL
3  i=1
4  while i==1:
5      print(i)
6  print("Good bye")
```

Since the condition always remains true, the program will never quit displaying the value of i, which is 1. It will never print the last line of the code. On a Linux machine, you need to press CTRL+C to interrupt the executing and come back to the command line.

8.10 while-else

Within a while loop, the statements are executed if the condition produces a boolean value True. Using an else statement within this structure allows the user to route the flow of the program if the condition returns the boolean value False, as shown in Listing 8-10.

Listing 8-10. while–else.py

```
1  # Python program to explain usage of while-else loop
2  i=0
3
4  while i<=5:
5      print(i)
6      i=i+1
7  else:
8      print("the value now execeeds 5")
```

Following is the result:

```
1  0
2  1
3  2
4  3
5  4
6  the value execeeds 5
```

As soon as the incremented value becomes 5, the flow is handled by statements under the `else` condition.

8.11 Summary

Functions enable the modular structure of programs. Additionally, controlling the flow of information as well as iterations has become the very basis of computational work in most applications. These two actions are performed by loops. Together, they make Python a powerful tool for various applications. Having a modular structure makes it easier to test and debug. Mastering the skill of writing functions and the skill of choosing the proper loop structure has become key indicators for ranking a programmer's performance in solving problems using Python codes. Hence, this chapter is very important for programmers.

CHAPTER 9

Object-Oriented Programming

9.1 Introduction

The idea of defining objects is quite natural to humans since we deal with objects in our day-to-day life. We understand an object as something that has

- a set of attributes and

- a related set of behaviors.

Early on, babies learn about objects in their environment. For example, as babies, we learn that a ball (*object*) has a color, shape, and size (*attributes*), and it rolls, skids, and bounces (*behavior*). In the early 1970s, Alan Kay at Xerox PARC (Palo Alto Research Center) worked on the concept of object-oriented programming (OOP) [1]. While working on a programming language called *Smalltalk* [2], he employed the ideas of OOP. He based his key ideas on applying computer programming to physical simulation. Most people understand the real world as various objects (having attributes and behaviors) interacting with each other. Thus, it is natural to adopt the same ideas while constructing a simulated world. All GUI-based systems inherit their philosophy from Kay's efforts toward these ideas and now all major programming languages follow them religiously.

9.2 Procedural Programming vs. OOP

While defining a computational task, you can define a set of procedures to solve a problem using *blocks* of data and connecting them as dictated by procedures. The paradigm that emphasizes setting procedures regardless of the type of data and its different usage patterns is called *procedural programming*. On the other hand, object-oriented programming places emphasis on objects and their relationships with one another as defined using operators (acting on objects, they change their values and other attributes) to solve a computational task. Python is an OOP language.

9.3 Objects

In the Python world, everything is an object. But what is an object? An object is an abstract concept to signify an entity on which computation can be performed. Just like a physical object, a computer's object has a *set of attributes* and a *related set of behaviors*. A number, string, pictures, videos, and files can be visualized as objects. Within numbers, you can subcategorize objects into other objects such as integers, floating point numbers, and boolean numbers, or their collection in an ordered or unordered fashion. Within strings, you can have characters, words, sentences, and so on. Within files, you can have text files, media files, data files, script files, and so forth. Let's explore some common Python objects:

```
1  >>>a=10.0
2  >>>help(a)
3  Help on float object :
4
5  class float(object)
6  |   float(x) -> floating point number
7  |
```

```
 8  |   Convert a string or number to a floating point number,
        if possible.
 9  .
10  .
11  .
12  >>>b=1+2j
13  >>>help(b)
14  Help on complex object:
15
16  class complex (object)
17  |  complex(real[, imag]) -> complex number
18  |
19  |  Create a complex number from a real part and an optional
        imaginary part.
20  |  This is equivalent to (real + imag*1j)   where imag
        defaults to 0.
21  .
22  .
23  .
24  >>>c=10
25  >>>help(c)
26  Help on int object:
27
28  class int(object)
29  |  int(x=0) -> int or long
30  |  int(x, base=10) -> int or long
31  |
32  |  Convert a number or string to an integer, or return 0 if
         no arguments
33  |  are given.   If x is floating point, the conversion
         truncates towards zero.
```

```
34 |  If x is outside the integer range, the function returns a
        long instead.
35 |
36 |  If x is not a number or if base is given, then x must
        be a string or
37 |  Unicode object representing an integer literal in the given
        base. The
38 |  literal can be preceded by '+' or '−' and be surrounded
        by whitespace.
39 |  The base defaults to 10.  Valid bases are 0 and 2−36.
        Base 0 means t o
40 |  interpret the base from the string as an integer literal.
41 |  >>> int('0b100', base=0)
42 |  4
43 .
44 .
45 .
```

It is interesting to note that Python classifies the three numbers stored in variables a, b, and c in distinct classes as objects. The *help* string details their usage. Anything and everything that needs to be computed can be imagined as an entity called an *object* with a *set of attributes* and *related set of behaviors.* The set of attributes is stored in instance variables and their behavior is probed by set of functions called *methods.*

The command dir() lists all the methods and attributes associated with the object:

```
1  >>> a=1
2  >>> dir(a)
3  ['__abs__', '__add__', '__and__', '__bool__', '__ceil__',
    '__class__', '__delattr__', '__dir__', '__divmod__',
    '__doc__', '__eq__', '__float__', '__floor__',
    '__floordiv__', '__format__', '__ge__',
```

```
          '__getattribute__', '__getnewargs__', '__gt__',
          '__hash__', '__index__', '__init__',
          '__init_subclass__', '__int__', '__invert__', '__le__',
          '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__',
          '__neg__', '__new__', '__or__', '__pos__', '__pow__',
          '__radd__', '__rand__', '__rdivmod__', '__reduce__',
          '__reduce_ex__', '__repr__', '__rfloordiv__',
          '__rlshift__', '__rmod__', '__rmul__', '__ror__',
          '__round__', '__rpow__', '__rrshift__', '__rshift__',
          '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
          '__sizeof__', '__str__', '__sub__', '__subclasshook__',
          '__truediv__', '__trunc__', '__xor__', 'bit_length',
          'conjugate', 'denominator', 'from_bytes', 'imag',
          'numerator', 'real', 'to_bytes']
 4
 5  >>> a.__abs__() # absolute value
 6  1
 7  >>> a.__neg__() #negative value
 8  -1
 9  >>> a.__float__() #floating point value
10  1.0
11  >>> a.bit_length() #Number of bits necessary to represent
        self in binary.
12  1
```

Objects can be operated by an attribute using for operator. Let's use the previous case as an example:

- a.__abs__() defines the **absolute** value of *integer* object (in other words, value 1).

- a.__neg__() defines the **negative** value of *integer* object (that is, value −1).

- `a.__float__()` defines the **floating point** value of *integer* object (value 1.0).

- `a.__hex__()` defines the **hexadecimal** value of *integer* object (value $0x1$).

- `a.__oct__()` defines the **octal** value of *integer* object (value 01).

- `a.bit_length` returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

The following sections will explain the ways to define objects and their usage. First, let's understand that different *types* of data must be treated differently in a computer for their mathematical nature and for defining computational resource requirements intelligently.

9.4 Types

An object has an associated *type*. Type dictates the memory storage requirements and what can be done computationally with an object. For example, `int` and `float` are distinct types of an object in the sense that `int` stores integers and `float` stores floating point numbers. While `float` needs to store information about how many digits precede and succeed the decimal point, `int` objects do not need to worry about the same. Similarly, a complex number is stored in another *type* of object (aptly named `complex`) since it needs to store two aspects of a complex numbers: their real and imaginary parts. In this way, they must be stored quite differently in computer memory and then used quite differently in subsequent computations.

The type of an object can be obtained by using the built-in function called type(), which takes the object whose type needs to be scanned. For example:

```
1   >>> a =1
2   >>> type(a)
3   <class 'int'>
4   >>> b=1.0
5   >>> type(b)
6   <class 'float'>
7   >>> c =2+3j
8   >>> type(c)
9   <class 'complex'>
10  >>> d = 'd'
11  >>> type(d)
12  <class 'str'>
13  >>> e = 'addition'
14  >>> type(e)
15  <class 'str'>
16  >>> f = [1,2,3]
17  >>> type(f)
18  <class 'list'>
19  >>> g = 1,2,3
20  >>> type(g)
21  <class 'tuple'>
22  >>> import numpy as np
23  >>> a=np.array([1,2,3])
24  >>> type(a)
25  <class 'numpy.ndarray'>
26  type(np.sin)
27  <class 'numpy.ufunc'>
```

From this practice code, it is easy to understand that line 1 is found to make an object of the type integer (`int`), line 4 is found to make an object of the type floating point number (`float`), line 7 is found to make an object of the type complex number (`complex`), lines 10 and 13 are found to make an object of the type string (`str`), line 16 is found to make an object of the type list (`list`), line 19 is found to make an object of the type tuple (`tuple`), line 23 is found to make an object of the type numpy array (`numpy.ndarray`), and line 1 is found to make an object of the type a numpy function called `numpy.ufunc`.

9.5 Object Reference

Pythonic objects are represented as a *reference* to an object in memory. A reference is a value that references (points to) a memory location. This can be understood by the following example, which uses the `id()` function. The `id()` function returns a unique number representing the memory address of storage of input variable:

```
 1  >>> a1 = 10
 2  >>> id(a1)
 3  4469844832
 4  >>> a2=a1
 5  >>> id(a2)
 6  4469844832
 7  >>> a3=20
 8  >>> id(a3)
 9  4469845152
10  >>> a3=a2
11  >>> id(a2)
12  4469844832
13  >>> id(a3)
14  4469844832
```

```
15  >>> a1
16  10
17  >>> a2
18  10
19  >>> a3
20  10
```

As seen in the Python code, a1 is assigned a value of 10. In Python's language, an integer object is created (at a memory location whose id is given by 4469844832). The built-in function id() can be used to probe the memory location's id.

When a2=a1 statement is issued, we find that same id is assigned to both reference values a1 and a2. This is checked next. Now another integer object with value 20 is created and referenced with value a3. Its id is probed and found to be different from a1 and a2. Hence, each time an object is created, it is stored in different memory location. In other words, it is given a different id.

What happens when an object is referenced by two reference values. When we issue the command a3 = a2, a3 and a2 reference to the same object as stored in a2. But a2 shares the reference with a1, which stores an integer object valued 10. Hence, now all three reference values a1, a3, and a3 reference to same integer object having value 10.

9.5.1 Garbage Collection

What happened with the integer object 20, which was created at line 7? This object was *dereferenced*. Consequently, this memory location must be *deallocated*; its status should change from *currently-in-use* to *available-for-reuse*. This process is called *garbage collection* and is very important for efficient memory management. Python performs this automatically.

One of the most primitive methods is to maintain a reference count. For every object, a count of the total number of references to that object is maintained. If that count ever falls to 0, it is immediately deallocated. But more advanced algorithms [3, 4] are now used for the purpose of ruling out cases where this primitive idea may pose a problem. Unfortunately, discussing the ideas about Python's way of garbage collection is beyond the scope of this book. If you are interested in this topic, the following references will help in this regard [5, 6, 7].

9.5.2 Copy and Deepcopy

Referencing a single object with two reference values seems like copying. It has been observed in previous discussions that when users make changes to the original object, both copies are affected. While working with list objects, this can be an issue:

```
1  >>> list1 = [1,2,3]
2  >>> list2 = list1
3  >>> list1
4  [1, 2, 3]
5  >>> list2
6  [1, 2, 3]
7  >>> list1[1]=-1
8  >>> list1
9  [1, -1, 3]
10 >>> list2
11 [1, -1, 3]
```

First, we create a list object with value [1,2,3] whose reference value is list1. Next, we create another reference value, list2, in addition to the original reference value list1 referring to the original list object ([1,2,3]). When the second item on the list is altered by line number 7, it is reflected by both reference values since both of these point to the same object.

Hence, `list2` seems a copy of `list1`, which follows its original master diligently. Let's investigate if this pattern holds when its members of lists are lists themselves:

```
 1  >>> list1 = [1,2,3]
 2  >>> list2 = [list1,list1]
 3  >>> list1
 4  [1, 2, 3]
 5  >>> list2
 6  [[1, 2, 3], [1, 2, 3]]
 7  >>> list3=list2
 8  >>> list3
 9  [[1, 2, 3], [1, 2, 3]]
10  >>> list2[1]=[-1,-2,-3]
11  >>> list1
12  [1, 2, 3]
13  >>> list2
14  [[1, 2, 3], [-1, -2, -3]]
15  >>> list3
16  [[1, 2, 3], [-1, -2, -3]]
```

So, it seems that even if members of list objects are list objects themselves, the copying activity works the same way when members were other objects.

The following code shows what happens if we use `list` constructor:

```
 1  >>> list1 = list([1,2,3])
 2  >>> list2 = list([list1,list1,list1])
 3  >>> list2
 4  [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
 5  >>> list3 = list(list2)
 6  >>> list3
 7  [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

```
 8  >>> list2[1]=list([-1,0,1])
 9  >>> list2
10  [[1, 2, 3], [-1, 0, 1], [1, 2, 3]]
11  >>> list3
12  [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
13  >>> id(list2)
14  4416356864
15  >>> id(list3)
16  4416356936
```

Note that the reference value `list3` has different data because they now point to two different objects (as seen by their memory id values). Hence, a copy of the original list did not follow the master diligently under `list` constructor. Thus, `list3` is said to be a *shallow* copy of `list2`.

Sometimes, we may like to make a copy and keep it unchanged under various transformations even if `list` constructors are used. For this purpose, the module named `copy` provides two methods named `copy.copy()` and `copy.deepcopy()`. As per Python documentation [8]:

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original. This is performed using the `copy.copy()` method of `copy` module.

- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original. This is performed using the `copy.deepcopy()` method of `copy` module.

```
1  >>>import copy
2  >>> a = [[1, 2, 3], [4, 5, 6]]
3  >>> b = list(a)
4  >>> a
5  [[1, 2, 3], [4, 5, 6]]
```

```
 6  >>> b
 7  [[1, 2, 3], [4, 5, 6]]
 8  >>> a[0][1] = 10
 9  >>> a
10  [[1, 10, 3], [4, 5, 6]]
11  >>> b   # list b changes too => Not a deepcopy.
12  [[1, 10, 3], [4, 5, 6]]
```

Now, let's see how deepcopy works:

```
 1  >>> b = copy.deepcopy(a)
 2  >>> a
 3  [[1, 10, 3], [4, 5, 6]]
 4  >>> b
 5  [[1, 10, 3], [4, 5, 6]]
 6  >>> a[0][1] = 9
 7  >>> a
 8  [[1, 9, 3], [4, 5, 6]]
 9  >>> b    # list b doesn't change => Deep Copy
10  [[1, 10, 3], [4, 5, 6]]
```

As you can see, it enables b to retain its value even when a is changed.

9.6 Class

Until now, we have just discussed predefined objects and scanning their type and their memory id. Using special types of objects called *class*, we can create other objects.

The concept of class is also derived from the natural world. Objects in the real world are *classified* according to their attributes and behavior. An object is merely an instance of its class. For example, men and women are merely instances of the class *homo sapiens*. In a similar fashion, the keyword class defines a class of objects for which various instances can

be defined. For example, a class for numbers can have instances like integers, floating point numbers, complex numbers, rational numbers, and irrational numbers. Once defined, all members of this class will share certain attributes and behaviors common to numerals, but they can be further subdivided based on their differences. All members of this class will be quite different from a class of language having characters, words, sentences, and paragraphs. Now an instance of number class as an integer and an instance of language class as a character can be defined for computing. These instances of their respective classes will have their own attributes and methods (some of them shared with their class and some of their own). This object orientation helps us break down a complex system into small fundamental units for easier study.

Thus, a class defines the behaviors of a new kind of abstract entity, whereas an object is just a particular instance of that object. Classes and objects help to define actions of functions on data in a manageable way that can be managed in a similar fashion irrespective of the nature of data and functions. An interface describes the functionalities of an object, whereas its actual implementation defines how these functionalities can be performed. Classes have *constructors* (we have already discussed the constructor `list()` in Section 9.5.2) that dictate the creation of an object of a particular kind. Encapsulation and inheritance describe the hierarchal structure of classes with their super-classes. These concepts will be further elaborated in subsequent sections.

9.6.1 Creating a Class

The keyword `class` is used to create a class. For example, a built-in class named `int` defines the behavior of the object used to store integer numbers. When a variable is associated with an object of this class, it is called an instance of the class. For example, the statement `a=1` creates an object named `a`, which is an instance of `int` class. On the other hand, the statement `b=1.0` creates an object named as `b`, which is an instance

of float class. Depending on the type of objects stored in a and b, their behavior will considerably differ upon usage.

The keyword class is used to create an object. Its syntax is defined with rules as follows:

```
1  class Name_of_class:
2     '''Optional class documentation string'''
3     statements
```

It is customary to name classes in meaningful ways for ease of their usage. Similarly, it customary to put their details of usage in a documentation string. This string can be accessed via the following command:

Name_of_class. __doc__

Variables that belong to a particular object (defined by a class) are called *fields*. These fields can be probed by a function defined within the class for this purpose. Such functions are called *methods*.

9.6.2 Class Variables and Class Methods

Class variables indicate the data that universally applies to all the objects defined within a class. They are defined within class definitions and are valid under the *scope of class*. This signifies the fact that each and every object defined within that class has access to them. They can be accessed without giving reference to a class.

On the other hand, an instance variable is valid only within an instance of the class object. Hence, it is defined within the definition of the instance of the class (within the def black defining behavior of instance object).

Class methods are different from ordinary functions in just one way. They are required to have an extra first name. This first name needs to be added to the beginning of the parameter list. Users do not give a value for this parameter when they call the method. It is provided by Python's main program, which is run by the Python interpreter by default. This particular variable refers to the object itself. Thus, it is given the name self. Those methods that do not take any arguments have at least the argument as self. Listing 9-1 will make this concept clearer.

Listing 9-1. Ex–class.py

```
1   #Python code to explain defining
2   #and using class
3
4   #A class named python_program is defined
5   class first_program:
6       '''
7       This program has two methods: hi and hi_again
8       '''
9       greeting = 'Hello Everybody\n' #Class variable definition
10
11      #Now we define two instances of this class as "hi" and
        "hi_again"
12      def hi(self):
13          greet = 'Hello World!\n'
14          print(greet)
15      def hi_again(self):
16          greet_again = 'Hello again!\n'
17          print(greet_again)
18
19  #Procedures to call the class
20
21  x = first_program() # Calling the class
```

```
22  print('The type of object storing class calling is:', type(x))
23  x.hi() #Calling an instance of "x" class
24  x.hi_again() # Calling another instance of "x" class
25  # Results of both instances depend on thier definitions
26
27  #Acessing class variable
28  #Class variable can be accessed from anywhere
29  print(first_program.greeting)
30
31  #Printing the documentation string
32  print(first_program.__doc__)
```

The output can be seen as follows:

```
 1  $python Ex-class.py
 2  The type of object storing class calling is:
    <class'__main__.first_program'>
 3  Hello World!
 4
 5  Hello again!
 6
 7  Hello Everybody
 8
 9
10  This program has two methods: hi and hi_again
```

9.6.3 Constructor

All objects within a class are merely instances of the class. When this instance is initiated, a special kind of functions is called. A *constructor* is a function that is executed when an object is instantiated. Python always names a constructor as __init()__ since it sets the *initial* state of the object. It defines the initial state and the basic nature of this instance. It is

defined within class definition, but a user can omit this definition because every class automatically creates a default constructor. It is generally used to define the *initial values* of the instance variables, as shown in Listing 9-2.

Listing 9-2. Ex–class1.py

```
1  #Python code to show usage of __init__
2
3  # Defining a class "library"
4  # which stores information about book's name and price
5
6  #Defining class
7  class library:
8      'The base class for all books'
9     bookCount = 0 #Class variable setting count of books to zero
10
11     def __init__(self, name, price):# Initializing the
        instance of class with name, price and updated book
        count
12         self.name = name
13         self.price = price
14         library.bookCount += 1
15
16     def displayCount(self):
17       print("Total Employee %d",library.bookCount)
18
19     def displayBook(self):
20         print("Name : ", self.name,  ", Price: ", self.price)
21
22  #Creating objects
23
24  book1 = library("Introduction to python",100) # First
    instance of object "library"
```

```
25  print(type(book1))
26
27  book2 = library("Basic Python",150) # Second instance
    of object "library"
28  print(type(book2))
29
30  book3 = library("Intermediate Python",200) # Third instanc
    of object "library"
31  print(type(book3))
32
33  book4 = library("Advanced Python",300) # Four instance
    of object "library"
34  print(type(book2))
35
36  # Accessing attributes
37
38  print("Details of first book:")
39  book1.displayBook()
40
41  print("Details of second book:")
42  book2.displayBook()
43
44  print("Details of third book:")
45  book3.displayBook()
46
47  print("Details of fourth book:")
48  book4.displayBook()
49
50  print("Total Number of books= %d" % library.bookCount)
51
52  # Probing the bult-in attributes
53
```

```
54  print("library.__doc__:", library.__doc__)
55  print("library.__name__:", library.__name__)
56  print("library.__module__:", library.__module__)
57  print("library.__bases__:", library.__bases__)
58  print("library.__dict__:", library.__dict__)
```

The code Ex-class1.py can be explained as follows:

- Line 7 defines a class named library. This class intends to keep an account of the name of a book and its price.

- Line 9 defines a class variable named bookCount and sets it to zero value.

  – Since it is a class variable, it is shared by all instances.

  – It can be accessed by library.bookCount from inside or outside the class.

- This class has a constructor defined from line 11 to line 14 and two methods named displayCount and displayBook.

- Lines 11–14 define the constructor used to initialize all the instances:

  – It takes three arguments: self, name, and price.

  – self refers to the object itself.

  – name and price are initialized at lines 12 and 13, respectively.

  – These are required at the time of calling the class, that is, library(name,price).

  – This initialization information is utilized by all instances of the object library.

- Lines 16–17 define a method called `displayCount` and output value referred by the `bookCount` variable.

- Lines 19–20 define a method called `displayBook`, which gives the information about the name and price of the book.

- Lines 21–33 create a separate instance of object `library`.

  – As defined by the constructor, the two input arguments while calling the class must be the values stored in variables `name` and `price`.

- Lines 39, 44, 48, and 52 call the object(s) one-by-one, probing their attribute defined by the method `displayBook()`.

- Line 50 displays the total number of books using the method `bookCount()`.

Built-in Class Attributes

All Python classes have certain built-in attributes that can be accessed by using the dot operator. This is done from line 54 to line 58.

- `__doc__`

  – Gives the class documentation string

  – Gives `None` if not defined

- `__dict__`

  – Gives the dictionary containing the class's namespace

- – This dictionary contains the information about the object `library`, namely:

  - `__module__`
  - Defined methods (`displayCount()` and `displayBook()` in the present case)
  - Defined class variable (`bookCount` in the present case)
  - `__doc__`
  - Constructor `__init__`

- `__name__`

  - – Gives the name of class

- `__module__`

  - – Gives the name of the module within which the class is defined
  - – In the interactive mode, this is `__main__`

- `__bases__`

  - – A possibly empty tuple containing the base classes in the order of their occurrence in the base class list

The output of Python code Ex-`class1.py` is given as follows:

```
1  <class '__main__.library'>
2  <class '__main__.library'>
3  <class '__main__.library'>
4  <class '__main__.library'>
5  Details of first book:
6  Name:  Introduction to python , Price:   100
7  Details of second book:
8  Name:  Basic Python , Price:   150
```

```
 9  Details of third book:
10  Name :  Intermediate Python , Price:  200
11  Details of fourth book:
12  Name :  Advanced Python , Price:  300
13  Total Number of books= 4
14  library.__doc__: The base class for all books
15  library.__name__: library
16  library.__module__: __main__
17  library.__bases__: (<class 'object'>,)
18  library.__dict__: {'__module__': '__main__', '__doc__':
    'The base class for all books', 'bookCount': 4, '__init__':
    <function library.__init__ at 0x117f85ae8>, 'displayCount':
    <function library.displayCount at 0x117f85b70>, 'displayBook':
    <function library.displayBook at 0x117f9a488>, '__dict__':
    <attribute '__dict__' of 'library' objects>, '__weakref__':
    <attribute '__weakref__' of 'library' objects>}
```

9.7 Summary

In this chapter, we have illustrated how to define a new kind of object using the `class` construct. Since Python considers everything as an object, it is one of the favorite choices for scientific computation and simulation. Because it is easy to learn, users can start serious scientific computing very quickly. This fact makes it very popular among educators, students, and researchers. The ability to make classes and create your own modules enables developers to enrich the Python ecosystem on a daily basis. Hence, it isn't an overstatement to say that the object-oriented nature of Python and its open source ecosystem have made it one of the most popular programming languages.

9.8 Bibliography

[1]   http://propella.sakura.ne.jp/earlyHistoryST/
      EarlyHistoryST.html.

[2]   http://web.cecs.pdx.edu/~harry/musings/
      SmalltalkOverview.html.

[3]   R. Jones, A. Hosking, and E. Moss, *The garbage
      collection handbook: The art of automatic memory
      management*. Chapman & Hall/CRC, 2011.

[4]   B. Zorn, *Comparative performance evaluation of
      garbage collection algorithms*. PhD thesis, University
      of California, Berkeley, 1989.

[5]   https://docs.python.org/3/library/gc.html.

[6]   http://www.digi.com/wiki/developer/index.
      php/Python_Garbage_Collection.

[7]   http://arctrix.com/nas/python/gc/.

[8]   https://docs.python.org/2/library/copy.html.

Numerical Computing Formalism

10.1 Introduction

Numerical computation enables us to compute solutions for numerical problems, provided we can frame them into a proper format. This requires certain considerations. For example, if we digitize continuous functions, we are going to introduce certain errors due to the sampling at a finite frequency. Hence, a very accurate result would require a very fast sampling rate. In cases when a large dataset needs to be computed, it becomes a computationally intensive and time-consuming task. Also, users must understand that the numerical solutions are an approximation, at best, when compared to analytical solutions. The onus of finding their physical meaning and significance lies on us. The art of discarding solutions that do not have a meaning for a real-world scenario is something that a scientist/engineer develops over the years. Furthermore, a computational device is just as intelligent as its operator. The law of GIGO (garbage in, garbage out) is followed very strictly in this domain.

In this chapter, we will consider some of the important steps in solving a physical problem using numerical computation. Defining a problem in proper terms is just the first step. Making the right model and then using the right method to solve (solver) the problem distinguishes an experienced scientist/engineer from a novice.

10.2 Physical Problems

Everything in our physical world is governed by physical laws. Because of the men and women of science who toiled under difficult circumstances to come up with fine solutions for the natural events happening around us, we obtained mathematical theories for physical laws. To test these mathematical formalisms of physical laws, we use numerical computation. If it yields the same results as that of a real experiment, they validate each other. Numerical simulations can remove the need of doing an experiment altogether provided we have a well-tested mathematical formalism. For example, nuclear powers of our times don't have to test real nuclear bombs anymore. The data about nuclear explosions, which were obtained during real nuclear explosions, have enabled scientists to model these physical systems quite accurately, thus eliminating the need for a real test.

In addition to applications such as simulating a real experiment, modeling physical problems is a good educational exercise. While modeling, hands-on exercises enable students to explore the subject in depth and give a proper meaning of the topic under study. Solving numerical problems and visualizing the results make the learning permanent and also ignite the research related to flaws in mathematical theory, ultimately leading to new discoveries.

10.3 Defining a Model

Modeling is defined as writing equations for a physical system. As its name suggests, an equation pertains to equating two sides. An equation is written using an = sign where terms on the left-hand side are equal to terms on the right-hand side. The terms on either sides of equations can be numbers or expressions. For example:

$$3x + 4y + 9z = 10$$

This is an equation having a expression, 3x + 4y + 9z, on the left-hand side (LHS) and a term, 10, on the right-hand side (RHS). Please note that while LHS is an algebraic term, RHS is a number.

Expressions are written using functions that show a relation between two domains. For example, $f(x) = y$ illustrates a relationship of y to x using rules of algebra. Mathematics has a rich library of functions that can be used to make expressions. Choosing the proper function depends on problem. Some functions describe some situations better than others. For example, oscillatory behavior can be described in a reasonable manner using trigonometric functions such as $sin(x)$ and $cos(x)$. Objects moving in straight lines can be described using linear equations such as $y = mx + c$ where x is their present position, m is the constant rate of change of x, w.r.t y and c is the offset position. Objects moving in a curved fashion can be described by various nonlinear functions where the power of a dependent variable, like x in the previous sentence, is not 1.

In real life, we can have situations that can be a mixture of these scenarios. For example, an object can oscillate and move in a curved fashion at the same time. In such cases, we write an expression using a mixture of functions or find new functions that could explain the behavior of the object. Verifying the functions is done by finding solutions to equations describing the behavior and matching them with observations made about an object. If they match perfectly, we obtain perfect solutions. In most cases, an exact solution might be difficult to obtain. In these cases, we get an "approximate" solution. If the errors involved while obtaining an approximate solution are within toleration limits, the models can be acceptable.

As previously discussed, physical situations can be analytically solved by writing mathematical expressions in terms of functions involving dependent variables. The simplest problems have simple functions between dependent variables with a single equation. There can be situations where multiple equations are needed to explain a physical behavior. In the case of multiple equations being solved, the theory of matrices comes in handy.

Suppose the following equations define the physical behavior of a system:

$$-x+3y=4 \tag{10.1}$$

$$2x-4y=-3 \tag{10.2}$$

This system of two equations can be represented by a matrix equation, as follows:

$$\begin{bmatrix} -1 & 3 \\ 2 & -4 \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

Now using matrix algebra, the values of variables x and y can be found such that they satisfy the equations. These values are called the *roots* of these equations. These roots are the point in 2-D space (because we had two dependent variables) where the system will find stability for that physical problem. In this way, we can predict the behavior of the system without actually doing an experiment.

The mathematical concept of differentiation and integration becomes very important when we work with a dynamic system. When the system is constantly changing the values of dependent variables to produce a scenario, it is important to know the rate of change of these variables. When these variables are independent of each other, we use simple derivatives to define their rate of change. When they are not independent of each other, we use partial derivatives.

For example, Newton's second law of motion says that rate of change of velocity of an object is directly proportional to the force applied on it. We can show this concept mathematically:

$$F\alpha\frac{dy}{dx} \tag{10.3}$$

The proportionality is turned into equality by substituting for a constant of multiplication m such that the following is true:

$$F = m \times \frac{dy}{dx} \qquad (10.4)$$

If we know the values or expressions for F, this equation can be solved analytically and solutions can be found for this equation. However, in some cases, the analytical solution may be too difficult to obtain. In such cases, we digitize the system and find a numerical solution.

There are many methods to digitize and numerically solve a given function. Programs to implement a particular method to solve a function numerically are called a solver. A lot of solvers exist to solve a function. The choice of solver is critical to successfully obtain a solution. For example, Equation 10.4 is a differential equation. It is a first-order ordinary differential equation. A number of solvers exists to solve it including Euler and Runge-Kutta. The choice of a particular solver depends on the accuracy of its solution, the time taken for obtaining a solution, and the amount of memory used during the process. The latter is important where memory is not a freely expendable commodity as when using microcomputers with limited memory storage.

The advantage of using Python to perform a numerical computation lies in the fact that it has a very rich library of modules to perform various tasks required. The predefined functions have been optimized for speed and accuracy (in some cases, accuracy can be predefined). This enables the user to rapidly prototype the problem instead of concentrating on writing functions to do basic tasks and optimizing them for speed, accuracy, and memory usage.

10.4 Python Packages

A number of packages exists to perform numerical computation in a particular scientific domain. The web site [1] gives a list of packages. Installing packages can be attained by writing the command

```
>> pip install <package-name>
```

on the LINUX command line. Users are encouraged to check out the following packages for scientific computation:

- `numpy`: For numerical computation

- `scipy`: Superset of `numpy` that encompasses specific functions for physics apart from general mathematics in `numpy`

- `scikit-learn`: Machine learning

- `tensorflow`: Machine learning

- `Pandas`: Statistical data analytics

- `scikit-image`: Image processing

- `bokeh`: Interactive plotting

10.5 Python for Science and Engineering

Computers are used in both theoretical as well as experimental studies in science and engineering. In theoretical studies, computers are mainly used for solving problems where actions are iteratively performed for a large set of similar or different data points within a model of a real-world problem. Experimental investigations utilize computers for instrumentation and control. Hence, an ideal programming language for scientific investigation must perform these tasks in an efficient manner. Efficiency here encompasses the following:

236

- Ability to write the problem in simple, intuitive, and minimalistic syntax

- Minimum time of execution

- Ability to store and retrieve large amounts of data in an errorfree manner

- Ability to process data using parallel processing paradigm

- Ability to handle vast types of data sets

- Object-oriented programming

- Wide graphics capability

- Architecture independence

- Instrumentation and control system for a variety of platforms

- Networking

- Security

- Less time for prototyping a problem

Let's evaluate Python on these parameters to judge its usage for various scientific tasks. This exercise will give important clues to users before using Python for a scientific problem.

10.6 Prototyping a Problem

Python is an interpretive language. The process of interpretation removes the act of compiling the code prior to producing machine code. Python reads the code line-by-line and outputs machine code as soon as one line is interpreted correctly. Interpretive languages have great advantages in the act of prototyping a problem, but are notoriously slow in execution.

10.6.1 What Is Prototyping?

Prototyping a problem involves formulating a mathematical model for a real-life problem and then coding the mathematical model using the syntax of a particular programming language. Mathematical modeling involves the following:

- Formulating variables

- Marking them dependent and independent in nature

- Devising functional relationships between them by assigning known mathematical functions as assumptions

Once a mathematical model has been devised, it needs to be tested using a computer program to produce output, which is then analyzed in terms of meaningful predictions and/or conclusions for real-life problems under study. Prototyping involves repeating the process of making models and testing them several times to perform feasibility analysis of a particular model before deciding to choose the same and making a complete model. At this stage of prototyping, user may choose to ignore other efficiency-related parameters like faster execution.

The act of converting a mathematical model to a computer program involves expressing mathematical architecture in terms of syntax of the programming language. This can be done part-wise for a model or as a whole. Part-wise modeling would require compatibility of various parts. Python provides a series of advantages in this regard.

10.6.2 Python for Fast Prototyping

The advantage of interpretive architecture is *lessening the time to debug*. As a code is interpreted line-by-line, the codes runs fine until it encounters a problem. This helps in identifying and isolating the problematic part of the code, thus enabling faster debugging. The overall result is a dramatic

reduction of time for prototyping a solution as most of the time for devising a solution is spent in debugging. In addition to this fact, Python is devised for *simple, intuitive,* and *minimalistic* syntax, which further accelerates the prototyping process, saving time for the people involved in solving the scientific problem. The ability to *easily visualize* problems with powerful graphic libraries such as `matplotlib` and `mayavi` adds value to the quality of the coding process because mistakes can be found more easily and they can be presented in a better manner.

Being open source and modular in structure, Python provides the ability to part-wise model develop faster by using existing code instead of reinventing the wheel by writing it again. Within the same version of Python (Python 2 or Python 3), the modules are compatible. Python also allows codes of some languages to run natively within a Python code. Using the Cython package, you can embed C code in a Python program. Similarly, using the Jython package, you can embed java code in a Python program. This allows programmers to choose various programming languages as per their abilities and still develop their model in Python, taking advantage of what their programming languages do not offer. This also enables programmers to use legacy code instead of writing it again in Python.

10.7 Large Dataset Handling

The `h5py` package enables developers to handle HDF5 binary data format, which is mostly used to store large amounts of data efficiently. Numerical computation using Python packages like `numpy` and `scipy` can then be operated on these data points in a vectorized manner. An array-based computing paradigm used for `numpy` and `scipy` becomes advantageous here since the large dataset interfaced using `h5py` can be operated as if it's an array. Thousands of datasets can be categorized, tagged, and then saved in a single file.

In the age of the Internet, users might like to fetch data from database servers, perform computation on a smaller chunk of data at a time, and send back results to application servers for further processing and report generation. For this purpose, Python provides a variety of packages. Python has a standard mechanism for accessing databases called the Database API (DB-API). The Python DB-API specifies a way to connect to databases and issue commands to them. Python DB-API fits nicely into existing Python code and allows Python programmers to easily store and retrieve data from databases [2].

DB-API includes the following:

- Connections that encompass the guidelines about connecting to databases

- Executing statements and stored procedures to query, update, insert, and delete data with cursors

  - A cursor is a Python objects that points to a particular location in the database.

  - Once a cursor is obtained, various methods like inserting, updating, and deleting data as well as querying data can be performed.

- Transactions with support for committing or rolling back a transaction

  - A transaction is a sequence of operations performed as a single logical unit of work having four distinct properties (ACID: Atomicity, Consistency, Isolation, and Durability).

  - The possibility to roll back a transaction is crucial for securing very important data.

- Examining metadata on the database module as well as on a database and table structure

  – Metadata associated with a database describes the features of that database.

  – The ability to access metadata allows developers to use the database judiciously.

- Defining the types of errors and providing exceptions using Python

A large list of databases has been included in the DB-API list, which enables developers to interact with multiple databases within a single code.

Specialized packages like `pandas` are designed to perform vectorized operations on large datasets in an efficient manner. These are used extensively nowadays in the field of big data. Most often, statistical analysis is needed for a dataset. The `pandas` package provides most of the statistical functions required for basic as well as advanced statistical analysis. Coupled with simple plotting libraries like `matplotlib`, quick verification of analysis and the ability to produce publication-quality graphs have extended a golden helping hand to developers.

10.8 Instrumentation and Control

Python cannot connect to a hardware directly unless the hardware maintains an operating system (OS) where Python is installed. There are microcomputers like Raspberry Pi (RPi) that do this. Hence, Python codes directly interact with connectors where sensors and actuators are connected. Since Python is an open source language, running on an open source hardware gives a lot of advantages to developers since ready-made solutions can be available in most cases. This reduces the total time of development drastically. RPi running Linux OS supports Python using the library `rpi.gpio` to perform, read, and write operations at GPIO (general-purpose input/

output) pins. Thus, using Python code, developers can read electrical signals from sensors connected to RPi. Developers can also design complicated electrical systems with actuators like motors connected to GPIO pins by programming in Python to drive them. In most cases, scientific inputs work in a feedback loop configuration where sensors read some physical parameters, and these values are used to drive actuators to perform tasks. Python can perform this task with ease. With this whole package being defined under open source license, developers are free to reconfigure it in any way desired. This enables scientists to develop customized equipment as needed for their experiments.

In the case of hardware not running an OS, Python cannot directly access the underlying hardware. It cannot interface directly with the software library modules provided by most hardware vendors either. In these cases, Python codes can be written to tap communication at a serial port or a USB device that utilizes what is referred to as a virtual serial port. In these cases, there are two options for developers: writing a C extension in the form of DLL (dynamic-Link library) or using a `ctypes` library, which provides methods to directly access function in external DLL. If DLL is already available via a vendor, `ctypes` can access its functionalities within a Python code.

Another great module is `PyVisa`[3]. `PyVisa` is a Python package that enables the developers to control all kinds of measurement devices independently of the interface (for example, GPIB, RS232, USB, and Ethernet). This is a great relief for complicated machines since a number of different protocols are used to send data over many different interfaces and bus systems (for instance, GPIB, RS232, USB, and Ethernet). Sometimes the programming language that a developer wishes to use might not have libraries that support both the device and its bus system. As a result, Virtual Instrument Software Architecture (VISA) was devised. VISA is a standard for configuring, programming, and troubleshooting instrumentation systems comprising GPIB, VXI, PXI, Serial, Ethernet, and/or USB interfaces.

In any case, the responsibility of understanding the signal output from electronic devices as well as managing data flow lies

with developers. Such developers must have sound knowledge of electronics, data communication, and Python programming as well as C/C++ programming. Developers must have a clear understanding of transmission media and its limits in terms of data transfer rates, signal configuration, and data blocks. They must also understand the type of connection, whether it be serial or parallel in nature.

10.9 Parallel Processing

As opposed to a single task being performed in a serial processing paradigm, parallel processing devises ways to perform two or more processes at the same time. A process is the smallest unit of computation done at a processor. Single-core processors can only perform serial processing in the traditional sense. Multicore processors are affordable and can be found in most hardwares nowadays where CPS has more than one core. Python provides a number of modules that enable multicore processing.

Two important concepts make up most of the parallel programming framework: *threads* and *processes*. A process, in the simplest terms, is an executing program. One or more threads run within a process. A thread is the basic unit to which the operating system allocates processor time. Now two approaches can be employed in parallel programming:

- Running code via threads

- Running code via multiple processes

A number of "jobs" can be submitted to different threads. Jobs can be considered as "subtasks" of a single process. Threads usually have access to the same memory areas (in other words, shared memory). This approach requires proper synchronization to avoid conflict. For example, if two processes wish to write to the same memory location at the same time, conflicts will result in errors. Hence, a safer approach is to submit multiple processes to completely separate memory locations (that is,

distributed memory). In this scenario, each process run completely independent from the other. The `multiprocessing` module (`https://docs.python.org/2/library/multiprocessing.html`) provides a simple way to allocate processes to parts of the codes that perform similar tasks so that they can be performed in a parallel fashion. For example, sorting operations on large datasets can be performed in a parallel fashion to reduce time. Similar mathematical calculations on different data points can be performed in a parallel fashion.

10.10 Summary

The various features of Python that we have discussed prove that it is a worthy candidate for an all-in-one solution for scientific tasks. Since Python is open source, the vast number of libraries help to reduce development times and costs. The availability to connect to databases of large varieties, to hardwares of varied configurations, and to soures on the Internet makes Python a favored option for scientific computation. In this chapter, we have not ilustrated the usage of the individual module for numerical computing and scientific work in general, but, instead, we have convered various facilities due to the limitations of the scope of one book.

In this book, we have illustrated the usage of the Python3 programming language to a beginner, specifically targeting engineers and scientists. Almost all branches of science and engineering requires numerical computation. Python is one alternative to perform numerical computation. Python has a library of optimized functions for general computation. Also, it has a variety of packages to perform a specialized job. This makes it an ideal choice for prototyping a numerical computation problem efficiently. Moreover, it has thousands of libraries for specific scientific tasks, both software- and hardware-oriented. For this reason, Python is being taught at most university to students of engineering and science. The community of developers is exponentially increasing. In the near future, don't be surprised if Python takes over the world!

10.11 Bibliography

[1]   https://pypi.python.org/pypi.

[2]   https://www.python.org/dev/peps/pep-0249/.

[3]   https://pyvisa.readthedocs.org/en/stable/.

Index

Q

W, X, Y

Z