Chapter 29
Introduction to Python

Peeyush Taori and Hemanth Kumar Dasararaju

1 Introduction

As data science is increasingly being adopted in the industry, the demand for data scientists is also growing at an astonishing pace. Data scientists are a rare breed of "unicorns" who are required to be omniscient and according to popular culture, a data scientist is someone who knows more statistics than a programmer and more programming than a statistician. One of the most important tools in a data scientist's toolkit is the knowledge of a general-purpose programming language that enables a data scientist to perform tasks of data cleaning, data manipulation, and statistical analysis with ease. Such requirements call for programming languages that are easy enough to learn and yet powerful enough to accomplish complex coding tasks. Two such de facto programming languages for data science used in industry and academia are Python and R.

In this chapter, we focus on covering the basics of Python as a programming language. We aim to cover the important aspects of the language that are most critical from the perspective of a budding data scientist. A detailed knowledge of Python can be gained through an excellent collection of books and Internet resources. Although prior programming experience is helpful, this chapter does not require any prior knowledge of programming.

P. Taori (✉)
London Business School, London, UK
e-mail: taori.peeyush@gmail.com

H. K. Dasararaju
Indian School of Business, Hyderabad, Telangana, India

B. Pochiraju, S. Seshadri (eds.), *Essentials of Business Analytics*, International Series in Operations Research & Management Science 264,
https://doi.org/10.1007/978-3-319-68837-4_29

1.1 What Is Python?

Python is a high-level, general-purpose programming language that was first introduced to the world in late 1980s. Although the name of the language comes across as a bit odd at first, the language (or its concepts) does not bear any resemblance to an actual Python and instead was named after its creator Van Guido Rossum's inspiration from a BBC Comedy series named "*Monty Python's Flying Circus*." Initially, Python was received mostly as a general-purpose scripting language, and it was used quite extensively as a language of choice for web programming, and scripting purposes. Over the past decade, it was realized that Python could be a great tool for the scientific computing community, and since then the language has seen an explosive growth in scientific computing and data analytics applications.

Python is an open-source language that allows anyone to contribute to Python environment by creating packages and making them available to other users. Python has a fairly large scientific community and is used in a variety of settings such as financial research, algorithms development, options and derivatives pricing, financial modeling, and trading systems.

1.2 Why Python for Data Science?

As stated at the start, Python is one of the de facto languages when it comes to data science. There are a number of reasons why Python is such a popular language among data scientists. Some of those reasons are listed below:

- Python is a high-level, general-purpose programming language that can be used for varied programming tasks such as web scraping, data gathering, data cleaning and manipulation, website development, and for statistical analysis and machine learning purposes.
- Unlike some of the other high level programming languages, Python is extremely easy to learn and implement, and it does not require a degree in computer science to become an expert in Python programming.
- Python is an *object-oriented programming* language. It means that everything in Python is an object. The primary benefit of using an object-oriented programming language is that it allows us to think of problem solving in a simpler and real-world manner, and when the code becomes too cumbersome then object-oriented languages are the best way to go.
- Python is an open-source programming language. This implies that a large community of developers contribute continually to the Python ecosystem.
- Python has an excellent ecosystem that comprises of thousands of modules and libraries (prepackaged functions) that do not require reinvention of the wheel, and most of the programming tasks can be handled by simply calling functions in one of these packages. This reduces the need for writing hundreds of lines of code, and makes development easier and faster.

- Python is an interpreted language that is platform independent. As compared to some of the other programming languages, you do not have to worry about underlying hardware on which the code is going to run. Platform independence essentially ensures that your code will run in the same manner on any platform/hardware that is supported by Python.

1.3 Limits of Python

While Python is a great programming language meant for general-purpose and scientific computing tasks, there are some limitations associated with Python. For most part, these limitations are not of concern for researchers. Although there are a number of statistical and econometric packages available for Python that make analysis very easy, there might be some specific functionality that might not be available in Python. In such cases, functions can be easily written to implement the functionality and distributed among the community for use. Alternatively, Python can be integrated with other programming languages/platforms, such as R, to make up for any functionality that is available in other platforms.

2 Installation and System Interface

There are multiple ways of installing the Python environment and related packages on your machine. One way is to install Python, and then add the required packages one by one. Another method (recommended one) is to work with an Integrated Development Environment (IDE). Working with an IDE greatly simplifies the task of developing, testing, deploying, and managing your project in one place. There are a number of such IDEs available for Python such as *Anaconda* and *Enthought Canopy*, some of which are paid versions while others are available for free for academic purpose. You can choose to use any IDE that suits your specifics needs. In this particular section, we are going to demonstrate installation and usage of one such IDE: Enthought Canopy. Enthought Canopy is a comprehensive package of Python language and comes pre-loaded with more than 14,000 packages. Canopy makes it very easy to install/manage libraries, and also provides a neat GUI environment for developing applications. In this chapter, we will focus on Python installation using Enthought Canopy. Below are the guidelines on how to install Enthought Canopy distribution.

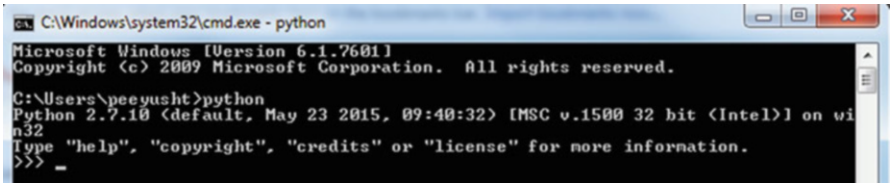

Fig. 29.1 Python working correctly on Windows

2.1 Enthought Canopy Installation

Enthought provides an academic license of Enthought Canopy distribution that is free to use for academic research purpose. You would need to register on the website[1] using your academic email ID, after which you can download and install Canopy.

- Go to https://www.enthought.com/downloads/ (accessed on Jun 19, 2018).
- Download Canopy Full Installer for the OS of your choice (Windows/Mac OS X).
- Run the downloaded file and install it by accepting default settings.
- If you are installing Canopy on OS X, make sure that Xcode is installed on your laptop. You can check if Xcode is installed by launching Terminal app (Application -> Utilities). In the Terminal, type gcc and press enter.

In order to check if Python is installed correctly, open Command Prompt, and type "python". If Python is installed correctly, you should see a message similar to the one shown in Fig. 29.1.

At the time of writing this section (June 19, 2018), the latest version of Python available is version 3.7. Other major version of Python that is used quite extensively is version 2.7. In this chapter, we demonstrate all coding examples using version 2.7 because version 3.7 of Python is not backward compatible. This implies that a number of Python packages that were developed for version 2.7 and earlier might not work very well with Python 3.7. Additionally, 2.7 is still one of the most widely used versions. While there are no drastic differences in the two versions, there are some minor differences that need to be kept in mind while developing the code.

2.2 Canopy Walkthrough

Launch the Canopy icon from your machine. There are three major components in Canopy distribution (Fig. 29.2):

1. A text editor and integrated IPython console.
2. A GUI-based package manager.
3. Canopy documentation.

[1] https://www.enthought.com/accounts/register (accessed on Jun 19, 2018)

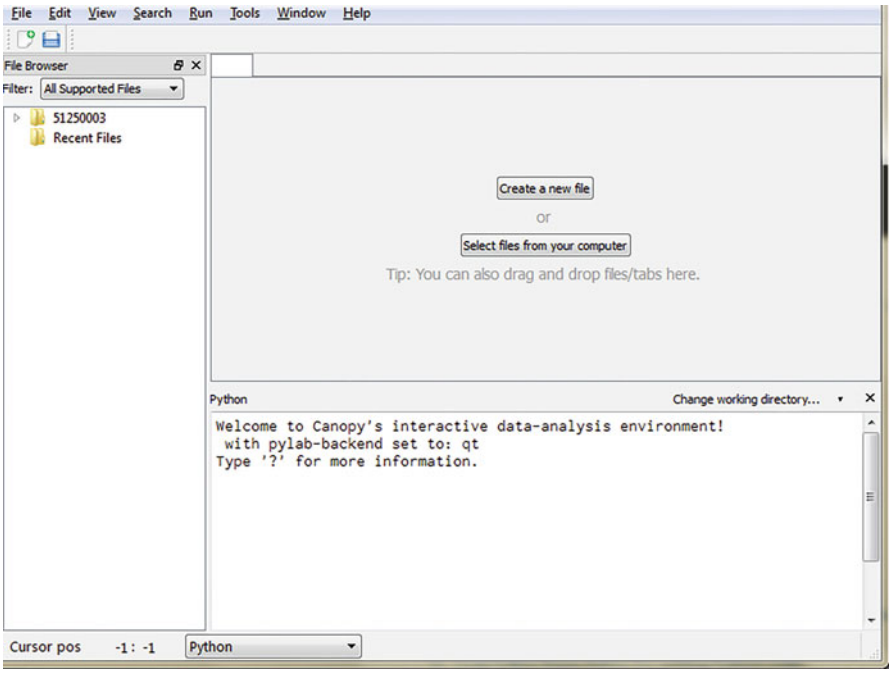

Fig. 29.2 Canopy interface

We will briefly discuss each of them.

2.2.1 Text Editor

The editor window has three major panes:

1. File Browser Pane: You can manage your Python code files here and arrange them in separate directories.
2. Code Editor Pane: Editor for writing Python code.
3. Python Pane: Contains IPython shell as well as allows you to run code directly from code editor.

2.2.2 Package Manager

The Package Manager allows you to manage existing packages and install additional packages as required. There are two major panes in Package Manager (Fig. 29.3):

1. Navigation Pane: It lists all the available packages, installed packages, and the history of installed packages.

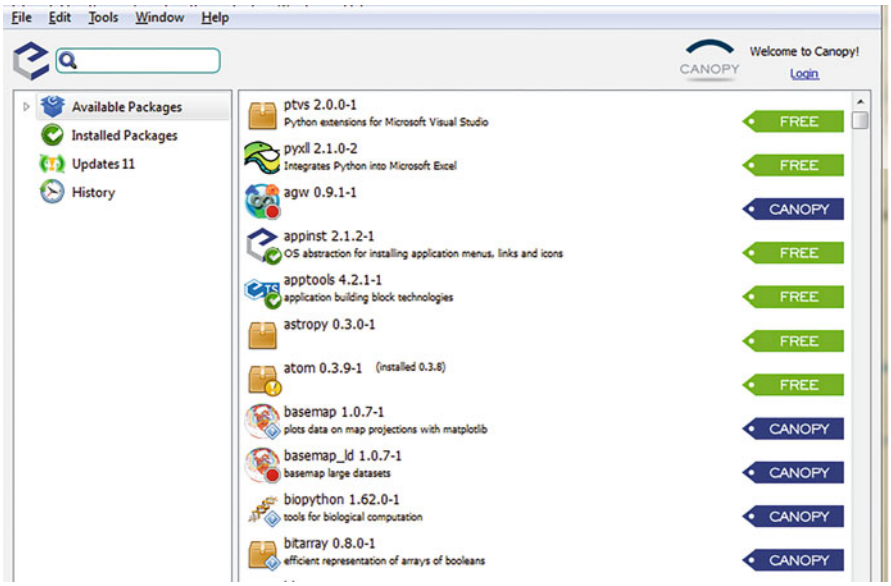

Fig. 29.3 Package Manager

2. Main Pane: This pane gives you more details about each package and allows you to manage the packages at individual level.

2.2.3 Documentation Browser

The documentation browser contains help files for Canopy software and some of the most commonly used Python packages such as Numpy, SciPy, and many more.

3 Hands-On with Python

In this section, we outline the Python programming language, and use the features and packages present in the language for data science related purposes. Specifically, we would be learning the language constructs, programming in Python, how to use these basic constructs to perform data cleaning, processing, and data manipulation tasks, and use packages developed by the scientific community to perform data analysis. In addition to working with structured (numerical) data, we will also be learning about how to work with unstructured (textual) data because Python has a lot of features to deal with both domains in an efficient manner.

We discuss the basic constructs of the language such as operators, data types, conditional statements, and functions, and specific packages that are relevant for data analysis and research purpose. In each section, we discuss a topic, code snippets, and exercise related to the sessions.

3.1 Programming Modes

There are two ways to write the codes in Python: *script* and *interactive*. The scripts mode is the one that most of the programmers would be familiar with, that is, all of the Python code is written in one text file and the file then executes on a Python interpreter. All Python code files must have a "**.py**" extension. This signals the interpreter that the file contains Python code. In the interactive mode, instead of writing all of the code together in one file, individual snippets of code are written in a command line shell and executed. Benefit of the interactive mode is that it gives immediate feedback for each statement, and makes program development more efficient. A typical practice is to first write snippets of code in interactive mode to test for functionality and then bundle all pieces of code in a .py file (Script mode). Enthought Canopy provides access to both modes. The top window in the text editor is where you can type code in script mode and run all or some part of it. In order to run a file, just click on the Run button in the menu bar and Python will execute the code contained in the file.

The bottom window in text editor acts as the Python interactive shell. In interactive mode what you type is immediately executed. For example, typing $1 + 1$ will respond with 2.

3.2 Syntax Formalities

Let us now get started with understanding the syntax of Python. The Python community prides itself in writing the code that is obvious to understand even for a beginner—this specific way is known as "Pythonic" in nature. Although it is true that Python is a very simple and easy language to learn and develop, it has some quirks—the biggest one of which is indentation. Let us first understand the importance of indentation before we start to tackle any other syntax features of the language. Please note that all the codes referred in the following sections are tested on Python 2.7 in the Enthought Canopy console.

3.3 Indentation

Whitespace is important in Python—this is known as *indentation*. Python makes use of whitespace for code structuring and marking logical breaks in the code. This is in contrast with other programming languages such as *R*, *Java*, and *C* that use braces for code blocks. Level of indentation of any code block is used to determine whether the code is part of the main program flow or whether it belongs to a particular branch of the program. Leading whitespaces such as tab and spaces are used for program indentation, and a group of statements that have the same indentation are considered to belonging to the same code block. If a particular code block is indented, then it must belong to a branch of the main program that has to be executed if a certain condition (such as if, for loops; more on them later) associated with the code block is met. Let us understand indentation with the help of simple examples:

Code

```
a = 7
       print ('Value of the variable is {}'.format(a))
         # Error! Look at the space at the beginning of line
print ('This is now correct. Value of variable a is {}'.format(a))
```

Output

```
(Once you comment second line)
This is now correct. Value of variable a is 7
```

In the above piece, each line is a Python statement. In the first statement, we are assigning a value of 7 to the variable "a." In the second statement, notice the space at the beginning. This is considered as indentation by Python interpreter. However, since any indentation block is supposed to have a conditional statement (such as if and for loop), the code here would give an error as the interpreter will consider the second statement as having a separate flow from the first statement. The third statement does not have any indentation (it is in the same block as the first statement) and thus will execute just fine.

It is important to remember that all statements that are expected to execute in the same block should have the same indentation. Presence of indentations improves readability of Python programs tremendously but it also requires a bit getting used to, especially if you are coming from languages such as C and Java where semicolon (**;**) marks the end of statements. You should also be careful with indentations because if you are not careful with them then they can cause errors in the program to say the least, and if gone undetected they can cause program to behave in an unpredictable manner. Most of the IDEs such as *Canopy* and *Anaconda* have in-built support for indentations that make program development easier.

3.4 Comments

Comments are required in any programming language to improve readability by humans. Comments are those sections of code that are meant for human comprehension, and are ignored by the Python interpreter when executing. In Python, you can write either single-line or multiline comments.

1. **Single-line comment**:
   A single line comment in Python begins with a pound (#) sign. Everything after the # sign is ignored by the interpreter till the end of line.

Code

```
print("This is code line, not a comment line")
#print("This is a comment line")
```

Output

```
This is code line, not a comment line
```

Note that in the above code snippet, the first line is the actual code that is executed, whereas the second line is a comment that is ignored by interpreter.

2. **Multiline comment**:
   Syntax for multiline comments is different from that of single-line comment. Multiline comments start and end with three single quotes (“’). Everything in between is ignored by the interpreter.

Code

```
“’
print("Multi line comment starts from here“)
print ("Multi line comment continuing. This will not be printed“)
”’
print("Multi line comment ended in above line. This line with
        be printed")
```

Output

```
Multi line comment ended in above line. This line with be printed
```

3.5 Object Structure

With a firm understanding of indentation and comments, let us now look at the building blocks of the Python programming language. A concept central to Python is that of object. Everything in Python, be it a simple variable, function, custom data structure is an object. This means that there is data and certain functions associated with every object. This makes programming very consistent and flexible. However, it does not imply that we have to think of objects every time we are coding in Python.

Behind the scenes everything is an object even if we explicitly use objects or not in our coding. Since we are just getting started, we will first focus on coding without objects and talk about objects later, once we are comfortable with the Pythonic way of programming.

3.6 Variables

There are some in-built data types in Python for handling different kinds of data: integer, floating point, string, Boolean values, date, and time. A neat feature of Python is that you do not need to mention what kind of data a variable holds; depending on the value assigned, Python automatically assigns a data type to the variable.

Think of a variable as a placeholder. It is any name that can hold a value and that value can vary over time (hence the name variable). In other terms, variables are reserved locations in your machine's memory to store different values. Whenever you specify a variable, you are actually allocating space in memory that will hold values or objects in future. These variables continue to exist while the program is running. Depending on the type of data a variable has, the interpreter will assign the required amount of memory for that variable. This implies that memory of a variable can increase or decrease dynamically depending on what type of data the variable has at the moment. You create a variable by specifying a name to the variable, and then by assigning a value to the variable by using equal sign (=) operator.

Code

```
variable1 = 100     # Variable that holds integer value
distance = 1500.0   # Variable that holds floating point value
institute = "ISB"   # Variable that holds a string
print(variable1)

print(distance)
print(institute)
print institute     # print statement has been discontinued
                       from Python3
```

Output

```
100
1500.0
ISB
ISB
```

Code

```
a = 0
b = 2
c = "0"
d = "2"
print(a + b)   # output as integer
print(c + d)   # output as string
print(type(a + b))
print(type(c + d))
```

Output

```
2
02
<type 'int'>
<type 'str'>
```

3.7 Naming Conventions for a Variable

Although a variable can be named almost anything, there are certain naming conventions that should be followed:

- A variable can start with either a letter (uppercase or lowercase) or an underscore (_) character.
- Remainder of the variable can contain any combination of letters, digits, and underscore characters.
- For example, some of the valid names for variables are _variable, variable1. 5Variable, >Smiley are not correct variable names.
- Variable names in Python are case-sensitive. This means that Variable and variable are two different variables.

3.8 Basic Data Types

In addition to complex data types, Python has five atomic (basic) data types. They are Number, String, List, Tuple, and Dictionary, respectively. Let us understand them one by one.

3.8.1 Numbers

Numbers are used to hold numerical values. There are four types of numbers that are supported in Python: integer, long integer, floating point (decimals), and complex numbers.

1. **Integer**: An integer type can hold integer values such as 1, 4, 1000, and -52,534. In Python, integers have a bit length of minimum 32 bits. This means that an integer data type can hold values in the range −2,147,483,648 to 2,147,483,647. An integer is stored internally as a string of digits. An integer can only contain digits and cannot have any characters or punctuations such as $.

Code and output

```
>>> 120+200
320
```

```
>>> 180-42
138
>>> 15*8
120
```

2. **Long Integer**: Simple integers have a limit on the value that they can contain. Sometimes the need arises for holding a value that is outside the range of integer numbers. In such a case, we make use of Long Integer data types. Long Integer data types do not have a limit on the length of data they can contain. A downside of such data types is that they consume more memory and are slow during computations. Use Long Integer data types only when you have the absolute need for it. Python distinguishes Long Integer value from an integer value by character L or l, that is, a Long Integer value has "L" or "l" in the end.

Code and output

```
>>> 2**32
4294967296L
```

3. **Floating Point Numbers**: Floating point data types are used to contain decimal values such as fractions.
4. **Complex Numbers**: Complex number data types are used to hold complex numbers. In data science, complex numbers are used rarely and unless you are dealing with abstract math there would be no need to use complex numbers.

3.8.2 Strings

A key feature of Python that makes it one of the de facto languages for text analytics and data science is its support for strings and string processing. Strings are nothing but an array of characters. Strings are defined as a sequence of characters enclosed by quotation marks (they can be single or double quotes). In addition to numerical data processing, Python has very strong string processing capabilities. Since strings are represented internally as an array of characters, it implies that it is very easy to access a particular character or subset of characters within a string. A sub-string of a string can be accessed by making use of indexes (position of a particular character in an array) and square brackets []. Indexes start with 0 in Python. This means that the first character in a string can be accessed by specifying string name followed by [followed by 0 followed by] (e.g., (stringname[0]). If we want to join two strings, then we can make use of the plus (+) operator. While plus (+) operator adds numbers, it joins strings and hence can work differently depending on what type of data the variables hold. Let us understand string operations with the help of a few examples.

Code

```
newstring = 'Hi. How are you?'
print(newstring)       # It will print entire string
print(newstring [0])   # It will print first character
```

```
print(newstring [3:7]) # It will print from 4th to 6th character
print(newstring [3:])  # It will print everything from 4th
                         character to end
print(newstring * 3)   # It will print the string three times
print(newstring + "I am very well, ty.") # It will concatenate
                                           two strings
```

Output

```
Hi. How are you?
H
 How
 How are you?
Hi. How are you?Hi. How are you?Hi. How are you?
Hi. How are you?I am very well, ty.
```

Strings in Python are immutable. Unlike other datasets such as lists, you cannot manipulate individual string values. In order to do so, you have to take subsets of strings and form a new string. A string can be converted to a numerical type and vice versa (wherever applicable). Many a times, raw data, although numeric, is coded in string format. This feature provides a clean way to make sure all of the data is in numeric form. Strings are a sequence of characters and can be tokenized. Strings and numbers can also be formatted.

3.8.3 Date and Time

Python has a built-in datetime module for working with dates and times. One can create strings from date objects and vice versa.

Code

```
import datetime
date1 = datetime.datetime(2014, 5, 16, 14, 45, 05)
print(date1.day)
print(date1)
```

Output

```
16
2014-05-16 14:45:05
```

3.8.4 Lists

Lists in Python are one of the most important and fundamental data structures. At the very basic level, List is nothing but an ordered collection of data. People with background in Java and C can think of list as an array that contains a number of elements. The difference here is that a list can contain elements of different data types. A list is defined as a collection of elements within square brackets "[ ]", and each element in a list is separated by commas. Similar to the individual characters

in a String, if you want to access individual elements in a list then you can do it by using the same terminology as used with strings, that is, using the indexes and the square brackets.

Code

```
alist = [ 'hi', 123 , 5.45, 'ISB', 85.4 ]
anotherlist = [234, 'ISB']

print(alist)        # It will print entire list
print(alist[0])     # It will print first element
print(alist[2:5])   # It will print 3rd through 5th element in list
print(alist[3:])    # It will print from element 4 till the end
print(anotherlist * 2) # It will print the list twice
print(alist + anotherlist) # It will concatenate the two lists
```

Output

```
['hi', 123, 5.45, 'ISB', 85.4]
hi
[5.45, 'ISB', 85.4]
['ISB', 85.4]
[234, 'ISB', 234, 'ISB']
['hi', 123, 5.45, 'ISB', 85.4, 234, 'ISB']
```

3.8.5 Tuples

A tuple is an in-built data type that is a close cousin to the list data type. While in a list you can modify individual elements of the list and can also add/modify the number of elements in the list, a tuple is immutable in the sense that once it is defined you cannot change either the individual elements or number of elements in a tuple. Tuples are defined in a similar manner as lists with a single exception—while lists are defined in square brackets "[]", tuples are defined using parenthesis "()". You should use tuples whenever there is a situation where you need to use lists that nobody should be able to modify.

Code

```
tupleone = ('hey', 125, 4.45, 'isb', 84.2)
tupletwo = (456, 'isb')

print(tupleone)       # It will print entire tuple
print(tupleone[0])    # It will print first element of tuple
print(tupleone[1:4])  # It will print 2nd to 4th element of tuple
print(tupleone[3:])   # It will print entire tuple from 4th
                                element till last
print(tupletwo * 2)   # It will print tuple twice
print(tupleone + tupletwo) # It will concatenate and print
                                the two tuples
```

Output

```
('hey', 125, 4.45, 'isb', 84.2)
'hey'
```

```
(125 , 4.45, 'isb')
('isb', 84.2)
(456, 'isb', 456, 'isb')
('hey', 125, 4.45, 'isb', 84.2, 456, 'isb')
```

If you try to update a tuple, then it would give you an error:

Code

```
tuple = ('hey', 234, 4.45, 'Alex', 81.4)
list = ['hey', 234, 4.45, 'Alex', 81.4]
tuple[2] = 1000    # Invalid (error: tuple object does not
                      support...)
list[2] = 1000     # Valid (it will change 4,45 to 1000)
```

Output

```
['hey', 234, 1000, 'Alex', 81.4]
```

3.8.6 Dictionary

Perhaps one of the most important built-in data structures in Python are dictionaries. Dictionaries can be thought of as arrays of elements where each element is a key–value pair. If you know the key or value then you can quickly look up for corresponding values/key respectively. There is no restriction on what key or values could be, and they can assume any Python data type. Generally, as industry practice we tend to use keys as containing either numbers or characters. Similar to keys, values can assume any data type (be it basic data types or complex ones). When you need to define a dictionary, it is done using curly brackets "{}" and each element is separated by a comma. An important point to note is that dictionaries are unordered in nature, which means that you cannot access an element of a dictionary by using the index, but rather you need to use keys.

Code

```
firstdict = {}
firstdict ['one'] = "This is first value"
firstdict [2]     = "This is second value"
seconddict = {'institution': 'isb','pincode':500111,
               'department': 'CBA'}

print(firstdict ['one'])
print(firstdict [2] )
print(seconddict)
print(seconddict.keys())   # It will print all keys in
                              the dictionary
print(seconddict.values()) # It will print all values in
                               the dictionary
```

Output

```
This is first value
This is second value
```

```
{'institution': 'isb','pincode':500111, 'department': 'CBA'}
['department', 'pincode', 'institution']
['CBA', 500111, 'isb']
```

3.9 Datatype Conversion

Quite often the need might arise where you need to convert a variable of a specific data type to another data type. For example, you might want to convert an int variable to a string, or a string to an int, or an int to a float. In such cases, you use type conversion operators that change the type of a variable. To convert a variable to integer type, use int(variable). To convert a variable to a string type, use str(variable). To convert a variable to a floating point number, use float(variable).

3.10 Python Operators

Operators in Python perform operations on two variables/data values. Depending on what type of data the variable contains, the operations performed by the same operator could differ. Listed below are the different operators in Python:

+ **(plus)**: It would add two numbers or variables if they are numbers. If the variables are string, then they would be concatenated. For example,

  4 + 6 would yield 10. 'Hey' + 'Hi' would yield 'HeyHi'.

− **(minus)**: It would subtract two variables.

* **(multiply)**: It would multiply two variables if they are numbers. If the variables are strings/lists, then they would be repeated by a said number of times. For example,

  3 * 6 would yield 18; '3'*4 would yield '3333'; 'ab*4' would yield 'abababab'.

** **(power)**: It computes x raised to power y. For example,

  4 ** 3 would yield 64 (i.e., 4 * 4 * 4)

/ **(divide)**: It would divide x by y.

// **(floor division)**: It would give the floor in a division operation. For example,

  5 // 2 would yield 2.

% **(modulo)**: Returns the remainder of the division. For example,

  8 % 3 gives 2. −25.5 % 2.25 gives 1.5.

< **(less than)**: Returns whether x is less than y. All comparison operators return True or False. Note the capitalization of these names. For example,

5 < 3 gives False and 3 < 5 gives True.

> (greater than): Returns whether x is greater than y. For example,

5 > 3 returns True. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns False.

<= (less than or equal to): Returns whether x is less than or equal to y. For example,

x = 3; y = 6; x <= y returns True.

>= (greater than or equal to): Returns whether x is greater than or equal to y. For example,

x = 4; y = 3; x >= 3 returns True.

== (equal to): Compares if the objects are equal. For example,

x = 2; y = 2; x == y returns True.
x = 'str'; y = 'stR'; x == y returns False.
x = 'str'; y = 'str'; x == y returns True.

!= (not equal to): Compares if the objects are not equal. For example,

x = 2; y = 3; x! = y returns True.

not (boolean NOT): If x is True, it returns False. If x is False, it returns True. For example,

x = True; not x returns False.

and (boolean AND): x and y returns False if x is False, else it returns evaluation of y. For example,

x = False; y = True; x and y returns False since x is False

or (boolean OR): If x is True, it returns True, else it returns evaluation of y. For example,

x = True; y = False; x or y returns True.

3.11 Conditional Statements and Loops

After the discussion of variables and data types in Python, let us now focus on the second building block of any programming language, that is, conditional statements. Conditional statements are branches in a code that are executed if a condition associated with the conditional statements is true. There can be many different types of conditional statements; however, the most prominent ones are if, while, and for. In the following sections, we discuss these conditional statements.

3.11.1 if Statement

We use an if loop whenever there is a need to evaluate a condition once. If the condition is evaluated to be true, then the code block associated with if condition is executed, otherwise the interpreter skips the corresponding code block. The condition along with the associated set of statements are called the if loop or if block. In addition to the if condition, we can also specify an else block that is executed if the if condition is not successful. Please note that the else block is entirely optional.

Code

```
var1 = 45
if var1 >= 43:
       print("inside if block")
elif var1 <= 40:
       print("inside elif block")
else:
       print("inside else block")
```

Output

```
inside if block
```

3.11.2 while Loop

Whereas an if loop allows you to evaluate the condition once, the while loop allows you to evaluate a condition multiple number of times depending on a counter or variable that keeps track of the condition being evaluated. Hence, you can execute the associated block of statements multiple times in a while block. Similar to an if loop, you can have an optional else loop in the while block, see for loop example below:

Code

```
counter = 0
while (counter < 5):
         print('Current counter: {}'.format(counter))
         counter = counter + 1

print("While loop ends!")
```

Output

```
Current counter: 0
Current counter: 1
Current counter: 2
Current counter: 3
Current counter: 4
While loop ends!
```

3.11.3 for Loop

In many ways for loop is similar to a while loop in the sense that it allows you to iterate the loop multiple times depending on the condition being evaluated. However, for loop is more efficient in the sense that we do not have to keep count of incrementing or decrementing the counter of condition being evaluated. In the while loop, onus is on the user to increment/decrement the counter otherwise the loop runs until infinity. However, in for loop the loop itself takes care of the increment/decrement.

Code

```
for a in range(1,8):
       print (a)
else:
       print('For loop ends')
```

Output

```
1
2
3
4
5
6
7
For loop ends
```

3.11.4 break Statement

Sometimes the situation might arise in which you might want to break out of a loop before the loop finishes completion. In such cases, we make use of the break statement. The break statement will break out of the loop whenever a particular condition is being met.

Code

```
for a in range(1,8):
       if a == 4:
             break
       print(a)

print('Loop Completed')
```

Output

```
1
2
3
Loop completed
```

3.11.5 Continue statement

Whereas the *break* statement completely skips out of the loop, the *continue* statement skips the rest of the code lines in a current loop and goes to the next iteration.

Code

```
while True:
       string = input('Type your input: ')
       if string == 'QUIT':
             break
       if len(string) < 6:
             print 'String is small'
             continue
       print('String input is not sufficient')
```

Output

```
Type your input: 'Hi'
String is small

Type your input: 'abc'
String is small

Type your input: 'verylarge'
String input is not sufficient

Type your input: 'QUIT'
```

3.12 Reading Input from Keyboard

Whenever you need user to enter input from keyboard or you need to read keyboard input, you can make use of two in-built functions—"raw_input" and "input." They allow you to read text line from standard keyboard input.

3.12.1 raw_input Function

It will read one line from keyboard input and give it to the program as a string.

```
string= raw_input(‘Provide the input: ‘);
print(‘Input provided is: {}‘.format(string))
```

In the above-mentioned example, the user would get a prompt on the screen with title "Provide the input." The second line will then print whatever input the user has provided.

```
Provide the input: Welcome to Python
Input provided is: Welcome to Python
```

3.12.2 Input Function

"input" is similar to raw_input() with an exception. While "raw_input" assumes that the entered value is a text, "input" would assume that entered text is indeed a Python expression and will proceed to evaluate the Python expression and would provide the output of the expression.

```
string= input('Provide the input: ');
print('Input provided is: {}'.format(string))
```

For example:

```
Provide the input: [a*3 for a in range(1,6,2)]
Input provided is: [1,9,15]
```

3.13 Working with Files

Most of the times, in addition to using variables and in-built data structures, we would be working with external files to get data input and to write output to. For this purpose, Python provides functions for file opening and closing. A key concept in dealing with files is that of a "file" object. Let us understand this in a bit more detail.

3.13.1 open() Function

File object is the handle that actually allows you to create a link to the file you want to read/write to. In order to be able to read/write a file, we first need to use an object of file type. In order to do so, we make use of open() method. When open() executes, it will result in a file object that we would then use to read/write data for external files.

Syntax

```
file_object = open(file_name [, access_mode][, buffering])
```

Let us understand this function in slightly more detail. The file_name requires us to provide the file name that we want to access. You can specify either an existing file on the filesystem or you can specify a new file name as well. Access_mode tells Python in which mode the file should be opened. There are a number of modes to do so; however, the most common ones are read, write, and append. A more detailed knowledge of each mode type is given in Table 29.1. Finally, buffer mode tells us how to buffer the data. By default, value for buffer is 0. This means that there is no buffering. If it is 1, then it implies that there would be buffering whenever a file is being accessed.

Table 29.1 Access_modes list

Mode	Brief overview
r	Default mode. Opens file in read only mode with pointer at the start of file.
rb	Similar to r. Only difference being that the file being read in binary format.
r+	When this mode is used, then file can be used for both reading and writing.
rb+	Similar to r+ except that reading and writing will happen in the binary format.
w	File can be accessed for writing only. Creates a new file if there is no existing file with the same name.
wb	Same as w except that it is opened in binary format.
w+	Similar to r+.
wb+	Similar to rb+.
a	When this mode is used, data is appended to the file. In w mode, data is being overwritten. Pointer in this mode is at the end of file rather than at the beginning.
ab	Similar to a except that it is in binary mode.
a+	Similar to w+ with append features.
ab+	Similar to wb+ with append features.

3.13.2 close() Function

Once we have opened the file for reading/writing purposes, we would then need to close the connection with the file. It is done using the close() method. close() will flush out any unwritten data to the file and will close the file object that we had opened earlier using open() function. Once the close() method is called, we cannot do any more reads/writes on the file. In order to do so, we would again have to open the file using open() method.

Syntax

```
file_object.close();
```

Code

```
# File Open
file1 = open('transactions.txt', 'wb')
print('File Name: {}'.format(file1.name))

# Close file
file1.close()
```

Output

```
File Name: transactions.txt
```

3.13.3 Reading and Writing Files

While the open() and close() methods allow us to open/close a connection to a file, we need to make use of read() or write() methods to actually read or write data to a file.

3.13.4 write() Function

When we call write(), it will write the data as a string to the file that we opened earlier.

Syntax

```
file_object.write(string);
```

String here is the data that has to be written to the file.

Code

```
# File Open
file1 = open('sample.txt', 'wb')
file1.write('This is my first output.\nIt looks good!!\n');

# Close file
file1.close()
```

When we run the above code, a file sample.txt would be created and the string mentioned in write() function would be written to the file. The string is given by:

```
This is my first output.
It looks good!!
```

3.13.5 read() Function

Just as write() method writes data to a file, read() would read data from an open file.

Syntax

```
file_object.read([counter]);
```

You would notice that we have passed an argument called counter here. When we do this, then it tells the interpreter to read the specified number of bytes. If no such argument is provided, then the reading will read the entire text.

Code

```
# File Open
fileopen = open('sample.txt', 'r+')
string = fileopen.read(10);
print('Output is: {}'.format(string))

# Close file
fileopen.close()
```

Output

```
Output is: This is my
```

3.14 Build Custom Function

While Python is a great programming language with a number of in-built functions (such as those for printing, file reads and writes), oftentimes you would need to write your own piece of functionality that is not available elsewhere (e.g., you might want to write a specific piece of logic pertaining to your business). For such cases, rather than write the same code again at multiple places in code, we make use of functions. Functions are nothing but reusable code pieces that need to be written once, and can then be called using their names elsewhere in the code. When we need to create a function, we need to give a name to the function and the associated code block that would be run whenever the function is called. A function name follows the same naming conventions as a variable name. Any function is defined using keyword def. This tells the interpreter that the following piece of code is a function. After def, we write the function names along with parentheses and any arguments that the function would expect within the parenthesis. Following this, we would write the code block that would be executed every time the function is called.

Code

```
def firstfunc():
     print('Hi Welcome to Python programming!')
         # code block that is executed for the function
# Function ends here #

firstfunc () # Function called first time
firstfunc () # Called again
```

Output

```
Hi Welcome to Python programming!
Hi Welcome to Python programming!
```

In the above-mentioned code snippet, we created a function "*firstfunc*" using the syntax of the function. In this case, the function expects no parameters and that is why we have empty parentheses. Function arguments are the variables that we pass to the function that the function would then use for its processing. Note that the names given in the function definition are called **parameters**, whereas the values you supply in the function call are called **arguments**.

Code

```
def MaxFunc(a1, a2):
       if a1 > a2:
             print('{} is maximum'.format(a1))
       elif a1 == a2:
             print('{} is equal to {}'.format(a1, a2))
       else:
             print('{} is maximum'.format(a2))

MaxFunc (8, 5) # directly give literal values
x = 3
```

```
y = 1
MaxFunc (x, y) # give variables as arguments
MaxFunc (5, 5) # directly give literal values
```

Output

```
8 is maximum
3 is maximum
5 is equal to 5
```

In the above code snippet, we created a function "*MaxFunc*." MaxFunc requires two parameters (values) a1 and a2. The function would then compare the values and find the maximum of two values. In the first function call, we directly provided the values of 8 and 5 in the function call. Second time, we provided the variables rather than values for the function call.

3.14.1 Default Value of an Argument

If you want to make some parameters of a function optional, use default values in case the user does not want to provide values for them. This is done with the help of default argument values. You can specify default argument values for parameters by appending to the parameter name in the function definition the assignment operator (=) followed by the default value. Note that the default argument value should be a constant. More precisely, the default argument value should be immutable.

Code

```
def say(message, times = 1):
    print(message * times)

say('Hello')
say('World', 5)
```

Output

```
Hello
WorldWorldWorldWorldWorld
```

The function named "say" is used to print a string as many times as specified. If we do not supply a value, then by default, the string is printed just once. We achieve this by specifying a default argument value of 1 to the parameter times. In the first usage of *say*, we supply only the string and it prints the string once. In the second usage of *say*, we supply both the string and an argument 5 stating that we want to say the string message five times.

Only those parameters that are at the end of the parameter list can be given default argument values, that is, you cannot have a parameter with a default argument value preceding a parameter without a default argument value in the function's parameter list. This is because the values are assigned to the parameters by position. For example, def func(a, b=5) is valid, but def func(a=5, b) is not valid.

3.14.2 Return Statement

The "*return*" statement is used to return from a function, that is, break out of the function. You can optionally return a value from the function as well.

Code

```
def maximum(x, y):
       if x > y:
             return x
       elif x == y:
             return 'The numbers are equal'
       else:
             return y
print(maximum(2, 3))
```

Output

```
3
```

3.15 Modules

You can reuse code in your program by defining functions once. If you want to reuse a number of functions in other programs that you write, you can use modules. There are various methods of writing modules, but the simplest way is to create a file with a ".py" extension that contains functions and variables.

Another method is to write the modules in the native language in which the Python interpreter itself was written. For example, you can write modules in the C programming language and when compiled, they can be used from your Python code when using the standard Python interpreter.

A module can be imported by another program to make use of its functionality. This is how we can use the Python standard library as well. The following code demonstrates how to use the standard library modules.

Code

```
import os
print os.getcwd()
```

Output

```
<Your current working directory>
```

3.15.1 Byte-Compiled .pyc Files

Importing a module is a relatively costly affair, so Python does some tricks to make it faster. One way is to create byte-compiled files with the extension ".pyc", which is an intermediate form that Python transforms the program into. This "*.pyc*" file is

useful when you import the module the next time from a different program—it will be much faster since a portion of the processing required in importing a module is already done. Also, these byte-compiled files are platform-independent.

Note that these ".pyc" files are usually created in the same directory as the corresponding ".py" files. If Python does not have permission to write to files in that directory, then the ".pyc" files will not be created.

3.15.2 from . . . import Statement

If you want to directly import the "*argv*" variable into your program (to avoid typing the sys. everytime for it), then you can use the "from sys import argv" statement. In general, you should avoid using this statement and use the import statement instead since your program will avoid name clashes and will be more readable.

Code

```
from math import sqrt
print('Square root of 16 is {}'.format(sqrt(16)))
```

3.15.3 Build Your Own Modules

Creating your own modules is easy; this is because every Python program is also a module. You just have to make sure that it has a ".py" extension. The following is an example for the same:

Code (save as mymodule.py)

```
def sayhi():
       print('Hi, this is mymodule speaking.')__version__='0.1'
```

The above was a sample module; there is nothing particularly special about it compared to our usual Python program. Note that the module should be placed either in the same directory as the program from which we import it, or in one of the directories listed in sys.path.

Code (Another module—save as mymodule_demo.py)

```
import mymodule
mymodule.sayhi()
print("version {}".format(mymodule.__version__))
```

Output

```
Hi, this is mymodule speaking.
Version 0.1
```

3.16 Packages

In the hierarchy of organizing your programs, variables usually go inside functions. Functions and global variables usually go inside modules. What if you wanted to organize modules? This is where packages come into the picture.

Packages are just folders of modules with a special __init__.py file that indicates to Python that this folder is special because it contains Python modules. Let us say you want to create a package called "world" with subpackages "asia," "africa," etc., and these subpackages in turn contain modules like "india," "madagascar," etc. Packages are just a convenience to hierarchically organize modules. You will see many instances of this in the standard library.

3.16.1 Relevant Packages

There are a number of statistical and econometric packages available on the Internet that can greatly simplify the research work. Following is the list of widely used packages:

1. NumPy: Numerical Python (NumPy) is the foundation package. Other packages and libraries are built on top of NumPy.
2. pandas: Provides data structures and processing capabilities similar to ones found in R and Excel. Also provides time series capabilities.
3. SciPy: Collection of packages to tackle a number of computing problems in data analytics, statistics, and linear algebra.
4. matplotlib: Plotting library. Allows to plot a number of 2D graphs and will serve as primary graphics library.
5. IPython: Interactive Python (IPython) shell that allows quick prototyping of code.
6. Statsmodels: Allows for data analysis, statistical model estimation, statistical tests, and regressions, and function plotting.
7. BeautifulSoup: Python library for trawling the Web. Allows you to pull data from HTML and XML pages.
8. Scikits: A number of packages for running simulations, machine learning, data mining, optimization, and time series models.
9. RPy: This package integrates R with Python and allows users to run R code from Python. This package can be really useful if certain functionality is not available in Python but is available in R.