Python
for
Data Science

Contents

Chapter 1

Introduction

Welcome to our first module on *programming*. In this module, we will be discussing the nuts and bolts of the Python programming language ranging from the very basic to more advanced topics.

Python is a general-purpose programming language that is becoming more and more popular for

- performing data analysis,
- automating tasks,
- learning data science,
- machine learning, etc.

1.1 What is Python?

Python is a dynamic, interpreted (bytecode-compiled) language that is used in a wide range of domains and technical fields. It was developed by Guido van Rossum in 1991. It was mainly developed for code readability and its syntax is such that it allows programmers to code/express concepts in fewer lines of code. Compared to compiled languages like C, Java, or Fortran, we do not need to declare the type of variables, functions, etc. when we write code in Python. This makes our code short and flexible. Python tracks the types of all values at runtime and flags code that does not make sense as it runs. On the Python website[1], we find the following executive summary.

[1] https://www.python.org/doc/essays/blurb/

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms and can be freely distributed.

1.2 Where is Python used?

Python is used by the novice programmer as well as by the highly skilled professional developer. It is being used in academia, at web companies, in large corporations and financial institutions. It is used for

- *Web and Internet development*: Python is used on the server side to create web applications.
- *Software development*: Python is used to create GUI applications, connecting databases, etc.
- *Scientific and Numeric applications*: Python is used to handle big data and perform complex mathematics.
- *Education*: Python is a great language for teaching programming, both at the introductory level and in more advanced courses.
- *Desktop GUIs*: The Tk GUI library[2] included with most binary distributions of Python is used extensively to build desktop applications.
- *Business Applications*: Python is also used to build ERP and e-commerce systems.

1.3 Why Python?

Python is characterized by many features. Let's examine a few of them here:

[2] https://wiki.python.org/moin/TkInter

- **Simple**
  - Compared to many other programming languages, coding in Python is like writing simple strict English sentences. In fact, one of its oft-touted strengths is how Python code appears like pseudo-code. It allows us to concentrate on the solution to the problem rather than the language itself.
- **Easy to Learn**
  - As we will see, Python has a gentler learning curve (compared to languages like C, Java, etc.) due to its simple syntax.
- **Free and Open Source**
  - Python and the majority of supporting libraries available are open source and generally come with flexible and open licenses. It is an example of a FLOSS(Free/Libré and Open Source Software). In layman terms, we can freely distribute copies of open source software, access its source code, make changes to it, and use it in new free programs.
- **High-level**
  - Python is a programming language with strong abstraction from the details of the underlying platform or the machine. In contrast to low-level programming languages, it uses natural language elements, is easier to use, automates significant areas of computing systems such as resource allocation. This simplifies the development process when compared to a lower-level language. When we write programs in Python, we never need to bother about the lower-level details such as managing the memory used by programs we write, etc.
- **Dynamically Typed**
  - Types of variables, objects, etc. in Python are generally inferred during runtime and not statically assigned/declared as in most of the other compiled languages such as C or Fortran.
- **Portable/Platform Independent/Cross Platform**
  - Being open source and also with support across multiple platforms, Python can be ported to Windows, Linux and Mac

OS. All Python programs can work on any of these platforms without requiring any changes at all if we are careful in avoiding any platform-specific dependency. It is used in the running of powerful severs and also small devices like the Raspberry Pi[3].

In addition to the above-mentioned platforms, following are some of the other platforms where Python can be used

* FreeBSD OS
* Oracle Solaris OS
* AROS Research OS
* QNX OS
* BeOS
* z/OS
* VxWorks OS
* RISC OS

- **Interpreted**
  - A programming language can be broadly classified into two types viz. compiled or interpreted.
  - A program written in a compiled language like C or C++ requires the code to be converted from the original language (C, C++, etc.) to a machine-readable language (like binary code i.e. 0 and 1) that is understood by a computer using a compiler with various flags and options. This compiled program is then fed to a computer memory to run it.
  - Python, on other hand, does not require compilation to machine language. We directly *run* the program from the source code. Internally, Python converts the source code into an intermediate form known as byte code and then translates this into the native language of the underlying machine. We need not worry about proper linking and the loading into memory. This also enables Python to be much more portable, since we can run the same program onto another platform and it works just fine!
  - The 'CPython' implementation is an interpreter of the language that translates Python code at runtime to executable byte code.
- **Multiparadigm**

  - Python supports various programming and implementation paradigms, such as *Object Oriented*, *Functional*, or *Procedural* programming.

- **Extensible**
  - If we need some piece of code to run fast, we can write that part of the code in C or C++ and then use it via our Python program. Conversely, we can embed Python code in a C/C++ program to give it *scripting* capabilities.

- **Extensive Libraries**
  - The Python Standard Library[4] is huge and, it offers a wide range of facilities. It contains built-in modules written in C that provides access to system functionality such as I/O operations as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are listed below
    * Text Processing Modules
    * Data Types
    * Numeric and Mathematical Modules
    * Files and Directory Modules
    * Cryptographic Modules
    * Generic Operating System Modules
    * Networking Modules
    * Internet Protocols and Support Modules
    * Multimedia Services
    * Graphical User Interfaces with Tk
    * Debugging and Profiling
    * Software Development, Packaging and Distribution
  - In addition to the Python Standard Library, we have various other third-party libraries which can be accessed from Python Package Index[5].

- **Garbage Collection**
  - Python takes care of memory allocation and deallocation on its own. In other words, a programmer does not have to manage

memory allocation and need not have to preallocate and deallocate memory before constructing variables and objects. Additionally, Python provides Garbage Collector[6] interface to handle garbage collection.

1.4 History of Python

Python is the brainchild of Guido van Rossum who started its developmental efforts in the 1980s. Its name has nothing to do with anything serpentine, it's in fact inspired by the British comedy Monty Python! The first Python implementation was in December 1989 in the Netherlands. Since then, Python has gone through major turnarounds periodically. The following can be considered milestones in the development of Python:

- **Python 0.9.0** released in February 1991
- **Python 1.0** released in January 1994
- **Python 2.0** released in October 2000
- **Python 2.6** released in October 2008
- **Python 2.7** released in July 2010
- **Python 3.0** released in December 2008
- **Python 3.4** released in March 2014
- **Python 3.6** released in December 2016
- **Python 3.7** released in June 2018

Often times it is quite confusing for newcomers that there are **two major versions 2.x and 3.x available**, still being developed and in parallel use since 2008. This will likely persist for a while since both versions are quite popular and used extensively in the scientific and software development community. One point to note is that they are not entirely code compatible between the versions. We can develop programs and write code in either version but there will be syntactical and other differences. This handbook is based on the 3.x version, but we believe most of the code examples should work with version 2.x as well with some minor tweaks.

1.5 Python 3 versus Python 2

The first version of the Python 3.x was released at the end of 2008. It made changes that made some of the old Python 2.x code incompatible. In this section, we will discuss the difference between the two versions. However, before moving further one might wonder why Python 3 and not Python 2. The most compelling reason for porting to Python 3 is, Python 2.x will not be developed after 2020. So it's no longer a good idea to start new projects in Python 2.x. There won't ever be a Python 2.8. Also, Python 2.7 will only get security updates from the Python 3 development branch. That being said, most of the code we write will work on either version with some small caveats.

A non-exhaustive list of features only available in 3.x releases are shown below.

- strings are Unicode by default
- clean Unicode/bytes separation
- exception chaining
- function annotations
- the syntax for keyword-only arguments
- extended tuple unpacking
- non-local variable declarations

We now discuss some of the significant changes between the two versions.

- **Unicode and Strings**
  - There are two types of strings which can be broadly classified as *byte sequences* and *Unicode strings*.
  - Byte sequences have to be literal characters from the ASCII alphabet. Unicode strings can hold onto pretty much any character we put in there. In Unicode, we can include various languages and, with the right encoding, emoji as well.
  - In Python 2, we have to mark every single Unicode string with a `u` at the beginning, like `u'Hello Python!'`. As we use Unicode string every now and then, it becomes cumbersome to type a `u` for every Unicode string. If we forget to prefix the `u`, we would have a byte sequence instead.

  - With the introduction to Python 3, we need not write a `u` every time. All strings are now Unicode by default and we have to mark byte sequences with a `b`. As Unicode is a much more common scenario, Python 3 has reduced development time for everyone this way.

- **Division with Integers**

  - One of the core values of Python is to never do anything implicitly. For example, never turn a number into string unless a programmer codes for it. Unfortunately, Python 2 took this a bit too far. Consider the following operation

    ```
    5 / 2
    ```

  - The answer we expect here is `2.5`, but instead Python 2 will return only `2`. Following the core value mentioned above, Python will return the output of the same type as the input type. Here, the input is integer and Python returned the output as the integer.
  - Again, this has been fixed in Python 3. It will now output `2.5` as the output to the above problem. In fact, it gives a float output to every division operation.

- **Print Function**

  - The most significant change brought in Python 3 is with regard to the `print` keyword.
  - In Python 2, if we are to output any text, we use `print` followed by output text, which would internally work as a function to render output.
  - In Python 3, `print` has parentheses and hence, if we are to output any text, we use `print()` with the output text inside parentheses.

- **Input Function**

  - There has been an important change to the `input()` function.
  - In Python 2, we have `raw_input()` and `input()` functions for capturing user input from a terminal and standard input devices. `raw_input()` would capture input and treat everything as a string. Whereas, `input()` function helps a user in a way that

if an integer is inputted such as `123`, it would be treated as an integer without being converted to a string. If a string is inputted for `input()`, Python 2 will throw an error.
  - In Python 3, `raw_input()` is gone and `input()` no longer evaluates the data it receives. We always get back a string whatever the input may be.

- **Error Handling**
  - There is a small change in the way each version handles errors.
  - In Python 3, we need to use `as` keyword in `except` clause while checking error, whereas `as` keyword is not required in Python 2. Consider the following example

```
# Python 2
try:
     trying_to_check_error
except NameError, err: # 'as' keyword is NOT needed
     print (err, 'Error Occurred!')

# Python 3
try:
     trying_to_check_error
except NameError as err: # 'as' keyword is needed
     print (err, 'Error Occurred!')
```

- **`__future__` module**
  - `__future__` module is introduced in Python 3 to allow backward compatibility, i.e. to use the features of Python 3 in code developed in Python 2.
  - For example, if we are to use the division feature with float output or `print()` in Python 2, we can do so by using this module.

```
# Python 2 Code
from __future__ import division
from __future__ import print_function

print 5/2 # Output will be 2.5

print('Hello Python using Future module!')
```

Although we have discussed most of the key differences between two versions, there are many other changes being introduced or changed in Python 3 such as `next()` for generators and iterators, how `xrange()` became `range()`, etc.

1.6 Key Takeaways

1. Python is a high level and cross-platform language developed by Guido van Rossum in 1991.
2. It is used in many fields ranging from simple web development to scientific applications.
3. It is characterized by features such as ease of learning and extensibility to other languages.
4. In addition to built-in or standard libraries known as Python Standard Libraries, Python also offers support for third-party libraries.
5. It supports multiple programming styles like Object Oriented, Procedural and Functional.
6. There are two major versions of Python: 2.x and 3.x. The code developed in either version are to some extent compatible with each other.
7. The latest version in the Python 2.x franchise is Python 2.7. There won't be any new update in Python 2.x after 2020.

Chapter 2

Getting Started with Python

As we have seen in the previous section, Python offers different versions, comes in a variety of distributions and is suited for a myriad combinations of platform and devices. Thanks to its versatility, we can use Python to code nearly any task that we can think of logically. One of the most important tasks that a computer performs is mathematical computation. Python provides a direct interface to this fundamental functionality of modern computers. In fact an introduction of Python could be started by showing how it can be used as a tool for simple mathematical calculations.

2.1 Python as a Calculator

The easiest way to perform mathematical calculations using Python is to use the *Console*, which can be used as a fancy calculator. To begin with the simplest mathematical operations, such as addition, subtraction, multiplication and division, we can start using the Python *Console* using the following expressions.

```
# Addition
In []: 5 + 3
Out[]: 8

# Subtraction
In []: 5 - 3
Out[]: 2
```

```
# Multiplication
In []: 5 * 3
Out[]: 15

# Division
In []: 5 / 3
Out[]: 1.6666666666666667

# Modulo
In []: 5 % 2
Out[]: 1
```

NOTE: The content after the # symbols are comments and can be ignored when typing the examples. We will examine comments in more detail in the later sections. Here, `In` refers to an input provided to the Python interpreter and `Out` represents the output returned by the interpreter. Here we use the IPython console to perform the above mathematical operations. They can also be performed in the Python IDLE (Integrated Development and Learning Environment) (aka The Shell), the Python console, or Jupyter notebook in a similar fashion. Basically, we have a host of interfaces to choose from and programmers choose what they find most comfortable. We will stick to the Python Console interface to write and run our Python code in this handbook. To be clear, each of the above-mentioned interfaces connects us to the Python interpreter (which does the computational heavy lifting behind the scenes).

Let us dissect the above examples. A simple Python expression is similar to a mathematical expression. It consists of some numbers, connected by a mathematical *operator*. In programming terminology, numbers (this includes integers as well as numbers with fractional parts ex. 5, 3.89) are called *numeric literals*. An operator (e.g. `+`, `-`, `/`) indicates mathematical operation between its *operands*, and hence the *value* of the expression. The process of deriving the value of an expression is called the *evaluation* of the expression. When a mathematical expression is entered, the Python interpreter automatically evaluates and displays its value in the next line.

Similar to the / division operator, we also have the // integer division operator. The key difference is that the former outputs the decimal value known as a *float* which can be seen in the above example and the latter outputs an *integer* value i.e. without any fractional parts. We will discuss about the `float` and `integer` datatype in more detail in the upcoming sections. Below is an example of an integer division where Python returns the output value without any decimals.

```
In []: 5 // 3
Out[]: 1
```

We can also use expressions as operands in longer *composite expressions*. For example,

```
# Composite expression
In []: 5 + 3 - 3 + 4
Out[]: 9
```

In the example above, the order of evaluation is from *left* to *right*, resulting in the expression `5 + 3` evaluating first. Its value `8` is then combined with the next operand `3` by the `-` operator, evaluating to the value `5` of the composite expression `5 + 3 - 3`. This value is in turn combined with the last literal `4` by the `+` operator, ending up with the value `9` for the whole expression.

In the example, operators are applied from left to right, because `-` and `+` have the same priority. For an expression where we have more than one operators, it is not necessary all the operators have the same priority. Consider the following example,

```
In []: 5 + 3 * 3 - 4
Out[]: 10
```

Here, the expression above evaluated to `10`, because the `*` operator has a higher priority compared to `-` and `+` operators. The expression `3 * 3` is evaluated first resulting in the value of `9` which will be combined with the operand `5` by the operator `+` producing the value of `14`. This value is in turn combined with the next operand `4` by the operator `-` which results in the final value of `10`. The order in which operators are applied is called

operator precedence. In Python, mathematical operators follow the natural precedence observed in mathematics.

Similar to mathematical functions, Python allows the use of brackets `(` and `)` to manually specify the order of evaluation, like the one illustrated below:

```
# Brackets
In []: (5 + 3) * (3 - 4)
Out[]: -8
```

The expression above evaluated to the value `-8` as we explicitly defined the precedence for the expression `5 + 3` to be evaluated first resulting in the value `8`, followed by `3 - 4` producing the value `-1` and then finally we combined both the values `8` and `-1` with operator `*` resulting in the final value to be `-8`.

In the examples above, an operator connects two operands, and hence they are called *binary operators*. In contrast, operators can also be *unary* which take only one operand. Such an operator is `-` known as negation.

```
# Negation
In []: - (5 + 3)
Out[]: -8
```

First, We compute the expression `5 + 3` resulting in the value `8` and secondly, we negate it with `-` operator producing the final value of `-8`.

2.1.1 Floating Point Expressions

Whatever examples we have seen so far were performed on integers also to yield integers. Notice that for the expression `5 / 3` we would get a real number as an output even though the operands are integers. In computer science, real numbers are typically called *floating point numbers*. For example:

```
# Floating Point Addition
In []: 5.0 + 3
Out[]: 8.0
```

```
# Floating Point Multiplication
In []: 5.0 * 3
Out[]: 15.0

# Floating Point Exponential
In []: 5.0 ** 3
Out[]: 125.0

# Floating Point Exponential
In []: 36 ** 0.5
Out[]: 6.0
```

For the above example, the last part calculates the positive square root of `36`.

Python provides a very convenient option to check the type of number, be it an output of an expression or the operand itself.

```
In []: type(5)
Out[]: int

In []: type(5.0)
Out[]: float

In []: type(5.0 ** 3)
Out[]: float
```

The command `type(x)` returns the type of `x`. This command is a *function* in Python where `type()` is a built-in function in Python. We can call a function with a specific *argument* in order to obtain a return value. In the above example, calling a function *type* with the argument `5` returns the value `int` which means `5` is an Integer. The function calls can also be considered as an expression similar to mathematical expressions, with a function name followed by a comma-separated list of arguments enclosed within parentheses. The value of a function call expression is the return value of the function. Following the same example discussed above, *type* is the function name, which takes a single argument and returns the type of argument. As a result, the call to function `type(5.0)` returns the value as a `float`.

We can also convert the type of an argument using the following built-in functions. For example,

```
In []: float(5)
Out[]: 5.0

In []: type(float(5))
Out[]: float

In []: int(5.9)
Out[]: 5

In []: type(int(5.9))
Out[]: int
```

As can be seen in the above example, using the `float` function call, which takes a single argument, we can convert an integer input to a float value. Also, we cross verify it by using the *type* function. Likewise, we have an `int` function using which we can change a float input to the integer value. During the conversion process *int* function just ignores the fractional part of the input value. In the last example, the return value of `int(5.9)` is `5`, even though `5.9` is numerically closer to the integer 6. For floating point conversion by rounding up an integer, we can use the `round` function.

```
In []: round(5.9)
Out[]: 6

In []: round(5.2)
Out[]: 5
```

A call to the `round` function will return a value numerically closer to the argument. It can also round up to a specific number of digits after the decimal point in the argument. We then need to specify two arguments to the function call, with the second argument specifying the number of digits to keep after the decimal point. The following examples illustrate the same. A *comma* is used to separate the arguments in the function call.

```
In []: round(5.98765, 2)
Out[]: 5.99
```

```
In []: round(5.98765, 1)
Out[]: 6.0
```

Another useful function is `abs`, which takes one numerical argument and returns its absolute value.

```
In []: abs(-5)
Out[]: 5

In []: abs(5)
Out[]: 5

In []: abs(5.0)
Out[]: 5.0
```

We can also express a floating point expression using *scientific notation* in the following manner.

```
In []: 5e1
Out[]: 50.0

In []: 5e-1
Out[]: 0.5

In []: 5E2
Out[]: 500.0
```

2.2 Python Basics

We have just completed a brief overview of some functionalities in Python. Let's now get ourselves acquainted with the basics of Python.

2.2.1 Literal Constants

We have seen literals and their use in the above examples. Here we define what they are. Some more concrete examples of numeric literals are `5`, `2.85`, or string literals are `I am a string` or `Welcome to EPAT!`.

It is called a *literal* because we use its value literally. The number `5` always represents itself and nothing else -- it is a *constant* because its value cannot be changed. Similarly, value `2.85` represents itself. Hence, all these are said to be a literal constant.

2.2.2 Numbers

We have already covered numbers in detail in the above section. Here we will discuss it in brief. Numbers can be broadly classified into two types - integer and float.

Examples of an integer number (*int* for short) are - `5`, `3`, `2`, etc. It is just a whole number.

Examples of a floating point number (*floats* for short) are - `2.98745`, `5.5`, `5e-1`, etc. Here, `e` refers to the power of 10. We can write either `e` or `E`, both work just fine.

> **NOTE:** As compared to other programming languages, we do not have separate `long` or `double`. In Python, `int` can be of any length.

2.2.3 Strings

Simply put, a string is a sequence of characters. We use strings almost everywhere in Python code. Python supports both ASCII and Unicode strings. Let us explore strings in more detail.

Single Quote - We can specify a string using single quotes such as `'Python is an easy programming language!'`. All spaces and tabs within the quotes are preserved as-is.

Double Quotes - We can also specify string using double quotes such as `"Yes! Indeed, Python is easy."`. Double quotes work the same way single quotes works. Either can be used.

Triple Quotes - This is used as a delimiter to mark the start and end of a comment. We explain it in greater detail in the next topic.

Strings are immutable - This means once we have created a string we cannot change it. Consider the following example.

```
In []: month_name = 'Fanuary'

# We will be presented with an error at this line.
In []: month_name[0] = 'J'
Traceback (most recent call last):

File "<ipython-input-24>", line 1, in <module>
    month_name[0] = 'J'

TypeError: 'str' object does not support item assignment
```

Here, `month_name` is a variable that is used to hold the value `Fanuary`. Variables can be thought of as a container with a name that is used to hold the value. We will discuss variables in detail in the upcoming sections.

In the above example, we initialize variable `month_name` with an incorrect month name `Fanuary`. Later, we try to correct it by replacing the letter `F` with `J`, where we have been presented with the `TypeError` telling us that strings do not support change operation.

2.2.4 Comments

We have already seen comments before. Comments are used to annotate codes, and they are not interpreted by Python. Comments in Python start with the hash character `#` and end at the end of the physical line in the code. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. This type of comment is also known as a single-line comment.

The other way we can annotate code is by using a multi-line comment that serves as a reference or documentation for others to understand the code.

Let us have a look at the single-line comment in the following example.

```
# Following line adds two integer numbers
In []: 5 + 3
Out[]: 8
```

We can write a comment after code line to annotate what particular line does as depicted in the following example.

```
In []: 5 + 3 # Add two literals
Out[]: 8

In []: # This is also a comment!
```

Python does not support multi-line/block comments. However, we can span comments across multiple lines using the same notation that we use for single line comments. Block comments are intended to the same level as that of code. Each line of a block comment starts with a `#` and a single space.

```
# This is an example of a multi-line comment
# in Python that spans several lines and
# describes the code. It can be used to
# annotate anything like the author name, revisions,
# the purpose of a script, etc. It uses a delimiter
# to mark its start and end.
```

> It is always a good programming practice to generously intersperse our code with comments. Remember: The code tells you how, the comment tells you why.

2.2.5 `print()` function

The `print()` function is a very versatile tool that is used to print anything in Python.

> **NOTE:** In Python 2, `print` is just a statement without parenthesis and NOT a function, whereas in Python 3, `print()` is a function and the content that we need to be outputted goes inside a parenthesis. We have covered this in detail in Chapter 1 under the topic 'Python 2 versus Python 3'.

Let us visit a few examples to understand how the `print()` works.

```
# Simple print function
In []: print("Hello World!")
Out[]: Hello World!

# Concatenating string and integer literal
In []: print("January", 2010)
Out[]: January 2010

# Concatenating two strings
In []: print("AAPL", "is the stock ticker of Apple Inc.")
Out[]: AAPL is the stock ticker of Apple Inc.

# Concatenating a variable and string.
# Here, stock_name is a variable containing stock name.
In []: print(stock_name + " is the stock name of
             Microsoft Corporation.")
Out[]: MSFT is the stock name of Microsoft Corporation.
```

As we can see in the above examples, `print()` can be used in a variety of ways. We can use it to print a simple statement, concatenate with a literal, concatenate two strings, or combine a string with a variable. A common way to use the `print()` function is *f-strings* or *formatted string literal*.

```
# f-strings
In []: print(f'The stock ticker for Apple Inc
             is {stock_name}.')
Out[]: The stock ticker for Apple Inc is AAPL.
```

The above string is called formatted string literal. Such strings are preceded by the letter `f` indicating that it be formatted when we use variable names between curly brackets `{}`. `stock_name` here is a variable name containing the symbol for a stock.

One more way to print a string is by using *%-formatting* style.

```
# %-formatting strings
In []: print("%s is currently trading at %.2f."
```

```
                    %(stock_name, price))
Out[]: AAPL is currently trading at 226.41.
```

Here we print the current trading price of `AAPL` stock. A stock name is stored in the variable `stock_name`, and its price is stored in the variable `price`. `%s` is used for specifying a string literal and `%f` is used to specify float literal. We use `%.2f` to limit two digits after the decimal point.

2.2.6 `format()` function

Another useful function for printing and constructing string for output is `format()` function. Using this function, we can construct a string from information like variables. Consider the following code snippet.

```
In []: stock_ticker = 'AAPL'
In []: price = 226.41
In []: print('We are interested in {x} which is currently
            trading at {y}'.format(x=stock_ticker, y=price))
```

Upon running the above code, we will be presented with the following output.

```
# Output
Out[]: We are interested in AAPL which is currently
       trading at 226.41
```

Above code will first prepare a string internally by substituting the `x` and `y` placeholders with variables `stock_ticker` and `price` respectively, and then prints the final output as a single string. Instead of using placeholders, we can also construct a string in the following manner:

```
In []: print('We are interested in {0} which is currently
             trading at {1}'.format(stock_ticker, price))
```

Here, the output will be similar to the above illustration. A string can be constructed using certain specifications, and the `format` function can be called to substitute those specifications with corresponding arguments of the `format` function. In the above example, `{0}` will be substituted by variable `stock_ticker` and similarly, `{1}` will get a value of `price`. Numbers provided inside the specification are optional, and hence we can also write the same statement as follows

```
print('We are interested in {} which is currently trading
      at {}'.format(stock_ticker, price))
```

which will provide the same exact output as shown above.

2.2.7 Escape Sequence

Escape characters are generally used to perform certain tasks and their usage in code directs the compiler to take a suitable action mapped to that character.

Suppose we want to write a string `That's really easy.`. If we are to write this within a double quote `"`, we could write it as `"That's really easy."`, but what if we are to write the same string within single quote like `'That's really easy.'`, we cannot write it because we have three `'` and the Python interpreter will get confused as to where the string starts and ends. Hence, we need to specify that the string does not end at `s` in the string, instead, it is a part of the string. We can achieve this by using the *escape sequence*. We can specify it by using a `\` (backslash character). For example,

```
In []: 'That\'s really easy.'
Out[]: "That's really easy."
```

Here, we preceded `'` with a `\` character to indicate that it is a part of the string. Similarly, we need to use an escape sequence for double quotes if we are to write a string within double quotes. Also, we can include the backslash character in a string using `\\`. We can break a single line into multiple lines using the `\n` escape sequence.

```
In []: print('That is really easy.\nYes, it really is.')
Out[]: That is really easy.
       Yes, it really is.
```

Another useful escape character is `\t` tab escape sequence. It is used to leave tab spaces between strings.

```
In []: print('AAPL.\tNIFTY50.\tDJIA.\tNIKKEI225.')
Out[]: AAPL.    NIFTY50.        DJIA.    NIKKEI225.
```

In a string, if we are to mention a single \ at the end of the line, it indicates that the string is continued in the next line and no new line is added. Consider below example:

```
In []: print('AAPL is the ticker for Apple Inc. \
  ...:       It is a technology company.')
Out[]: AAPL is the ticker for Apple Inc. It is a
       technology company.
```

Likewise, there are many more escape sequences which can be found on the official Python documentation[1].

2.2.8 Indentation

Whitespaces are important in Python. Whitespace at the start of a line is called *indentation*. It is used to mark the start of a new code block. A block or code block is a group of statements in a program or a script. Leading spaces at the beginning of a line are used to determine the indentation level, which in turn is used to determine the grouping of statements. Also, statements which go together *must* have same indentation level.
A wrong indentation raises the error. For example,

```
stock_name = 'AAPL'

# Incorrect indentation. Note a whitespace at
# the beginning of line.
  print('Stock name is', stock_name)

# Correct indentation.
print('Stock name is', stock_name)
```

Upon running the following code, we will be presented with the following error

```
File "indentation_error.py", line 2
    print('Stock name is', stock_name)
    ^
IndentationError: unexpected indent
```

[1] https://docs.python.org/3.6/reference/lexical_analysis.html#literals

The error indicates to us that the syntax of the program is invalid. That is, the program is not properly written. We *cannot* indent new blocks of statements arbitrarily. Indentation is used widely for defining new block while defining functions, control flow statement, etc. which we will be discussing in detail in the upcoming chapters.

> NOTE: We either use four spaces or tab space for indentation. Most of the modern editors do this automatically for us.

2.3 Key Takeaways

1. Python provides a direct interface known as a Python Console to interact and execute the code directly. The advanced version of the Python console is the IPython (Interactive Python) console.
2. A whole number is called an integer and a fractional number is called a float. They are represented by `int` and `float` data types respectively.
3. Expressions are evaluated from left to right in Python. An expression with a float value as input will return the float output.
4. Strings are immutable in Python. They are written using a single quote or double quote and are represented by `str` data type. Any characters enclosed within quotes is considered a string.
5. Comments are used to annotate code. In Python, `#` character marks the beginning of a single line comment. Python discards anything written after the `#`.
6. Multiline comments are within triple single or double quotes. They start and end with either """ or ’’’.
7. Use the `type()` function to determine the type of data, `print()` to print on the standard output device and `format()` to format the output.
8. Use the escape character to escape certain characters within a string. They can be used to mark tab within a line, a new line within a string and so on.
9. Blocks of code are separated using indentation in Python. Code statements which go together within a single block must have the same indentation level. Otherwise, Python will generate an `IndentationError`.

Chapter 3

Variables and Data Types in Python

We have previously seen that a variable can take data in various formats such as a string, an integer, a number with fractional parts (float), etc. It is now time to look at each of these concepts in greater detail. We start by defining a variable.

3.1 Variables

A variable can be thought of as a container having a *name* which is used to store a *value*. In programming parlance, it is a reserved memory location to store values. In other words, a variable in a Python program gives necessary data to a computer for processing.

In this section, we will learn about variables and their types. Let start by creating a variable.

3.1.1 Variable Declaration and Assignment

In Python, variables need NOT be declared or defined in advance, as is the case in many other programming languages. In fact, Python has no command for declaring a variable. To create a variable, we assign a value to it and start using it. An assignment is performed using a single equal

sign = a.k.a. Assignment operator. A variable is created the moment we assign the first value to it.

```
# Creating a variable
In []: price = 226
```

The above statement can be interpreted as a variable `price` is assigned a value `226`. It is also known as *initializing* the variable. Once this statement is executed, we can start using the `price` in other statements or expressions, and its value will be substituted. For example,

```
In []: print(price)
Out[]: 226 # Output
```

Later, if we change the value of `price` and run the `print` statement again, the new value will appear as output. This is known as *re-declaration* of the variable.

```
In []: price = 230 # Assigning new value
In []: print(price) # Printing price
Out[]: 230 # Output
```

We can also chain assignment operation to variables in Python, which makes it possible to assign the same value to multiple variables simultaneously.

```
In []: x = y = z = 200 # Chaining assignment operation
In []: print(x, y, z) # Printing all variables
Out[]: 200 200 200 # Output
```

The chained assignment shown in the above example assigns the value `200` to variables `x`, `y`, and `z` simultaneously.

3.1.2 Variable Naming Conventions

We use variables everywhere in Python. A variable can have a short name or more descriptive name. The following list of rules should be followed for naming a variable.

- A variable name must start with a letter or the underscore character.

```
stock = 'AAPL' # Valid name
_name = 'AAPL' # Valid name
```

- A variable name cannot start with a number.

```
1stock = 'AAPL' # Invalid name
1_stock = 'AAPL' # Invalid name
```

- A variable name can only contain alpha-numeric characters(A-Z, a-z, 0-9) and underscores(_).

```
# Valid name. It starts with a capital letter.
Stock = 'AAPL'

# Valid name. It is a combination of alphabets
# and the underscore.
stock_price = 226.41

# Valid name. It is a combination of alphabets
# and a number.
stock_1 = 'AAPL'

# Valid name. It is a combination of a capital
# letter, alphabets and a number.
Stock_name_2 = 'MSFT'
```

- A variable name cannot contain whitespace and signs such as +, -, etc.

```
# Invalid name. It cannot contain the whitespace.
stock name = 'AAPL'

# Invalid name. It cannot contain characters
# other than the underscore(_).
stock-name = 'AAPL'
```

- Variable names are case-sensitive.

```
# STOCK, stock and Stock all three are different
# variable names.
```

```
STOCK = 'AAPL'
stock = 'MSFT'
Stock = 'GOOG'
```

Remember that *variable names are case-sensitive* in Python.

- Python keywords cannot be used as a variable name.

```
# 'str', 'is', and 'for' CANNOT be used as the
# variable name as they are reserved keywords
# in Python. Below given names are invalid.
str = 'AAPL'
is = 'A Variable'
for = 'Dummy Variable'
```

The following points are *de facto* practices followed by professional programmers.

- Use a name that describes the purpose, instead of using dummy names. In other words, it should be meaningful.

```
# Valid name but the variable does not
# describe the purpose.
a = 18

# Valid name which describes it suitably
age = 18
```

- Use an underscore character _ to separate two words.

```
# Valid name.
stockname = 'AAPL'

# Valid name. And it also provides concise
# readability.
stock_name = 'AAPL'
```

- Start a variable name with a small alphabet letter.

```
# Valid name.
Stock_name = 'AAPL'

# Valid name. Additionally, it refers to uniformity
# with other statements.
stock_name = 'AAPL'
```

NOTE: Adhering to these rules increases readability of code. Remember these are good coding practices (and recommended but by no means necessary to follow) which you can carry with you to any programming language, not just Python.

3.2 Data Types

Having understood what variables are and how they are used to store values, its time to learn data types of values that variables hold. We will learn about primitive data types such as numeric, string and boolean that are built into Python. Python has four basic data types:

- Integer
- Float
- String
- Boolean

Though we have already had a brief overview of integer, float and string in the previous section, we will cover these data types in greater detail in this section.

3.2.1 Integer

An integer can be thought of as a numeric value without any decimal. In fact, it is used to describe any *whole number* in Python such as `7`, `256`, `1024`, etc. We use an integer value to represent a numeric data from negative infinity to infinity. Such numeric numbers are assigned to variables using an assignment operator.

```
In []: total_output_of_dice_roll = 6
In []: days_elapsed = 30
```

```
In []: total_months = 12
In []: year = 2019
```

We assign a whole number `6` to a variable `total_output_of_dice_roll` as there can be no fractional output for a dice roll. Similarly, we have a variable `days_elapsed` with value `30`, `total_months` having a value `12`, and `year` as `2019`.

3.2.2 Float

A float stands for *floating point number* which essentially means a number with fractional parts. It can also be used for rational numbers, usually ending with a decimal such as `6.5`, `100.1`, `123.45`, etc. Below are some examples where a float value is more appropriate rather than an integer.

```
In []: stock_price = 224.61
In []: height = 6.2
In []: weight = 60.4
```

> **NOTE:** From the statistics perspective, a float value can be thought of as a continuous value, whereas an integer value can correspondingly be a discrete value.

By doing so, we get a fairly good idea how data types and variable names go hand in hand. This, in turn, can be used in expressions to perform any mathematical calculation.

Let's revisit the topic *Python as a Calculator* very briefly but this time using variables.

```
# Assign an integer value
In []: x = 2

# Assign a float value
In []: y = 10.0

# Addition
In []: print(x + y)
Out[]: 12.0
```

```
# Subtraction
In []: print(x - y)
Out[]: -8.0

# Multiplication
In []: print(x * y)
Out[]: 20.0

# Division
In []: print(x / y)
Out[]: 0.2

# Modulo
In []: print(x % y)
Out[]: 2.0

# Exponential / Power
In []: print(x ** y)
Out[]: 1024.0
```

NOTE: Please note the precise use of *comments* used in the code snippet to describe the functionality. Also, note that *output* of all expressions to be *float* number as one of the literals used in the input is a float value.

Look at the above-mentioned examples and try to understand the code snippet. If you are able to get a sense of what's happening, that's great. You are well on track on this Pythonic journey. Nevertheless, let's try to understand the code just to get more clarity. Here, we assign an integer value of `2` to `x` and a float value of `10.0` to `y`. Then, we try to attempt various mathematical operations on these defined variables instead of using direct values. The obvious benefit is the flexibility that we get by using these variables. For example, think of a situation where we want to perform the said operation on different values such as `3` and `15.0`, we just need to re-declare variables `x` and `y` with new values respectively, and the rest of the code remains as it is.

3.2.3 Boolean

This built-in data type can have one of two values, `True` or `False`. We use an assignment operator = to assign a boolean value to variables in a manner similar to what we have seen for integer and float values. For example:

```
In []: buy = True

In []: print(buy)
Out[]: True

In []: sell = False

In []: print(sell)
Out[]: False
```

As we will see in upcoming sections, expressions in Python are often evaluated in the boolean context, meaning they are interpreted to represent their truth value. Boolean expressions are extensively used in logical conditions and control flow statements. Consider the following examples

```
# Checking for equality between 1 and itself using
# comparison operator '=='.
In []: 1 == 1
Out[]: True

# Checking for equality between values 1 and -1
In []: 1 == -1
Out[]: False

# Comparing value 1 with -1
In []: 1 > -1
Out[]: True
```

The above examples are some of the simplest boolean expressions that evaluate to either `True` or `False`.

> NOTE: We do NOT write `True` and `False` within quotes. It needs to be written without quotes. Also, the first letter needs to be

upper case followed by lower case letters. The following list will not be evaluated to a boolean value - 'TRUE' - TRUE - true - 'FALSE' - FALSE - false

3.2.4 String

A string is a collection of alphabets, numbers, and other characters written within a single quote ' or double quotes ". In other words, it is a sequence of characters within quotes. Let us understand how a string works with the help of some examples.

```
# Variable assignment with a string
In []: sample_string = '1% can also be expressed as 0.01'

# Print the variable sample_string
In []: sample_string
Out[]: '1% can also be expressed as 0.01'
```

In the above examples we have defined a string variable with name `sample_string` assigned a value `'1% can also be expressed as 0.01'`. It is interesting to note here that we have used a combination of alphabets, numbers and special characters for defining the variable. In Python, anything that goes within quotes is a string. Consider the following example,

```
In []: stock_price = '224.61'

In []: stock_price
Out[]: '224.61'
```

We define the variable `stock_price` assigning the string value `'224.61'`. The interesting thing to notice is the output of the variable is also a string. Python will not convert the data types implicitly whenever numeric values are given as a string.
We can concatenate two or more string using the + operator.

```
In []: 'Price of AAPL is ' + stock_price
Out[]: 'Price of AAPL is 224.61'
```

Concatenation operation using the + operator works only on a string. It does not work with different data types. If we try to perform the operation, we will be presented with an error.

```
# Re-declaring the variable with an integer value
In []: stock_price = 224.61

In []: 'Price of AAPL is ' + stock_price # Error line
Traceback (most recent call last):

File "<ipython-input-28>", line 1, in <module>
    'Price of AAPL is ' + stock_price

TypeError: must be str, not float
```

As expected, Python spat out a `TypeError` when we tried concatenating a string and float literal. Similar to the `+` operator, we can use the `*` operator with a string literal to produce the same string multiple times.

```
In []: string = 'Python! '

In []: string * 3
Out[]: 'Python! Python! Python! '
```

We can select a substring or part of a string using the slice operation. Slicing is performed using the square brackets `[]`. The syntax for slicing a single element from the string is `[index]` which will return an element at `index`. The index refers to the position of each element in a string and it begins with 0, which keeps on increasing in chronological order for every next element.

```
In []: string = 'EPAT Handbook!'

In []: string[0] # 0 refers to an element E
Out[]: 'E'

In []: string[1] # 1 refers to an element P
Out[]: 'P'
```

In the above example, an element `E` being the first character belongs to an index `0`, `P` being next to `E` belongs to an index `1`, and so on. Similarly, the index for element `b` will be `9`. Can you guess the index for element `k` in the above example?

To slice a substring from a string, the syntax used is `[start index:end index]` which will return the substring starting from an element at `start index` up to but not including an element at `end index`. Consider the following example, where we substring the string from an index `0` up to `4` which yields the output `'EPAT'`. Notice how the element `' '` at an index `4` is not included in the output. Similarly, we slice a substring as seen in the below example.

```
In []: string[0:4]
Out[]: 'EPAT'

In []: string[4]
Out[]: ' '

In []: string[5:13]
Out[]: 'Handbook'

In []: string[13]
Out[]: '!'
```

In Python, we cannot perform the slicing operation with an index not present in the string. Python will throw `IndexError` whenever it encounters slicing operation with incorrect index.

```
In []: string[141]
Traceback (most recent call last):

File "<ipython-input-36>", line 1, in <module>
    string[14]

IndexError: string index out of range
```

In the above example, the last index is `13`. The slicing operation performed with an index `14` will result in an error `IndexError` stating that index we are looking for is not present.

> **NOTE:** We list out some of the important points for string literals below: - In Python 3.x all strings are *Unicode* by default. - A string can be written within either `''` or `""`. Both work fine. -

Strings are immutable. (although you can modify the variable)
- An escape sequence is used within a string to mark a new line, provide tab space, writing \ character, etc.

3.2.5 Operations on String

Here we discuss some of the most common string methods. A method is like a function, but it runs *on* an object. If the variable `sample_string` is a string, then the code `sample_string.upper()` runs the `upper()` method on that string object and returns the result(this idea of running a method on an object is one of the basic ideas that make up Object Oriented Programming, OOP). Some methods also take an argument as a parameter. We provide a parameter to a method as an argument within parentheses.

- `upper()` method: This method returns the upper case version of the string.

```
In []: sample_string.upper()
Out[]: 'EPAT HANDBOOK!'
```

- `lower()` method: This method returns the lower case version of the string.

```
In []: sample_string.lower()
Out[]: 'epat handbook!'
```

- `strip()` method: This method returns a string with whitespace removed from the start and end.

```
In []: '  A string with whitespace at both \
       the ends.  '.strip()
Out[]: 'A string with whitespace at both the ends.'
```

- `isalpha()` method: This method returns the boolean value `True` if all characters in a string are letters, `False` otherwise.

```
In []: 'Alphabets'.isalpha()
Out[]: True
```

```
# The string under evaluation contains whitespace.
In []: 'This string contains only alphabets'.isalpha()
Out[]: False
```

- `isdigit()` method: This method returns the boolean value `True` if all characters in a string are digits, `False` otherwise.

```
In []: '12345'.isdigit()
Out[]: True
```

- `startswith(argument)` method: This method returns the boolean value `True` if the first character of a string starts with the character provided as an argument, `False` otherwise.

```
In []: 'EPAT Handbook!'.startswith('E')
Out[]: True
```

- `endswith(argument)` method: This method returns the boolean value `True` if the last character of a string ends with the character provided as an argument, `False` otherwise.

```
In []: 'EPAT Handbook!'.startswith('k')
Out[]: False # String ends with  the '!' character.
```

- `find(sub, start, end)` method: This method returns the lowest index in a string where substring `sub` is found within the slice `[start:end]`. Here, arguments `start` and `end` are optional. It returns `-1` if `sub` is not found.

```
In []: 'EPAT Handbook!'.find('EPAT')
Out[]: 0

In []: 'EPAT Handbook!'.find('A')
Out[]: 2 # First occurrence of 'A' is at index 2.

In []: 'EPAT Handbook!'.find('Z')
Out[]: -1 # We do not have 'Z' in the string.
```

- `replace(old, new)` method: This method returns a copy of the string with all occurrences of `old` replace by `new`.

```
Out[]: '00 01 10 11'.replace('0', '1')
Out[]: '11 11 11 11' # Replace 0 with 1

In []: '00 01 10 11'.replace('1', '0')
Out[]: '00 00 00 00' # Replace 1 with 0
```

- `split(delim)` method: This method is used to split a string into multiple strings based on the `delim` argument.

```
In []: 'AAPL MSFT GOOG'.split(' ')
Out[]: ['AAPL', 'MSFT', 'GOOG']
```

Here, the Python outputs three strings in a single data structure called *List*. We will learn `list` in more detail in the upcoming section.

- `index(character)` method: This method returns the index of the first occurrence of the `character`.

```
In []: 'EPAT Handbook!'.index('P')
Out[]: 1
```

Python will provide an error if the character provided as an argument is not found within the string.

```
In []: 'EPAT Handbook!'.index('Z')
Traceback (most recent call last):

File "<ipython-input-52>", line 1, in <module>
    'EPAT Handbook!'.index('Z')

ValueError: substring not found
```

- `capitalize()` method: This method returns a capitalized version of the string.

```
In []: 'python is amazing!'.capitalize()
Out[]: 'Python is amazing!'
```

- `count(character)` method: This method returns a count of an argument provided by `character`.

```
In []: 'EPAT Handbook'.count('o')
Out[]: 2

In []: 'EPAT Handbook'.count('a')
Out[]: 1
```

3.2.6 `type()` function

The inbuilt `type(argument)` function is used to evaluate the data type and returns the class type of the `argument` passed as a parameter. This function is mainly used for debugging.

```
# A string is represented by the class 'str'.
In []: type('EPAT Handbook')
Out[]: str

# A float literal is represented by the class 'float'.
In []: type(224.61)
Out[]: float

# An integer literal is represented by the class 'int'.
In []: type(224)
Out[]: int

# An argument provided is within quotation marks.
In []: type('0')
Out[]: str

# A boolean value is represented by the class 'bool'.
In []: type(True)
Out[]: bool

In []: type(False)
Out[]: bool

# An argument is provided within a quotation mark.
In []: type('False')
Out[]: str
```

```
# An object passed as an argument belongs to the
# class 'list'.
In []: type([1, 2, 3])
Out[]: list

# An object passed as an argument belongs to the
# class 'dict'.
In []: type({'key':'value'})
Out[]: dict

# An object passed as an argument belongs to the
# class 'tuple'.
In []: type((1, 2, 3))
Out[]: tuple

# An object passed as an argument belongs to the
# class 'set'.
In []: type({1, 2, 3})
Out[]: set
```

A `list`, `dict`, `tuple`, `set` are native data structures within Python. We will learn these data structures in the upcoming section.

3.3 Type Conversion

We often encounter situations where it becomes necessary to change the data type of the underlying data. Or maybe we find out that we have been using an integer when what we really need is a float. In such cases, we can convert the data types of variables. We can check the data type of a variable using `type()` function as seen above.

There can be two types of conversion possible: *implicit* termed as coercion, and *explicit* often referred to as casting. When we change the type of a variable from one to another, this is called *typecasting*.

Implicit Conversion: This is an automatic type conversion and the Python interpreter handles this on the fly for us. We need not to specify any command or function for same. Take a look at the following example:

```
In []: 8 / 2
Out[]: 4.0
```

The division operation performed between two integers `8` being a dividend and `2` being a divisor. Mathematically, we expect the output to be `4` - an integer value, but instead, Python returned the output as `4.0` - a float value. That is, Python internally converted an integer `4` to float `4.0`.

Explicit Conversion : This type of conversion is user-defined. We need to explicitly change the data type for certain literals to make it compatible for data operations. Let us try to concatenate a string and an integer using the `+` operator.

```
In []: 'This is the year ' + 2019
Traceback (most recent call last):

File "<ipython-input-68>", line 1, in <module>
    'This is the year ' + 2019

TypeError: must be str, not int
```

Here we attempted to join a string `'This is the year '` and an integer `2019`. Doing so, Python threw an error `TypeError` stating incompatible data types. One way to perform the concatenation between the two is to convert the data type of `2019` to string explicitly and then perform the operation. We use `str()` to convert an integer to string.

```
In []: 'This is the year ' + str(2019)
Out[]: 'This is the year 2019'
```

Similarly, we can explicitly change the data type of literals in the following manner.

```
# Integer to float conversion
In []: float(4)
Out[]: 4.0

# String to float conversion
In []: float('4.2')
```

```
Out[]: 4.2

In []: float('4.0')
Out[]: 4.0

# Float to integer conversion
In []: int(4.0)
Out[]: 4 # Python will drop the fractional part.

In []: int(4.2)
Out[]: 4

# String to integer conversion
In []: int('4')
Out[]: 4

# Python does not convert a string literal with a
# fractional part, and instead, it will throw an error.
In []: int('4.0')
Traceback (most recent call last):

File "<ipython-input-75>", line 1, in <module>
    int('4.0')

ValueError: invalid literal for int() with base 10: '4.0'

# Float to string conversion
In []: str(4.2)
Out[]: '4.2'

# Integer to string conversion
In []: str(4)
Out[]: '4'
```

In the above example, we have seen how we can change the data type of literals from one to another. Similarly, the boolean data type represented by `bool` is no different. We can typecast `bool` to `int` as we do for the rest. In fact, Python internally treats the boolean value `False` as `0` and `True` as `1`.

```
# Boolean to integer conversion
In []: int(False)
Out[]: 0

In []: int(True)
Out[]: 1
```

It is also possible to convert an integer value to boolean value. Python converts `0` to `False` and rest all integers gets converted to `True`.

```
# Integer to boolean conversion
In []: bool(0)
Out[]: False

In []: bool(1)
Out[]: True

In []: bool(-1)
Out[]: True

In []: bool(125)
Out[]: True
```

In this section, we started with familiarizing ourselves with variables, and then went on to define them, understanding data types, their internal workings, and type conversions.

3.4 Key Takeaways

1. A variable is used to store a value that can be used repetitively based on the requirement within a program.
2. Python is a loosely typed language. It is not required to specify the type of a variable while declaring it. Python determines the type of the variable based on the value assigned to it.
3. Assignment operator = is used for assigning a value to a variable.
4. Variable names should start with either a letter or an underscore character. They can contain alpha-numeric characters only. It is a good programming practice to have descriptive variable names.

5. Variable names are case sensitive and cannot start with a number.
6. There are four primitive data types in Python:
   (a) Integer represented by `int`
   (b) Float represented by `float`
   (c) String represented by `str`
   (d) Boolean (`True` or `False`) represented by `bool`
7. Internally, `True` is treated as 1 and `False` is treated as 0 in Python.
8. A substring or a part of a string is selected using the square brackets `[]` also known as slice operation.
9. Type conversion happens either implicitly or explicitly.
   (a) Implicit type conversion happens when an operation with compatible data types is executed. For example, `4/2` (integer division) will return `2.0` (float output).
   (b) When an operation involves incompatible data types, they need to be converted to compatible or similar data type. For example: To print a string and an integer together, the integer value needs to be converted to a string before printing.

Chapter 4

Modules, Packages and Libraries

To start with, a module allows us to organize Python code in a systematic manner. It can be considered as a file consisting of Python code. A module can define functions, classes and variables. It can also include runnable code. Consider writing code directly on the Python or IPython console. The definitions that we create(functions and variables) will be lost if we quit the console and enter it again. Therefore, in order to write a longer program, we might consider switching to a text editor to prepare an input for the interpreter and running it with that file as an input instead. This is known as writing a *script*. As a program gets longer, we may want it to split it into several small files for easier maintenance. Also, we may want to use a handy function that we have written in several programs without copying its definition into each program.

To support this, Python has a way to put a code definition in a file and use them in another script or directly in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or in the program that we code.

As we discussed above, a module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. For instance, we create a file called `arithmetic.py` in the current directory with the following contents:

```
# -*- coding: utf-8 -*-
"""
Created on Fri Sep 21 09:29:05 2018
@filename: arithmetic.py
@author: Jay Parmar
"""

def addition(a, b):
    """Returns the sum of of two numbers"""
    return a + b

def multiply(a, b):
    """Returns the product of two numbers"""
    return a * b

def division(dividend, divisor):
    """
    Performs the division operation between the dividend
    and divisor
    """
    return dividend / divisor


def factorial(n):
    """Returns the factorial of n"""
    i = 0
    result = 1
    while(i != n):
        i = i + 1
        result = result * i
    return result
```

We are now ready to import this file in other scripts or directly into the Python interpreter. We can do so with the following command:

```
In []: import arithmetic
```

Once we have imported the module, we can start using its definition in the script without re-writing the same code in the script. We can access func-

tions within the imported module using its name. Consider an example below:

```
In []: result = arithmetic.addition(2, 3)

In []: print(result)
Out[]: 5

In []: arithmetic.multiply(3, 5)
Out[]: 15

In []: arithmetic.division(10, 4)
Out[]: 2.5

In []: arithmetic.factorial(5)
Out[]: 120
```

A module name is available as a string within a script or the interpreter as the value of the global variable `__name__`.

```
In []: arithmetic.__name__
Out[]: 'arithmetic'
```

When we import the module named `arithmetic`, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `arithmetic.py` in a list of directories given by the variables `sys.path`.

This variable is initialized from the following locations:

- The directory containing the input script (or the current directory).
- PYTHONPATH (An environment variable)
- The installation-dependent path.

Here, the module named `arithmetic` has been created that can be imported into the other modules as well. Apart from this, Python has a large set of built-in modules known as the *Python Standard Library*, which we will discuss next.

4.1 Standard Modules

Python comes with a library of standard modules also referred to as the Python Standard Library[1]. Some modules are built into the interpreter; these modules provide access to operations that are not part of the core of the language but are either for efficiency or to provide access to tasks pertaining to the operating system. The set of such modules available also depends on the underlying platform. For example, winreg[2] module is available only on the Windows platform.

Python's standard library is very extensive and offers a wide range of facilities. The library contains built-in modules that provide access to system functionality such as file I/O operations as well as modules that provide standardized solutions for many problems that occur in everyday programming.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems, Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

One particular module that deserves attention is `sys`, which is built into every Python interpreter. This module provides access to variables used or maintained by the interpreter and to functions that interact with the interpreter. It is always available and used as follows:

```
In []: import sys

# Returns a string containing the copyright pertaining to
# the Python interpreter
In []: sys.copyright
Out[]: 'Copyright (c) 2001-2018 Python Software Foundation.
       \nAll Rights Reserved.\n\nCopyright (c) 2000
       BeOpen.com.\nAll Rights Reserved.\n\n
       Copyright (c) 1995-2001 Corporation for National
```

[1] https://docs.python.org/3/library/
[2] https://docs.python.org/3/library/winreg.html#module-winreg

```
        Research Initiatives.\nAll Rights Reserved.\n\n
        Copyright (c) 1991-1995 Stichting Mathematisch
        Centrum, Amsterdam.\nAll Rights Reserved.'

# Return the name of the encoding used to convert between
# unicode filenames and bytes filenames.
In []: sys.getfilesystemencoding()
Out[]: 'utf-8'

# Returns information regarding the Python interpreter
In []: sys.implementation
Out[]: namespace(cache_tag='cpython-36',
       hexversion=50726384, name='cpython',
       version=sys.version_info(major=3, minor=6,
       micro=5, releaselevel='final', serial=0))

# Returns a string containing a platform identifier
In []: sys.platform
Out[]: 'win32'

# Returns a string containing the version number of the
# Python interpreter plus additional information on the
# compiler
In []: sys.version
Out[]: '3.6.5 |Anaconda, Inc.| (default, Mar 29 2018,
       13:32:41) [MSC v.1900 64 bit (AMD64)]'
```

In the above examples, we discussed a handful of functionalities provided by the `sys` module. As can be seen, we can use it to access system level functionality through Python code. In addition to this, there are various other built-in modules in Python. We list some of them below based on their functionality.

- Text Processing : `string`, `readline`, `re`, `unicodedata`, etc.
- Data Types : `datetime`, `calendar`, `array`, `copy`, `pprint`, `enum`, etc.
- Mathematical : `numbers`, `math`, `random`, `decimal`, `statistics`, etc.
- Files and Directories : `pathlib`, `stat`, `glob`, `shutil`, `filinput`, etc.
- Data Persistence: `pickle`, `dbm`, `sqlite3`, etc.
- Compression and Archiving: `gzip`, `bz2`, `zipfile`, `tarfile`, etc.

- Concurrent Execution: `threading`, `multiprocessing`, `sched`, `queue`, etc.
- Networking: `socket`, `ssl`, `asyncio`, `signal`, etc.
- Internet Data Handling: `email`, `json`, `mailbox`, `mimetypes`, `binascii`, etc.
- Internet Protocols: `urllib`, `http`, `ftplib`, `smtplib`, `telnetlib`, `xmlrpc`, etc.

In addition to the standard library, there is a growing collection of several thousand modules ranging from individual modules to packages and entire application development frameworks, available from the *Python Package Index*.

4.2 Packages

Packages can be considered as a collection of modules. It is a way of structuring Python's module namespace by using "dotted module names". For example, the module name `matplotlib.pyplot` designates a submodule named `pyplot` in a package named `matplotlib`. Packaging modules in such a way saves the author of different modules from having to worry about each other's global variable names and the use of dotted module names saves the author of multi-module packages from having to worry about each other's module names.

Suppose we want to design a package (a collection of modules) for the uniform handling of various trading strategies and their data. There are many different data files based on data frequencies, so we may need to create and maintain a growing collection of modules for the conversion between the various data frequencies. Also, there are many different strategies and operations that we might need to perform. All of this put together means we would have to write a never-ending stream of modules to handle the combinatorics of data, strategies, and operations. Here's a possible package structure to make our lives easier.

```
strats/                          Top-level package
    __init__.py                  Initialize strats package
    data/                        Sub-package for data
        __init__.py
```

```
        equity.py                       Equity module
        currency.py
        options.py
        ...
    strategies/                         Sub-package for strategies
        __init__.py
        rsi.py                          RSI module
        macd.py
        smalma.py
        peratio.py
        fundamentalindex.py
       statisticalarbitrage.py
        turtle.py
        ...
    operations/                         Sub-package for operations
        __init__.py
        performanceanalytics.py
        dataconversion.py
        ...
```

When importing the package, Python searches through the directories in `sys.path` looking for the package subdirectory. The `__init__.py` file is required to make Python treat the directories as containing packages. If we are to use this package, we can do so in the following manner:

```
import strats.data.equity
import strats.strategies.statisticalarbitrage
```

Above statements loads the `equity` and `statisticalarbitrage` modules from the `data` and `strategies` sub-packages respectively under the `strats` package.

4.3 Installation of External Libraries

One of the great things about using Python is the number of fantastic code libraries (apart from the Python Standard Library) which are readily available for a wide variety of domains that can save much coding or make a particular task much easier to accomplish. Before we can use such

external libraries, we need to install them.

The goal here is to install software that can automatically download and install Python modules/libraries for us. Two commonly used installation managers are conda[3] and pip[4]. We choose to go with `pip` for our installations.

> `pip` comes pre-installed for Python >= 2.7 or Python >= 3.4 downloaded from Python official site[5]. If the Anaconda distribution has been installed, both `pip` and `conda` are available to manage package installations.

4.3.1 Installing `pip`

We can install a `pip` via the command line by using the *curl command*, which downloads the pip installation *perl* script.

```
curl -O https://bootstrap.pypa.io/get-pip.py
```

Once it is downloaded, we need to execute it in the command prompt with the Python interpreter.

```
python get-pip.py
```

If the above command fails on a Mac and Linux distribution due to permission issues (most likely because Python does not have permission to update certain directories on the file system. These directories are read-only by default to ensure that random scripts cannot mess with important files and infect the system with viruses), we may need to run following command.

```
sudo python get-pip.py
```

4.3.2 Installing Libraries

Now that we have installed `pip`, it is easy to install python modules since it does all the work for us. When we find a module that we want to use,

[3] https://conda.io/docs/
[4] https://pip.pypa.io/en/stable/installing/
[5] https://www.python.org

usually the documentation or installation instructions will include the necessary pip command.

The Python Package Index[6] is the main repository for third-party Python packages. The advantage of a library being available on PyPI is the ease of installation using `pip install <package_name>` such as

```
pip install pandas
pip install numpy
pip install nsepy
```

Remember, again if the above command fails on a Mac and Linux distribution due to permission issue, we can run the following command:

```
sudo pip install pandas
sudo pip install nsepy
```

The above examples will install the latest version of the libraries. To install a specific version, we execute the following command:

```
pip install SomeLibrary==1.1
```

To install greater than or equal to one version and less than another:

```
pip install SomeLibrary>=1, < 2
```

Listed below are some of the most popular libraries used in different domains:

- Data Science : `NumPy`, `pandas`, `SciPy`, etc
- Graphics : `matplotlib`, `plotly`, `seaborn`, `bokeh`, etc.
- Statistics : `statsmodels`
- Machine learning : `SciKit-Learn`, `Keras`, `TensorFlow`, `Theano`, etc.
- Web scraping : `Scrapy`, `BeautifulSoup`,
- GUI Toolkit : `pyGtk`, `pyQT`, `wxPython`, etc.
- Web Development : `Django`, `web2py`, `Flask`, `Pyramid`, etc.

We can upgrade already installed libraries to the latest version from PyPI using the following command:

[6] https://pypi.org/

```
pip install --upgrade SomeLibrary
```

> Remember, `pip` commands are run directly within the command prompt or shell without the python interpreter. If we are to run `pip` commands from Jupyter Notebook we need to prefix it with the `!` letter like `!pip install SomeLibrary`.

4.4 Importing modules

Now that we have installed the module that we are interested in, we can start using it right away. First, we need to import the installed module in our code. We do so with the *import* statement. The import statement is the most common way of invoking the module machinery.

4.4.1 `import` statement

We can import any module, either internal or external into our code using the import statement. Take a look at below example:

```
# Importing an internal module
In []: import math

# Importing an external library
In []: import pandas
```

The above example will import all definitions within the imported library. We can use these definitions using `.` (dot operator). For example,

```
# Accessing the 'pi' attribute of the 'math' module
In []: math.pi
Out[]: 3.141592653589793

# Accessing the 'floor' function from the 'math' module
In []: math.floor
Out[]: <function math.floor>

# Accessing the 'floor' method from the 'math'
In []: math.floor(10.8)
Out[]: 10
```

```
# Accessing the 'DataFrame' module from the 'pandas'
# library
In []: pandas.DataFrame
Out[]: pandas.core.frame.DataFrame
```

As seen in the above example, we can access attributes and methods of the imported library using the dot operator along with the library name. In fact, the library we import acts as an object and hence, we can call its attributes using the dot notation. We can also alias a library name while importing it with the help of the `as` keyword.

```
# Aliasing math as 'm'
In []: import math as m

# Aliasing pandas as 'pd'
In []: import pandas as pd

# Aliasing numpy as 'np'
In []: import numpy as np
```

An alias can be used in the same way as the module name.

```
In []: m.pi
Out[]: 3.141592653589793

In []: m.e
Out[]: 2.718281828459045

In []: m.gamma
Out[]: <function math.gamma>
```

4.4.2 Selective imports

The other way to import a definition/module is to import all *definitions* in that particular module or all modules in the particular package. We can do so by using `from` keyword.

```
# Import all definitions of math module
In []: from math import *
```

```
# Import all definitions from pyplot module of matplotlib
# library
In []: from matplotlib.pyplot import *

# Accessing definitions directly without module name
In []: pi
Out[]: 3.141592653589793

In []: e
Out[]: 2.718281828459045

In []: floor(10.8)
Out[]: 10
```

Remember, when definitions are directly imported from any module, we need not use the module name along with the dot operator to access them. Instead, we can directly use the definition name itself. Similarly, it is also possible to import the specific definition that we intend to use, instead of importing all definitions. Consider a scenario where we need to floor down a value, we can import the `floor()` definition from the `math` module, rather than importing every other unnecessary definition. Such import is called *selective import*.

```
# Selective import
# Import only floor from math module
In []: from math import floor

In []: floor(10.8)
Out[]: 10

# Error line as the ceil is not imported from math module
In []: ceil(10.2)
Traceback (most recent call last):

File "<ipython-input-33>", line 1, in <module>
    ceil(10.2)

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

```
# Error line as the math is not imported
# Only the floor from math is imported
In []: math.ceil(10.2)
Traceback (most recent call last):

File "<ipython-input-34>", line 1, in <module>
    math.ceil(10.2)

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

In the above example, we selectively import the `floor` from the `math` module. If we try to access any other definition from the `math` module, Python will return an error stating *definition* not defined as the interpreter won't be able to find any such definition in the code.

4.4.3 The Module Search Path

Let's say we have a module called `vwap_module.py` which is inside the folder `strategy`. We also have a script called `backtesting.py` in a directory called `backtest`.

We want to be able to import the code in `vwap_module.py` to use in `backtesting.py`. We do so by writing `import vwap_module` in `backtesting.py`. The content might look like this:

```
# Content of strategy/vwap_module.py
def run_strategy():
    print('Running strategy logic')

# Content of backtest/backtesting.py
import vwap_module

vwap_module.run_strategy()
```

The moment the Python interpreter encounters line `import vwap_module`, it will generate the following error:

```
Traceback (most recent call last):
```

```
  File "backtest/backtesting.py", line 1, in <module>
    import vwap_module

ModuleNotFoundError: No module named 'vwap_module'
```

When Python hits the line `import vwap_module`, it tries to find a package or a module called `vwap_module`. A module is a file with a matching extension, such as `.py`. Here, Python is looking for a file `vwap_module.py` in the same directory where `backtesting.py` exists, and not finding it.

Python has a simple algorithm for finding a module with a given name, such as `vwap_module`. It looks for a file called `vwap_module.py` in the directories listed in the variable `sys.path`.

```
In []: import sys

In []: type(sys.path)
Out[]: list

In []: for path in sys.path:
  ...:     print(path)
  ...:

C:\Users\...\Continuum\anaconda3\python36.zip
C:\Users\...\Continuum\anaconda3\DLLs
C:\Users\...\Continuum\anaconda3\lib
C:\Users\...\Continuum\anaconda3
C:\Users\...\Continuum\anaconda3\lib\site-packages
C:\Users\...\Continuum\anaconda3\lib\site-packages\win32
C:\Users\...\Continuum\anaconda3\lib\site-packages\win32\lib
C:\Users\...\Continuum\anaconda3\lib\site-packages\Pythonwin
C:\Users\...\.ipython
```

In the above code snippet, we print paths present in the `sys.path`. The `vwap_strategy.py` file is in the `strategy` directory, and this directory is not in the `sys.path` list.

Because `sys.path` is just a Python list, we can make the import statement work by appending the `strategy` directory to the list.

```
In []: import sys
In []: sys.path.append('strategy')

# Now the import statement will work
In []: import vwap_strategy
```

There are various ways of making sure a directory is always on the `sys.path` list when you run Python. Some of them are

- Keep the directory into the contents of the `PYTHONPATH` environment variable.
- Make the module part of an installable package, and install it.

As a crude hack, we can keep the module in the same directory as the code file.

4.5 `dir()` function

We can use the built-in function `dir()` to find which names a module defines. It returns a sorted list of strings.

```
In []: import arithmetic

In []: dir(arithmetic)
Out[]:
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'addition',
 'division',
 'factorial',
 'multiply']
```

Here, we can see a sorted list of names within the module `arithmetic`. All other names that begin with an underscore are default Python attributes associated with the module (we did not define them.)

Without arguments, `dir()` lists the names we have defined currently:

```
In []: a = 1

In []: b = 'string'

In []: import arithmetic

In []: dir()
Out[]:
['__builtins__',
 'a',
 'arithmetic',
 'b',
 'exit',
 'quit']
```

Note that it lists all types of names: variables, modules, functions, etc. The `dir()` does not list the names of built-in functions and variables. They are defined in the standard module `builtins`. We can list them by passing `builtins` as an argument in the `dir()`.

```
In []: import builtins

In []: dir(builtins)
Out[]: ['ArithmeticError', 'AssertionError',
        'AttributeError', 'BaseException',
        'BlockingIOError', 'BrokenPipeError',
        'BufferError', 'BytesWarning', 'ChildProcessError',
        'ConnectionAbortedError', 'ConnectionError',
        'ConnectionRefusedError', 'ConnectionResetError',
        'DeprecationWarning', 'EOFError', 'Ellipsis',
        'EnvironmentError', 'Exception', 'False',
        'SyntaxError', ... ]
```

4.6 Key Takeaways

1. A module is a Python file which can be referenced and used in other Python code.
2. A single module can also have multiple Python files grouped together.
3. A collection of modules are known as packages or libraries. The words library and package are used interchangeably.
4. Python comes with a large set of built-in libraries known as the Python Standard Library.
5. Modules in Python Standard Library provides access to core system functionality and solutions for many problems that occur in everyday programming.
6. The `sys` library is present in every Python installation irrespective of the distribution and underlying architecture and it acts as an intermediary between the system and Python.
7. In addition to built-in libraries, additional third-party/external libraries can be installed using either the `pip` or `conda` package managers.
8. The pip command comes pre-installed for Python version >= 2.7 or Python >=3.4
9. A library (either built-in or external) needs to be imported into Python code before it can be used. It can be achieved using `import library_name` keyword.
10. It is a good idea to alias the library name that we import using an `as` keyword.
11. It is always a good programming practice to selectively import only those modules which are required, instead of importing the whole library.
12. Python will look out for the library being imported in the module search path. If the library is not available in any of the paths listed by module search path, Python will throw an error.
13. The `dir()` function is used to list all attributes and methods of an object. If a library name is passed as an argument to the `dir()`, it returns sub-modules and functions of the library.

Chapter 5

Data Structures

In this section we will learn about various built-in data structures such as tuples, lists, dictionaries, and sets. Like a variable, data structures are also used to store a value. Unlike a variable, they don't just store a value, rather a collection of values in various formats. Broadly data structures are divided into *array*, *list* and *file*. Arrays can be considered a basic form of data structure while files are more advanced to store complex data.

5.1 Indexing and Slicing

Before we dive into the world of data structures, let us have a look at the concept of *indexing* and *slicing* which is applicable to all data structures in Python. A string can be thought of as a sequence of characters. Similatly, data structures store sequences of objects (floats, integers, strings, etc.).

Consider a sequence of 10 characters ranging from A to J where we assign a unique position to each literal in a sequence. The position assigned to each character is a sequence of integers beginning with 0 up to the last character. These increase successively by 1 as can be seen below.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Sequence	A	B	C	D	E	F	G	H	I	J

In the above sequence, the character `A` is at index `0`, `B` at `1`, `C` at `2`, and so on. Notice how the index increases in chronological order by one unit at

each step. Whenever a new character is appended to this sequence, it will be appended at the end, and will be assigned the next index value (in the above example, the new index will be `10` for the new character). Almost all data structures in Python have an index to position and locate the element.

Elements within the sequence can be accessed using the square brackets `[]`. It takes `index` of an element and returns the element itself. The syntax for accessing a single element is as follows:

```
sequence[i]
```

The above statement will return the element from sequence at index `i`. We can access multiple elements from the sequence using the syntax `[start index : end index]` in the following manner:

```
sequence[si : ei]
```

The above statement will return values starting at index `si` up to but NOT including the element at index `ei`. This operation is referred to as *slicing*. For example:

```
sequence[0:4] will return elements from 'A' to 'D' and not up
to 'E'. Element at the last index in the provided range will
not be returned.
```

Python also supports negative indexing to access elements from the sequence end and it starts with -1 as follows:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Sequence	A	B	C	D	E	F	G	H	I	J
Negative Index	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

A sequence can also be sliced using the negative indexing. In order to access the last element, we write

```
sequence[-1]
```

and it will return the element `J`. Similarly, a range can be provided to access multiple elements.

```
sequence[-5:-1] will return elements from 'F' to 'I'
```

5.2 Array

An array can be thought of as a container that can hold a fixed number of data values of the same type. Though the use of array is less popular in Python as compared to other languages such as C and Java, most other data structures internally make use of arrays to implement their algorithms. An array consists of two components, viz *Element* and *Index*.

- Element: These are the actual data values to be stored in an array.
- Index: Each element in array is positioned at the particular location depicted by an index. Python follows *zero based indexing* which means an index will always start with 0.

We can create an array by using the built-in `array` module. It can be created as follows:

```
In []: from array import *

In []: arr = array('i', [2, 4, 6, 8])

In []: arr
Out[]: array('i', [2, 4, 6, 8])

In []: type(arr)
Out[]: array.array
```

In the above example, we import the `array` method from the `array` module and then initialize the variable `arr` with values `2`, `4`, `6`, and `8` within the square brackets. The `i` represents the data type of values. In this case, it represents integer. Python array documentation[1] provides more information about the various type codes available in the Python.

5.2.1 Visualizing an Array

An array declared above can be represented in the following manner:

[1] https://docs.python.org/3.4/library/array.html

Index	**0**	**1**	**2**	**3**
Element	2	4	6	8

From the above illustration, following are the points to be considered.

- Index starts with 0.
- Array length is 4 which means it can store 4 values.
- Array can hold values with single data type only.
- Each element can be accessed via its index.

5.2.2 Accessing Array Element

We use slicing operation to access array elements. Slicing operation is performed using the square brackets `[]`. It takes an index of an element we are interested in. It can be noticed that the index of the first element in the above array is 0. So, in order to access an element at the position 3, we use the notation `arr[2]` to access it.

```
# Here, 2 represents the index of element 6
In []: arr[2]
Out[]: 6

In []: arr[0]
Out[]: 2
```

5.2.3 Manipulating Arrays

The `array` module provides a wide variety of operations that can be performed based on the requirement. We will learn some of the most frequently used operations.

We use insertion operation to insert one or more data elements into an array. Based on the requirement, an element can be inserted at the beginning, end or any given index using the `insert()` method.

```
# Inserting an element at the beginning
In []: arr.insert(0, 20)
```

```
In []: arr
Out[]: array('i', [20, 2, 4, 6, 8])

# Inserting an element at the index 3
In []: arr.insert(3, 60)

In []: arr
Out[]: array('i', [20, 2, 4, 60, 6, 8])
```

An element can be deleted from an array using the built-in `remove()` method.

```
In []: arr.remove(20)

In []: arr
Out[]: array('i', [2, 4, 60, 6, 8])

In []: arr.remove(60)

In []: arr
Out[]: array('i', [2, 4, 6, 8])
```

We can update an element at the specific index using the assignment operator = in the following manner:

```
# Update an element at index 1
In []: arr[0] = 1

In []: arr
Out[]: array('i', [1, 4, 6, 8])

# Update an element at index 3
In []: arr[3] = 7

In []: arr
Out[]: array('i', [1, 4, 6, 7])
```

In addition to the above mentioned operation, the `array` module provides a bunch of other operations that can be carried out on an array such as reverse, pop, append, search, conversion to other types, etc.

Though Python allows us to perform a wide variety of operations on arrays, the built-in array module is rarely used. Instead, in real world programming most programmers prefers to use NumPy arrays provided by the NumPy library.

5.3 Tuples

In Python, tuples are part of the standard library. Like arrays, tuples also hold multiple values within them separated by commas. In addition, it also allows storing values of different types together. Tuples are immutable, and usually, contain a heterogeneous sequence of elements that are accessed via unpacking or indexing.

To create a tuple, we place all elements within brackets `()`. Unlike arrays, we need not import any module for using tuples.

```
# Creating a tuple 'tup' with elements of the same
# data type
In []: tup = (1, 2, 3)

In []: tup
Out[]: (1, 2, 3)

# Verifying the type
In []: type(tup)
Out[]: tuple

# Creating a tuple 'tupl' with elements of different data
# types
In []: tupl = (1, 'a', 2.5)

In []: type(tupl)
Out[]: tuple
```

The tuple `tupl` created above can be visualized in the following manner:

Index	**0**	**1**	**2**
Element	1	'a'	2.5

A tuple can also be created without using the brackets.

```
# Creating a tuple without brackets
In []: tup = 1, 2, 3

In []: type(tup)
Out[]: tuple
```

We can repeat a value multiple times within a tuple as follows:

```
In []: tupl = (1,) * 5 # Note trailing comma

In []: tupl
Out[]: (1, 1, 1, 1, 1)
```

5.3.1 Accessing tuple elements

A slice operation performed using the square brackets `[]` is used to access tuple elements. We pass the index value within the square brackets to get an element of our interest. Like arrays, tuples also have an index and all elements are associated with the particular index number. Again, the index starts with '0'.

```
# Access an element at index 0
In []: tup[0]
Out[]: 1

# Access an element at index 2
In []: tup[2]
Out[]: 1
```

Python throws an error if we try to access an element that does not exist. In other words, if we use the slice operation with a non-existent index, we will get an error.

```
In []: tup[3]
Traceback (most recent call last):

File "<ipython-input-30>", line 1, in <module>
```

```
    tup[3]

IndexError: tuple index out of range
```

In the above example, we try to access an element with index 3 which does not exist. Hence, Python threw an error stating `index out of range`.
The built-in `len()` function is used to check the length of a tuple.

```
In []: len(tup)
Out[]: 3

In []: len(tupl)
Out[]: 5
```

5.3.2 Immutability

In Python, tuple objects are immutable. That is, once they are created, it cannot be modified. If we try to modify a tuple, Python will throw an error.

```
In []: tup[1] = 10
Traceback (most recent call last):

File "<ipython-input-33>", line 1, in <module>
    tup[1] = 10

TypeError: 'tuple' object does not support item assignment
```

As expected, the interpreter threw an error depicting the tuple object to be immutable.

5.3.3 Concatenating Tuples

Python allows us to combine two or more tuples or directly concatenate new values to an existing tuple. The concatenation is performed in the following manner:

```
In []: t1 = (1, 2, 3)

In []: t2 = (4, 5)
```

```
In []: t1 + t2
Out[]: (1, 2, 3, 4, 5)
```

Tuples can be concatenated using operators `*=` and `+=`.

```
In []: t1 = (1, 2, 3)

In []: t1 += 4, 5

In []: t1
Out[]: (1, 2, 3, 4, 5)
```

5.3.4 Unpacking Tuples

In one of the above example, we encountered the statement `tup = 1, 2, 3` which is in turn an example of *tuple packing*. That is we pack various values together into a single variable `tup`. The reverse operation is also possible:

```
In []: tup
Out[]: (1, 2, 3)

In []: x, y, z = tup
```

The above statement performs the *unpacking* operation. It will assign the value `1` to the variable `x`, `2` to `y`, and `3` to `z`. This operation requires that there are as many variables on the left hand side of the equal sign as there are elements in the tuple.

5.3.5 Tuple methods

Tuple being one of the simple objects in Python, it is easier to maintain. There are only two methods available for tuple objects:

- index() : This method returns the index of the element.

```
In []: tup
Out[]: (1, 2, 3)
```

```
# Returns the index of value '3'.
In []: tup.index(3)
Out[]: 2
```

- count() : This method counts the number of occurrences of a value.

```
In []: tup = (1, 1, 1, 1, 1)

In []: tup.count(1)
Out[]: 5
```

Some of the reasons why tuples are useful are given below:

- They are faster than lists.
- They protect the data as they are immutable.
- They can be used as keys on dictionaries.

5.4 Lists

A `list` is a data structure that holds an ordered collection of items i.e. we can store a *sequence* of items in a list. In Python, lists are created by placing all items within square brackets `[]` separated by comma.

It can have any number of items and they may be of different data types and can be created in the following manner:

```
# Empty list
In []: list_a = []

In []: list_a
Out[]: []

# List with integers
In []: list_b = [1, 2, 3]

In []: list_b
Out[]: [1, 2, 3]
```

```
# List with mixed data types
In []: list_c =[1, 2.5, 'hello']

In []: list_c
Out[]: [1, 2.5, 'hello']
```

A list can also have another list as an item. This is called *nested list*.

```
In []: a_list = [1, 2, 3, ['hello', 'stock'], 4.5]
```

5.4.1 Accessing List Items

Like with any other data structure, slice operator is used to access list items or a range of list items. It can be used in the following manner.

```
In []: stock_list = ['HP', 'GOOG', 'TSLA', 'MSFT', 'AAPL',
                     'AMZN', 'NFLX']

# Accessing an element at index 2
In []: stock_list[2]
Out[]: 'TSLA'

# Accessing multiple elements using slicing
In []: stock_list[1:4]
Out[]: ['GOOG', 'TSLA', 'MSFT']

# Accessing last element using negative index
In []: stock_list[-1]
Out[]: 'NFLX'
```

5.4.2 Updating Lists

Unlike tuples, lists are mutable. That is, we can change the content even after it is created. Again, the slicing operation helps us here

```
In []: stock_list
Out[]: ['HP', 'GOOG', 'TSLA', 'MSFT', 'AAPL', 'AMZN',
        'NFLX']
```

```
# Updating the first element
In []: stock_list[0] = 'NVDA'

# Updating the last three elements
In []: stock_list[-3:] = ['AMD', 'GE', 'BAC']

In []: stock_list
Out[]: ['NVDA', 'GOOG', 'TSLA', 'AMD', 'GE', 'BAC']
```

It is also possible to add new elements to an existing list. Essentially a list is an object in Python. Hence, the list class provides various methods to be used upon the list object. There are two methods `append()` and `extend()` which are used to update an existing list.

- `append(element)` method adds a single element to the end of the list. It does not return the new list, just modifies the original list.
- `extend(list2)` method adds the elements in list2 to the end of the list.

```
In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT']

In []: stock_list.append('AMZN')

In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN']
```

In the above example, we add new element using the `append()` method. Let's add multiple elements to the list. In Python, whenever we are to add multiple literal to any object, we enclose it within list i.e. using `[]` the square brackets. The output that we expect is the appended list will all the new elements.

```
In []: stock_list.append(['TSLA', 'GE', 'NFLX'])

In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN', ['TSLA', 'GE',
       'NFLX']]
```

The output we got is not as per our expectation. Python amended the new element as a single element to the `stock_list` instead of appending three different elements. Python provides the `extend()` method to achieve this.

```
In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN']

In []: stock_list.extend(['TSLA', 'GE', 'NFLX'])

In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN', 'TSLA', 'GE',
        'NFLX']
```

To simplify, the `append()` method is used to add a single element to existing list and it takes a single element as an argument, whereas the `extend()` method is used to add multiple elements to existing list and it takes a list as an argument.

5.4.3 List Manipulation

Lists are one of the most versatile and used data structures in Python. In addition to the above discussed methods, we also have other useful methods at our disposal. Some of them are listed below:

- `insert(index, element)` : Inserts an item at a given position. The first argument is the index of the element before which to insert, so `list.insert(0, element)` inserts at the beginning of the list.

  ```
  # Inserting an element at index position 1.
  In []: stock_list.insert(1, 'AAPL')

  In []: stock_list
  Out[]: ['HP', 'AAPL', 'GOOG', 'MSFT', 'AMZN', 'TSLA', 'GE',
          'NFLX']
  ```

- `remove(element)` : Removes the first item whose value is `element` provided in an argument. Python will throw an error if there is no such item.

```
# Removing the element 'AAPL'
In []: stock_list.remove('AAPL')

In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN', 'TSLA', 'GE',
        'NFLX']

# Again removing the element 'AAPL'.
# This line will throw an error as there is no element
# 'AAPL' the list.
In []: stock_list.remove('AAPL')
Traceback (most recent call last):

File "<ipython-input-73>", line 1, in <module>
    stock_list.remove('AAPL')

ValueError: list.remove(x): x not in list
```

- `pop()` : This function removes and returns the last item in the list. If we provide the `index` as an argument, it removes the item at the given position in the list and returns it. It is optional to provide an argument here.

```
# Without providing index position as an argument. Returns
# and removes the last element in the list.
In []: stock_list.pop()
Out[]: 'NFLX'

In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN', 'TSLA', 'GE']

# Providing an index position as an argument. Returns and
# removes the element from the specific location.
In []: stock_list.pop(2)
Out[]: 'MSFT'

In []: stock_list
Out[]: ['HP', 'GOOG', 'AMZN', 'TSLA', 'GE']
```

- `index(element)` : Returns the index of the first item whose value is `element` provided in an argument. Python will throw an error if there is no such item.

```
In []: stock_list.index('GOOG')
Out[]: 1

In []: stock_list.index('GE')
Out[]: 4
```

- `count(element)` : Returns the number of times `element` appears in the list.

```
# Count the element 'GOOG'
In []: stock_list.count('GOOG')
Out[]: 1

# Appending the same list with 'GOOG'
In []: stock_list.append('GOOG')

In []: stock_list
Out[]: ['HP', 'GOOG', 'AMZN', 'TSLA', 'GE', 'GOOG']

# Again, counting the element 'GOOG'
In []: stock_list.count('GOOG')
Out[]: 2
```

- `sort()` : When called, this method returns the sorted list. The sort operation will be in place.

```
# Sorting the list. The same list will be updated.
In []: stock_list.sort()

In []: stock_list
Out[]: ['AMZN', 'GE', 'GOOG', 'GOOG', 'HP', 'TSLA']
```

- `reverse()` : This method reverses the elements of the list and the operation performed will be in place.

```
# Reversing the elements within the list.
In []: stock_list.reverse()

In []: stock_list
Out[]: ['TSLA', 'HP', 'GOOG', 'GOOG', 'GE', 'AMZN']
```

5.4.4 Stacks and Queues

The list methods make it very easy to use a list as a stack or queue. A *stack* is a data structure (though not available directly in Python) where the last element added is the first element retrieved, also known as *Last In, First Out (LIFO)*. A list can be used as a stack using the `append()` and `pop()` method. To add an item to the top of the stack, we use the `append()` and to retrieve an item from the top of the stack, we use the `pop()` without an explicit index. For example:

```
# (Bottom) 1 -> 5 -> 6 (Top)
In []: stack = [1, 5, 6]

In []: stack.append(4)    # 4 is added on top of 6 (Top)

In []: stack.append(5)    # 5 is added on top of 4 (Top)

In []: stack
Out[]: [1, 5, 6, 4, 5]

In []: stack.pop()    # 5 is removed from the top
Out[]: 5

In []: stack.pop()    # 4 is removed from the top
Out[]: 4

In []: stack.pop()    # 6 is removed from the top
Out[]: 6

In []: stack    # Remaining elements in the stack
Out[]: [1, 5]
```

Another data structure that can be built using list methods is *queue*, where

the first element added is the first element retrieved, also known as *First In, First Out (FIFO)*. Consider a queue at a ticket counter where people are catered according to their arrival sequence and hence the first person to arrive is also the first to leave.

In order to implement a queue, we need to use the `collections.deque` module; however, lists are not efficient for this purpose as it involves heavy memory usage to change the position of every element with each insertion and deletion operation.

It can be created using the `append()` and `popleft()` methods. For example,

```
# Import 'deque' module from the 'collections' package
In []: from collections import deque

# Define initial queue
In []: queue = deque(['Perl', 'PHP', 'Go'])

# 'R' arrives and joins the queue
In []: queue.append('R')

# 'Python' arrives and joins the queue
In []: queue.append('Python')

# The first to arrive leaves the queue
In []: queue.popleft()
Out[]: 'Perl'

# The second to arrive leaves the queue
In []: queue.popleft()
Out[]: 'PHP'

# The remaining queue in order of arrival
In []: queue
Out[]: deque(['Go', 'R', 'Python'])
```

5.5 Dictionaries

A Python dictionary is an unordered collection of items. It stores data in *key-value* pairs. A dictionary is like a phone-book where we can find the phone numbers or contact details of a person by knowing only his/her name i.e. we associate names *(keys)* with corresponding details *(values)*. Note that the keys must be unique just like the way it is in a phone book i.e. we cannot have two persons with the exact same name.

In a dictionary, pairs of keys and values are specified within curly brackets `{}` using the following notation:

```
dictionary = {key1 : value1, key2 : value2, key3 : value3}
```

Notice that the key-value pairs are separated by the colon `:` and pairs themselves are separated by `,`. Also, we can use only immutable objects like strings and tuples for the keys of a dictionary. Values of a dictionary can be either mutable or immutable objects. Dictionaries that we create are instances of the `dict` class and they are unordered, so the order that keys are added doesn't necessarily reflect the same order when they are retrieved back.

5.5.1 Creating and accessing dictionaries

A dictionary can be created either using the curly brackets `{}` or the method `dict()`. For example:

```
# Creating an empty dictionary using {}
In []: tickers = {}

In []: type(tickers)
Out[]: dict

# Creating an empty dictionary using the dict() method
In []: tickers = dict()

In []: type(tickers)
Out[]: dict
```

Let us create a dictionary with values of the same data type.

```
In []: tickers = {'GOOG' : 'Alphabet Inc.',
  ...:            'AAPL' : 'Apple Inc.',
  ...:            'MSFT' : 'Microsoft Corporation'}

In []: tickers
Out[]:
{'GOOG': 'Alphabet Inc.',
 'AAPL': 'Apple Inc.',
 'MSFT': 'Microsoft Corporation'}
```

Next, we will create a dictionary with multiple data types.

```
In []: ticker = {'symbol' : 'AAPL',
  ...:           'price' : 224.95,
  ...:           'company' : 'Apple Inc',
  ...:           'founded' : 1976,
  ...:           'products' : ['Machintosh', 'iPod',
                               'iPhone', 'iPad']
                }
```

We can also provide a dictionary as a value to another dictionary key. Such a dictionary is called *nested dictionary*. Take a look at below example:

```
In []: tickers = {'AAPL' : {'name' : 'Apple Inc.',
  ...:                      'price' : 224.95
  ...:                     },
  ...:            'GOOG' : {'name' : 'Alphabet Inc.',
  ...:                      'price' : 1194.64
  ...:                     }
  ...:           }
```

Keys in a dictionary should be unique. If we supply the same key for multiple pairs, Python will ignore the previous value associated with the key and only the recent value will be stored. Consider the following example:

```
In []: same_keys = {'symbol' : 'AAPL',
  ...:              'symbol' : 'GOOG'}

In []: same_keys
Out[]: {'symbol': 'GOOG'}
```

In the above example, Python discarded the value `AAPL` and retained the latest value assigned to the same key. Once we have created dictionaries, we can access them with the help of the respective keys. We use the slice operator `[]` to access the values; however, we supply a `key` to obtain its value. With the dictionaries created above, we can access values in the following manner:

```
In []: ticker
Out[]:
{'symbol': 'AAPL',
 'price': 224.95,
 'company': 'Apple Inc',
 'founded': 1976,
 'products': ['Machintosh', 'iPod', 'iPhone', 'iPad']}

In []: tickers
Out[]:
{'AAPL': {'name': 'Apple Inc.', 'price': 224.95},
 'GOOG': {'name': 'Alphabet Inc.', 'price': 1194.64}}

# Accessing the symbol name
In []: ticker['symbol']
Out[]: 'AAPL'

# Accessing the ticker price
In []: ticker['price']
Out[]: 224.95

# Accessing the product list
In []: ticker['products']
Out[]: ['Machintosh', 'iPod', 'iPhone', 'iPad']

# Accessing the item at position 2 in the product list.
In []: ticker['products'][2]
Out[]: 'iPhone'

# Accessing the first nested dictionary from the
# 'tickers' dictionary
In []: tickers['AAPL']
```

```
Out[]: {'name': 'Apple Inc.', 'price': 224.95}

# Accessing the price of 'GOOG' ticker using chaining
# operation
In []: tickers['GOOG']['price']
Out[]: 1194.64
```

5.5.2 Altering dictionaries

A value in a dictionary can be updated by assigning a new value to its corresponding key using the assignment operator =.

```
In []: ticker['price']
Out[]: 224.95

In []: ticker['price'] = 226

In []: ticker['price']
Out[]: 226
```

A new key-value pair can also be added in a similar fashion. To add a new element, we write the new `key` inside the square brackets `[]` and assign a new value. For example:

```
In []: ticker['founders'] = ['Steve Jobs', 'Steve Wozniak',
                             'Ronald Wayne']

In []: ticker
Out[]:
{'symbol': 'AAPL',
 'price': 226,
 'company': 'Apple Inc',
 'founded': 1976,
 'products': ['Machintosh', 'iPod', 'iPhone', 'iPad'],
 'founders': ['Steve Jobs', 'Steve Wozniak',
              'Ronald Wayne']}
```

In the above example, we add the key `founders` and assign the list `['Steve Jobs', 'Steve Wozniak', 'Ronald Wayne']` as value. If we are to delete

any key-value pair in the dictionary, we use the built-in `del()` function as follows:

```
In []: del(ticker['founders'])

In []: ticker
Out[]:
{'symbol': 'AAPL',
 'price': 226,
 'company': 'Apple Inc',
 'founded': 1976,
 'products': ['Machintosh', 'iPod', 'iPhone', 'iPad']}
```

5.5.3 Dictionary Methods

The `dict` class provides various methods using which we can perform a variety of operations. In addition to these methods, we can use built-in `len()` functions to get the length of a dictionary.

```
In []: len(ticker)
Out[]: 5

In []: len(tickers)
Out[]: 2
```

Now we discuss some of the popular methods provided by the `dict` class.

- `items()` : This method returns a object containing all times in the calling object.

```
In []: ticker.items()
Out[]: dict_items([('symbol', 'AAPL'), ('price', 226),
                   ('company', 'Apple Inc'),
                   ('founded', 1976),
                   ('products', ['Machintosh', 'iPod',
                                 'iPhone', 'iPad']])
```

- `keys()` : This method returns all keys in the calling dictionary.

```
In []: ticker.keys()
Out[]: dict_keys(['symbol', 'price', 'company', 'founded',
                  'products'])
```

- `values()` : This method returns all values in the calling object.

```
In []: ticker.values()
Out[]: dict_values(['AAPL', 224.95, 'Apple Inc', 1976,
                    ['Machintosh', 'iPod', 'iPhone',
                    'iPad']])
```

- `pop()` : This method pops the item whose key is given as an argument.

```
In []: tickers
Out[]:
{'GOOG': 'Alphabet Inc.',
 'AAPL': 'Apple Inc.',
 'MSFT': 'Microsoft Corporation'}

In []: tickers.pop('GOOG')
Out[]: 'Alphabet Inc.'

In []: tickers
Out[]: {'AAPL': 'Apple Inc.',
'MSFT': 'Microsoft Corporation'}
```

- `copy()` : As the name suggests, this method copies the calling dictionary to another dictionary.

```
In []: aapl = ticker.copy()

In []: aapl
Out[]:
{'symbol': 'AAPL',
 'price': 224.95,
 'company': 'Apple Inc',
 'founded': 1976,
 'products': ['Machintosh', 'iPod', 'iPhone', 'iPad']}
```

- `clear()` : This method empties the calling dictionary.

```
In []: ticker.clear()

In []: ticker
Out[]: {}
```

- `update()` : This method allows to add new key-pair value from another dictionary.

```
In []: ticker1 = {'NFLX' : 'Netflix'}

In []: ticker2 = {'AMZN' : 'Amazon'}

In []: new_tickers = {}

In []: new_tickers.update(ticker1)

In []: new_tickers.update(ticker2)

In []: new_tickers
Out[]: {'NFLX': 'Netflix', 'AMZN': 'Amazon'}
```

5.6 Sets

A set is an unordered and unindexed collection of items. It is a collection data type which is mutable, iterable and contains no duplicate values. A set in Python represents the mathematical notion of a set.

In Python sets are written using the curly brackets in the following way:

```
In []: universe ={'GOOG', 'AAPL', 'NFLX', 'GE'}

In []: universe
Out[]: {'AAPL', 'GE', 'GOOG', 'NFLX'}
```

We cannot access items in a set by referring to an index (slicing operation), since sets are unordered the item has no index. But we can loop through all items using the `for` loop which will be discussed in the upcoming section. Once a set set is created, we cannot change its items, but we can add new items using the `add()` method.

```
In []: universe.add('AMZN')

In []: universe
Out[]: {'AAPL', 'AMZN', 'GE', 'GOOG', 'NFLX'}
```

Python won't add the same item again nor will it throw any error.

```
In []: universe.add('AMZN')

In []: universe.add('GOOG')

In []: universe
Out[]: {'AAPL', 'AMZN', 'GE', 'GOOG', 'NFLX'}
```

In order to add multiple items, we use the `update()` method with new items to be added within a list.

```
In []: universe.update(['FB', 'TSLA'])

In []: universe
Out[]: {'AAPL', 'AMZN', 'FB', 'GE', 'GOOG', 'NFLX', 'TSLA'}
```

We can use the inbuilt `len()` function to determine the length of a set.

```
In []: len(universe)
Out[]: 7
```

To remove or delete an item, we can use the `remove()` or `discard()` methods. For example,

```
In []: universe.remove('FB')

In []: universe.discard('TSLA')

In []: universe
Out[]: {'AAPL', 'AMZN', 'GE', 'GOOG', 'NFLX'}
```

If we try to remove an item using the `remove()` which is not present in the set, Python will throw an error.

```
In []: universe.remove('FB')
Traceback (most recent call last):

File "<ipython-input-159>", line 1, in <module>
    universe.remove('FB')

KeyError: 'FB'
```

The `discard()` method will not throw any error if we try to discard an item which is not present in the set.

```
In []: universe
Out[]: {'AAPL', 'AMZN', 'GE', 'GOOG', 'NFLX'}

In []: universe.discard('FB')
```

We use the `clear()` method to empty the set.

```
In []: universe.clear()

In []: universe
Out[]: set()
```

Following the mathematical notation, we can perform set operations such as union, intersection, difference, etc. in Python using the set. Consider the following examples:

- We define two sets `tech_stocks` and `fin_stocks` as follows:

  ```
  In []: tech_stocks = {'AMD', 'GOOG', 'AAPL', 'WDC'}

  In []: fin_stocks = {'BAC', 'BMO', 'JPLS'}
  ```

- `union()` method: This method allows performing a union between sets. This operation returns all elements within both sets.

  ```
  # Performs the 'union` operation
  In []: universe = tech_stocks.union(fin_stocks)

  # 'universe' contains all elements of both sets
  ```

```
In []: universe
Out[]: {'AAPL', 'AMD', 'BAC', 'BMO', 'GOOG', 'JPLS', 'WDC'}
```

- `intersection()` method: This method performs the intersection between sets. It returns only elements which are available in both sets.

```
# Only elements present in the 'universe' set and
# and 'fin_stocks` are returned
In []: universe.intersection(fin_stocks)
Out[]: {'BAC', 'BMO', 'JPLS'}
```

- `difference()` method: This method performs the difference operation and returns a set containing all elements of the calling object but not including elements of the second set.

```
# All elements of the 'universe' set is returned except
# elements of the 'fin_stock'
In []: universe.difference(fin_stocks)
Out[]: {'AAPL', 'AMD', 'GOOG', 'WDC'}
```

- `issubset()` method: This method checks whether all elements of calling set is present within a second set or not. It returns true if the calling set is subset of the second set, false otherwise.

```
# True, as the 'universe' contains all elements of
# the 'fin_stocks'
In []: fin_stocks.issubset(universe)
Out[]: True

# Can you guess why it resulted in to False?
In []: universe.issubset(tech_stocks)
Out[]: False
```

- `isdisjoint()` method: This method checks for the intersection between two sets. It returns true if the calling set is disjoint and not intersected with the second set, false otherwise.

```
# True, none of the set contains any element of each other
In []: fin_stocks.isdisjoint(tech_stocks)
```

```
Out[]: True

# False, the 'universe' set contains elements of
# the 'fin_stocks' set
In []: fin_stocks.isdisjoint(universe)
Out[]: False
```

- `issuperset()` method: This method checks whether the calling set contains all elements of the second set. It returns true, if the calling set contains all elements of the second set, false otherwise.

```
# True, the 'universe' set contains all elements of
# the 'fin_stocks'
In []: universe.issuperset(fin_stocks)
Out[]: True

# True, the 'universe' set contains all elements of
# the 'tech_stocks'
In []: universe.issuperset(tech_stocks)
Out[]: True

# False, the 'fin_stocks' set does not contains all
# elements of the 'universe' set
In []: fin_stocks.issuperset(universe)
Out[]: False
```

5.7 Key Takeaways

1. Data structures are used to store a collection of values.
2. Arrays, tuples, lists, sets and dictionaries are primitive data structures in Python. Except for dictionaries, all data structures are sequential in nature.
3. In Python, indexing starts with 0 and negative indexing starts with -1.
4. Elements within data structures are accessed using the slicing operation `[start_index:end_index]` where `start_index` is inclusive and `end_index` is exclusive.
5. An array can hold a fixed number of data values of the same type. Arrays are not built-in data structures. We can use an `array` library to

perform basic array operations.

6. A tuple can hold multiple values of different types within it separated by commas. Tuples are enclosed within parentheses `()` and they are immutable.
7. A list holds an ordered collection of items. Lists are created by placing all items within square brackets `[]` separated by a comma. They can also be used to implement other data structures like stacks and queues.
8. A list can also have another list as an item. This is called a nested list.
9. Dictionary stores data in the form of a key-value pair. It can be created using the curly brackets `{}`. Element/Pair within the dictionary is accessed using the corresponding keys instead of an index.
10. Sets are an unordered data structure created using the curly brackets `{}`. It cannot contain duplicate elements.

Chapter 6

Keywords & Operators

Python is a high-level language that that allows us to nearly write programs in a natural language like English. However, there are certain words and symbols used internally by Python which carry a definite unambiguous meaning. They can be used only in certain pre-defined ways when we program. We will explore such words known as keywords and various operators in this section.

6.1 Python Keywords

Keywords are reserved (plainspeak for 'set aside' or 'on hold') words in Python. They are built into the core language. They cannot be used as a variable name, class name, function name or any other identifier. These keywords are used to define the syntax and semantics of the Python.

Since Python is case-sensitive, so are the keywords. All keywords are in lowercase except `True`, `False` and `None`. In this section, we will learn various keywords.

- The `and` keyword is a logical operator used to combine conditional statements and it returns `True` if both statements are `True`, `False` otherwise.

```
In []: (5 > 3) and (4 > 2)
Out[]: True
```

- The `as` keyword is used to create an alias. Consider the following example where we create an alias for the `calendar` module as `c` while importing it. Once aliased we can refer to the imported module with its alias.

```
In []: import calendar as c

In []: c.isleap(2019)
Out[]: False
```

- The `assert` keyword is used while debugging code. It allows us to check if a condition in code returns True, if not Python will raise an `AssertionError`. If condition returns True, no output is displayed.

```
In []: stock = 'GOOG'

In []: assert stock == 'GOOG'

In []: assert stock == 'AAPL'
Traceback (most recent call last):

File "<ipython-input-6>", line 1, in <module>
    assert stock == 'AAPL'

AssertionError
```

- The `break` keyword is used to break a `for` loop and `while` loop.

```
# A for loop that will print from 0 to 9
# Loop will break when 'i' will be greater than 3
In []: for i in range(10):
  ...:   if i > 3:
  ...:     break
  ...:   print(i)

# Output
0
1
2
3
```

- The `class` keyword is used to create a class.

```
In []: class stock():
  ...:     name = 'AAPL'
  ...:     price = 224.61
  ...:

In []: s1 = stock()

In []: s1.name
Out[]: 'AAPL'

In []: s1.price
Out[]: 224.61
```

- The `continue` keyword is used to end the current iteration in a `for` loop or `while` loop, and continues to the next iteration.

```
# A for loop to print 0 to 9
# When `i` will be 4, iteration will continue without
# checking further
In []: for i in range(9):
  ...:     if i == 4:
  ...:         continue
  ...:     else:
  ...:         print(i)

# Output: It does not contain 4 as loop continued with
# the next iteration
0
1
2
3
5
6
7
8
```

- The `def` keyword is used to create/define a function within Python.

```
In []: def python_function():
  ...:     print('Hello! This is a Python Function.')

In []: python_function()
Out[]: Hello! This is a Python Function.
```

- The `del` keyword is used to delete objects. It can also be used to delete variables, lists, or elements from data structures, etc.

```
# Define the variable 'a'
In []: a = 'Hello'

# Delete the variable 'a'
In []: del(a)

# Print the variable 'a'. Python will throw an error as the
# variable 'a' does not exist.
In []: a
Traceback (most recent call last):

File "<ipython-input-17>", line 1, in <module>
    a

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

- The `if` keyword is used as a logical test between boolean expression(s). If the boolean expression evaluates to `True`, the code following the 'if' condition will be executed.
- The `elif` keyword is used to define multiple if conditions. It is also referred to as 'else if'. If the expression evaluates to `True`, the code following the condition will be executed.
- The `else` keyword is used to define the code block that will be executed when all `if` conditions above it fail. It does not check for any condition, it just executes the code if all the conditions above it fail.

```
In []: number = 5
  ...:
  ...: if number < 5:
```

```
...:     print('Number is less than 5')
...: elif number == 5:
...:     print('Number is equal to 5')
...: else:
...:     print('Number is greater than 5')
...:

# Output
Number is equal to 5
```

- The `try` keyword is used to define the try...except code block which will be followed by the code block defined by `except` keyword. Python tries to execute the code within `try` block and if it executes successfully, it ignores subsequent blocks.

- The `except` keyword is used in try...except blocks for handling any error raised by the `try` block. It is used to define a code block that will be executed if the `try` block fails to execute and raises any error.

```
# Python will throw an error here because the variable 'y'
# is not defined
In []: try:
...:     y > 5
...: except:
...:     print('Something went wrong.')
...:

# Output
Something went wrong.
```

- The `finally` keyword is used to try...except block to define a block of code that will run no matter if the try block raises an error or not. This can be useful to close objects and clean up resources.

```
# Code in the 'finally' block will execute whether or not
# the code in the 'try' block raises an error
In []: try:
...:     y > 5
...: except:
```

```
...:     print('Something went wrong.')
...: finally:
...:     print('The try...except code is finished')
...:

# Output
Something went wrong.
The try...except code is finished
```

- The `False` keyword is used to represent the boolean result false. It evaluates to 0 when it is cast to an integer value.

```
In []: buy_flag = False

In []: int(False)
Out[]: 0
```

- The `True` keyword is used to represent the boolean result true. It evaluates to 1 when cast to an integer value.

```
In []: buy_flag = True

In []: int(True)
Out[]: 1
```

- The `for` keyword is used to define/create a for loop.

```
In []: for i in range(5):
...:     print(i)
...:
...:

# Output
0
1
2
3
4
```

- The `import` keyword is used to import external libraries and modules in to current program code.

```
In []: import pandas

In []: import numpy
```

- The `from` keyword is used while importing modules and libraries in Python. It is used to import specific modules from the libraries.

```
# Importing DataFrame module from the pandas library
In []: from pandas import DataFrame

# Importing time module from the datetime package
In []: from datetime import time

# Importing floor function from the math module
In []: from math import floor
```

- The `global` keyword is used to declare a global variables to be used from non-global scope.

```
In []: ticker = 'GOOG'
  ...:
  ...: def stocks():
  ...:     global ticker
  ...:     # Redeclaring or assigning new value 'MSFT' to
  ...:     # global variable 'ticker'
  ...:     ticker = 'MSFT'
  ...:
  ...:
  ...: stocks()
  ...:
  ...: print(ticker)

# Output
MSFT
```

- The `in` keyword is used to check if a value is present in a sequence. It is also used to iterate through a sequence in a `for` loop.

```
In []: stock_list = ['GOOG', 'MSFT', 'NFLX', 'TSLA']

In []: 'GOOG' in stock_list
Out[]: True

In []: 'AMZN' in stock_list
Out[]: False
```

- The `is` keyword is used to test if two variables refers to the same object in Python. It returns true if two variables are same objects, false otherwise.

```
In []: stock_list = ['GOOG', 'MSFT', 'NFLX', 'TSLA']

In []: y = stock_list

# Checks whether the 'stock_list' and 'y' are same or not
In []: stock_list is y
Out[]: True

In []: y is stock_list
Out[]: True

# Reversing elements in the 'stock_list' also reverses
# elements in the 'y' as both are same
In []: stock_list.reverse()

In []: y
Out[]: ['TSLA', 'NFLX', 'MSFT', 'GOOG']
```

- The `lambda` keyword is used to create a small anonymous function in Python. It can take multiple arguments but accepts only a single expression.

```
# Creates an anonymous function that adds two values
# provided by 'x' and 'y'
In []: addition = lambda x, y : x + y

In []: addition(5, 2)
Out[]: 7
```

- The `None` keyword is used to define a null value, or no value at all. It is not same as 0, False, or an empty string. None is represented by a datatype of `NoneType`.

```
In []: x = None

In []: type(x)
Out[]: NoneType
```

- The `nonlocal` keyword is used to declare a variable that is not local. It is used to work with variables inside nested functions, where the variable should not belong to the inner function.

```
# Instead of creating a new variable 'x' within the
# 'nested_function' block, it will use the variable 'x'
# defined in the 'main function'
In []: def main_function():
  ...:   x = "MSFT"
  ...:   def nested_function():
  ...:     nonlocal x
  ...:     x = "GOOG"
  ...:   nested_function()
  ...:   return x
  ...:
  ...:
  ...: print(main_function())

# Output
GOOG
```

- The `not` keyword is a logical operator similar to the `and` operator. It returns the boolean value `True` if an expression is not true, `False` otherwise.

```
In []: buy = False

In []: not buy
Out[]: True
```

- The `or` keyword is a logical operator used to check multiple conditional statements and it returns `True` if at least one statement is `True`, `False` otherwise.

```
In []: (5 > 3) or (4 < 2)
Out[]: True
```

- The `pass` keyword is used as a placeholder for a null statement. It does nothing when used. If we have empty function definition, Python will return an error. Hence, it can be used as a placeholder in an empty function.

```
In []: def empty_function():
  ...:
  ...:
  ...:
In []: empty_function()

File "<ipython-input-49>", line 5
    empty_function()
                   ^
IndentationError: expected an indented block

In []: def empty_function():
  ...:     pass
  ...:
  ...:
  ...: empty_function()
```

- The `raise` keyword is used to raise an error explicitly in a code.

```
In []: x = 'Python'

In []: if not type(x) is int:
  ...:     raise TypeError('Only integers are allowed')
  ...:
Traceback (most recent call last):

File "<ipython-input-52>", line 2, in <module>
```

```
        raise TypeError('Only integers are allowed')

TypeError: Only integers are allowed
```

- The `return` keyword is used to return a value from a function or method.

```
In []: def addition(a, b):
  ...:     return a + b
  ...:

In []: addition(2, 3)
Out[]: 5
```

- The `with` keyword is used to wrap the execution of a block with methods defined by a context manager[1]. It simplifies exception handling by encapsulating common preparation and cleanup tasks. For example, the `open()` function is a context manager in itself, which allows opening a file, keeping it open as long as the execution is in context of the `with`, and closing it as soon as we leave the context. So simply put, some resources are acquired by the `with` statement and released when we leave the `with` context.

```
# Open the file 'abc.txt' using the 'with' keyword in the
# append mode
In []: with open('abc.txt', 'a') as file:
  ...:     # Append the file
  ...:     file.write('Hello Python')
  ...:
  ...:
# We do not need to close the file as it will be called
# automatically as soon as we leave the 'with' block

# Open the file in the read mode
In []: with open('abc.txt', 'r') as file:
  ...:     print(file.readline())
  ...:
```

[1] https://docs.python.org/3/reference/datamodel.html#context-managers

```
  ...:
Out[]: Hello Python
```

6.2 Operators

Operators are constructs or special symbols which can manipulate or compute the values of operands in an expression. In other words, they are used to perform operations on variables and values. Python provides a bunch of different operators to perform a variety of operations. They are broadly categorized into the following:

6.2.1 Arithmetic operators

Arithmetic operators are used with numerical values to perform the common mathematical operations.

- `+` : This is an addition operator used to perform the addition between values.

```
In []: 5 + 3
Out[]: 8
```

- `-` : This is a subtraction operator used to perform the subtraction between operands.

```
In []: 5 - 2
Out[]: 3
```

- `*` : This is a multiplication operator used to multiply the operands.

```
In []: 5 * 2
Out[]: 10
```

- `/` : This is a division operator which performs the division operation and returns a float output.

```
In []: 10 / 2
Out[]: 5.0
```

- `%` : This is a modulus operator. It returns the remainder of the division operation.

```
In []: 16 % 5
Out[]: 1
```

- `**` : This operator is used to perform the exponentiation operation, sometimes referred to as the *raised to power* operation. This essentially performs the operation of raising one quantity to the power of another quantity.

```
In []: 2 ** 3
Out[]: 8

In []: 3 ** 2
Out[]: 9
```

- `//` : This operator is used to perform the floor division operation and it returns the integer output.

```
# Floor division operation
In []: 10 // 4
Out[]: 2

# Normal division operation
In []: 10 / 4
Out[]: 2.5
```

6.2.2 Comparison operators

Comparison operators are used to compare two or more values. It works with almost all data types in Python and returns either `True` or `False`. We define the below variables to understand these operators better.

```
In []: a = 5

In []: b = 3

In []: x = 5

In []: y = 8
```

- `==` : This is an *equal to* operator used to check whether two values are equal or not. It returns true if values are equal, false otherwise.

```
In []: a == x
Out[]: True

In []: a == b
Out[]: False
```

- `!=` : This is a *not equal to* operator and works exactly opposite to the above discussed *equal to* operator. It returns True if values are not equal, and false otherwise.

```
In []: a != x
Out[]: False

In []: a != b
Out[]: True
```

- `>` : This is a *greater than* operator used to check whether one value is greater than another value. It returns true if the first value is greater compared to the latter, false otherwise.

```
In []: y > x
Out[]: True

In []: b > y
Out[]: False
```

- `<` : This is a *less than* operator used to check whether one value is less than another value. It returns true if the first value is less compared to the latter, false otherwise.

```
In []: y < x
Out[]: False

In []: b < y
Out[]: True
```

- `>=` : This is a *greater than or equal to* operator used to check whether one value is greater than or equal to another value or not. It returns true if the first value is either greater than or equal to the latter value, false otherwise.

```
In []: a >= x
Out[]: True

In []: y >= a
Out[]: True

In []: b >= x
Out[]: False
```

- `<=` : This is a *less than or equal to* operator used to check whether one value is less than or equal to another value or not. It returns true if the first value is either less than or equal to the latter value, false otherwise.

```
In []: a <= x
Out[]: True

In []: y <= a
Out[]: False

In []: b <= x
Out[]: True
```

6.2.3 Logical operators

Logical operators are used to compare two or more conditional statements or expressions, and returns boolean result.

- `and` : This operator compares multiple conditional statements and returns true if all statements results in true, and false if any statement is false.

```
In []: 5 == 5 and 3 < 5
Out[]: True
```

```
In []: 8 >= 8 and 5 < 5
Out[]: False

In []: 5 > 3 and 8 == 8 and 3 <= 5
Out[]: True
```

- `or` : This operator compares multiple conditional statements and returns true if at least one of the statements is true, and false if all statements are false.

```
In []: 5 == 5 or 3 > 5
Out[]: True

In []: 3 <= 3 or 5 < 3 or 8 < 5
Out[]: True

In []: 3 < 3 or 5 < 3 or 8 < 5
Out[]: False
```

- `not` : This operator reverses the result. It returns true if the result is false, and vice versa.

```
In []: 3 == 3
Out[]: True

In []: not 3 == 3
Out[]: False

In []: 3 != 3
Out[]: False

In []: not 3 != 3
Out[]: True
```

6.2.4 Bitwise operator

Bitwise operators are used to compare and perform logical operations on binary numbers. Essentially operations are performed on each bit of a binary

number instead of a number. Binary numbers are represented by a combination of 0 and 1. For better understanding, we define following numbers (integers) and their corresponding binary numbers.

Number	Binary
201	1100 1001
15	0000 1111

In the above example, both 201 and 15 are represented by 8 bits. Bitwise operators work on multi-bit values, but conceptually one bit at a time. In other words, these operator works on 0 and 1 representation of underlying numbers.

- `&` : This is a bitwise *AND* operator that returns 1 only if *both* of its inputs are 1, 0 otherwise. Below is the truth table for the `&` operator with four bits.

Bits	1 2 3 4
Input 1	0 0 1 1
Input 2	0 1 0 1
& Output	0 0 0 1

We can compute the bitwise `&` operation between 201 and 15 as follows:

```
In []: 201 & 15
Out[]: 9
```

Let us understand with the help of a truth table, how Python returned the value 9.

		Binary Numbers
Input 1	201	1100 1001
Input 2	15	0000 1111
& Output	9	0000 1001

Python evaluated `&` operation based on each bit of inputs and re-

turned an integer equivalent of the binary output. In the above example, decimal equivalent of 0000 1001 is 9.

- `|` : This is a bitwise *OR* operator that returns 1 if *any* of its inputs are 1, 0 otherwise. Below is the truth table for the `|` operator with four bits.

Bits	**1 2 3 4**
Input 1	0 0 1 1
Input 2	0 1 0 1
`|` *Output*	0 1 1 1

The bitwise `|` operation between 201 and 15 can be performed in the following way:

```
In []: 201 | 15
Out[]: 207
```

The above operation can be verified via the truth table as shown below:

		Binary Numbers
Input 1	201	1100 1001
Input 2	15	0000 1111
Output	207	1100 1111

In the above example, Python evaluated `|` operation bitwise and returned the output 1100 1111 which is the binary equivalent of 207.

- `^` : This is a bitwise *XOR* operator that returns 1 only if *any one* of its input is 1, 0 otherwise. Below is the truth table for the XOR operation.

Bits	**1 2 3 4**
Input 1	0 0 1 1
Input 2	0 1 0 1
Output	0 1 1 0

Notice that it does not return 1 if all inputs are 1. The bitwise `^` can be

performed as follows:

```
In []: 201 ^ 15
Out[]: 198
```

The output returned by the Python can be verified via its corresponding truth table as shown below.

		Binary Numbers
Input 1	201	1100 1001
Input 2	15	0000 1111
^ *Output*	207	1100 0110

In the above example, Python performed the XOR operation between its input and returns the result as 1100 0110 which is the decimal equivalent of 207.

- ~ : This is a bitwise *NOT* operator. It is an unary operator that take only one input and inverts all the bits of an input, and returns the inverted bits. Consider the following truth table

Bits	**1**	**2**
Input	0	1
Output	1	0

- << : This is a bitwise left shift operator. It takes two inputs: *number to operate on* and *number of bits to shift*. It shifts bits to the left by pushing zeros in from the right and let the leftmost bits fall off. Consider the following example:

```
In []: 15 << 2
Out[]: 60
```

In the above example, we are shifting the number 15 left by 2 bits. The first input refers to the number to operate on and the second input refers to the number of bits of shift. We compute the truth table for the above operation as below:

		Binary
Input	15	0000 1111
<< *Output*	60	0011 1100

- >> : Similar to the left shift operator, we have a shift right operator that shifts bits right and fills zero on the left. While shifting bits to right, it let the rightmost bits fall off and add new zeros to the left.

```
In []: 201 >> 2
Out[]: 50
```

In the above example, the number 201 gets shifted right by 2 bits and we get 50 as an output. Its integrity can be verified by the following truth table.

		Binary Numbers
Input	201	1100 1001
>> *Output*	50	0011 0010

Bitwise operators find great significance in the quantitative trading domain. They are used to determine the trading signals based on different conditions. Generally, we assign 1 to the *buy signal*, -1 to the *sell signal* and 0 to the *no signal*. If we have multiple buy conditions and need to check if all conditions are satisfied, we use bitwise `&` operator to determine if all buy conditions are 1 and buy the asset under consideration. We will look at these things in more detail when we discuss *pandas* and *numpy* libraries.

6.2.5 Assignment operators

As the name suggests, assignment operators are used to assign values to variables.

- = : This operator assigns the value on its right side to the operand on its left.

```
In []: a = 5
In []: b = 3
```

We can also use this operator to assign multiple values to multiple operands on the left side. Number of values and operands must be same on both sides, else Python will throw an error.

```
In []: a, b = 5, 3

# Error line. Number of operands on both sides should
# be same.
In []: a, b = 5, 3, 8
Traceback (most recent call last):

File "<ipython-input-1-file>", line 1, in <module>
    a, b = 5, 3, 8

ValueError: too many values to unpack (expected 2)
```

- `+=` : This operator adds the operand on the right side with the operand on the left side and assigns the result back to the same operand on the left side.

```
In []: a += 2

In []: print(a)
Out[]: 7

# The above operation is same as the one mentioned below
In []: a = a + 2
```

- `-=` : This operator subtracts the operand on the right side with the operand on the left side and assigns the result back to the same operand on the left side.

```
In []: a -= 2

In []: print(a)
Out[]: 5
```

- `*=` : This operator multiplies the operand on the right side with the operand on the left side and assigns the result back to the same operand on the left side.

```
In []: a *= 2

In []: print(a)
Out[]: 10
```

- `/=` : This operator divides the operand on the left side by the operand on the right side and assigns the result back to the same operand on the left side.

```
In []: a /= 3

In []: print(a)
Out[]: 3.3333333333333335
```

- `%=` : This operator performs the division operation between operands and assigns the remainder to the operand on the left.

```
In []: a = 10
In []: a %= 3

In []: print(a)
Out[]: 1
```

- `**=` : This operator performs the exponential operation between operands and assigns the result to the operand on the left.

```
In []: a **= 3

In []: print(a)
Out[]: 8
```

- `//=` : This operator divides the left operand with the right operand and then assigns the result (floored to immediate integer) to the operand on the left.

```
In []: a = 10
In []: a //= 4

In []: print(a)
Out[]: 2
```

- `&=` : This operator performs the bitwise 'AND' operation between the operands and then assigns the result to the operand on the left side.

```
In []: a = 0
In []: a &= 1

# & operation results into 0 as one operand is 0 and
# the other is 1.
In []: print(a)
Out[]: 0
```

- `|=` : This operator performs the bitwise 'OR' operation between the operands and then assigns the result to the operand on the left side.

```
In []: a = 0
In []: a |= 1

# | operation results into 1
In []: print(a)
Out[]: 1
```

- `^=` : This operator performs the bitwise 'XOR' operation between the operands and then assigns the result to the operand on the left side.

```
In []: a = 1
In []: a ^= 1

# ^ operation results into 0 as both operands will be 1
In []: print(a)
Out[]: 0
```

- `>>=` : This operator shifts bits of the left operand to the right specified by the right operand and then assigns the new value to the operand on the left side.

```
In []: a = 32
In []: a >>= 2

In []: print(a)
Out[]: 8
```

- `<<=` : This operator shifts bits of the left operand to the left specified by the left operand and then assigns the new value to the operand on the left side.

```
In []: a = 8
In []: a <<= 2

In []: print(a)
Out[]: 32
```

6.2.6 Membership operators

These operators are used check whether the value/variable exist in a sequence or not.

- `in` : This operator returns `True` if a value exists in a sequence, `False` otherwise.

```
In []: stock_list = ['GOOG', 'MSFT', 'AMZN', 'NFLX']

In []: 'GOOG' in stock_list
Out[]: True

In []: 'AAPL' in stock_list
Out[]: False
```

- `not in` : This operator returns `True` if a value *does not* exists in a sequence, `False` otherwise.

```
In []: 'AAPL' not in stock_list
Out[]: True

In []: 'GOOG' not in stock_list
Out[]: False
```

6.2.7 Identity operators

These operators are used to check if two values or objects belong to same memory location or refer to same instance in Python. They can be used in the following way:

- `is` : This operator returns `True` if both operands are identical, `False` otherwise.

```
In []: a = 3
In []: b = a

# True as both variables refers to same value in
# the memory
In []: a is b
Out[]: True

In []: x = 3

# True as Python will create new reference of variable 'x'
# to value 3 on the same memory location
In []: x is a
Out[]: True
```

- `is not` : This operator returns `True` if both operands are *not* on same memory location, `False` otherwise.

```
In []: stock_list = ['AMZN', 'NFLX']
In []: my_list = ['AMZN', 'NFLX']

In []: stock_list is my_list
Out[]: False

# Though both lists are identical, they will not be stored
# at the same memory location as lists are mutable.
In []: stock_list is not my_list
Out[]: True
```

6.2.8 Operator Precedence

In Python, expressions are evaluated from left to right order. That is, if there are multiple operators within an expression, Python will evaluate its value starting from left most operand and ultimately up to the right most operator. Consider the following example:

```
In []: 2 + 5 - 3 + 1
Out[]: 5
```

In the above example, Python will first evaluate `2 + 5` resulting into `7`, then subtracts `3` from it to get `4`, and finally adding `1` to obtain the final result of `5`. But this is not the case always. If we include more operators, Python will behave in the different manner. For example,

```
In []: 2 + 5 * 3
Out[]: 17
```

Based on the principle discussed above (left to right evaluation), Python should evaluate the above expression to `21`, but instead, it returned `17`. Here, the Python first evaluated `5 * 3` resulting into 15 and then added `2` to obtain the final value of `17`, because the operator `*` has higher precedence over the `+`.

If there are multiple operators in an expression, Python will execute operators at same precedence from the left to right order starting from operators having the highest precedence. Following table lists the operators from highest precedence to lowest.

Operators	**Precedence**
`()`	Parentheses
`**`	Exponential
`+, -, ~`	Positive, Negative, Bitwise NOT
`*, /, //, %`	Multiplication, Division, Floor Division, Modulus
`+, -`	Addition, Subtraction
`«, »`	Bitwise Left, Bitwise Right
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==, !=, >, >=, <, <=, is, is not, in, not in`	Comparison, Identity, Membership Operators
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR

As the above table lists the `()` with the highest precedence, it can be used to change the precedence of any operator to be highest. Any expression written inside the parentheses `()` gets highest precedence and evaluated first.

```
In []: (5 / 2) * (2 + 5)
Out[]: 17.5
```

In the above example, the order of evaluation will be `(5 / 2)` resulting into `2.5` followed by `(2 + 5)` evaluating to `7` and finally `2.5` multiplied with `7` with the multiplication `*` operator providing the output as `17.5`.

6.3 Key Takeaways

1. Keywords are reserved words in Python. They can be used only in certain predefined ways when we code. They are always available and cannot be used as a variable name, class name, function name or any other identifier.
2. Keywords are case-sensitive. All keywords are in lowercase except `True`, `False` and `None`.
3. In Python, logical expressions are evaluated from left to right.
4. Operators are special constructs or symbols which are used to perform operations on variables and literals.
5. Arithmetic operators are used for performing various mathematical operations.
6. Comparison operators are used for comparing two or more values. It works with almost all data types and returns either `True` or `False` as an output.
7. Logical operators are used for comparing two or more conditional statements or expressions. They return boolean `True` or `False` as an output.
8. Bitwise operators act on bits and perform bit by bit operations. These operators perform operations on a binary (`0` and `1`) equivalent of a decimal number.
9. Assignment operators are used for assigning values to variables.
10. Membership operators check whether the given value exists in a data structure or not.

11. Identity operators check if two objects belong to the same memory location or refer to the same instances or not.
12. Each operator has specific precedence in Python. The precedence of any expression can be changed using the parenthesis `()`.

Chapter 7

Control Flow Statements

The code we write gets executed in the order they are written. In other words, a program's *control flow* is the order in which the program's code executes. Using conditional statements such as `if` statements, and loops, we can define or alter the execution order of the code. This section covers a conditional `if` statement and `for` and `while` loops; functions are covered in the upcoming section. Raising and handling exceptions also affects the control flow which will be discussed in subsequent sections.

7.1 Conditional Statements

Often times it is required that a code should execute only if a condition holds true, or depending on several mutually exclusive conditions. Python allows us to implement such a scenario using an `if` statement.

7.1.1 The `if` statement

The `if` statement is used when we want a code to be executed under certain conditions only. It can be either a single condition or multiple conditions. The code within the `if` block will be executed if and only if the logical conditions are held true. We use a comparison operator to check the truthfulness of the condition. The `if` statement evaluates the output of any logical condition to be either `True` or `False`, and the codes within the `if` block gets executed if the condition is evaluated true.

Let us consider a scenario where we want to go long on a stock if `buy_condition` is `True`.

```
# Buy a stock when the buy condition is true
if buy_condition == True:
    position = 'Buy'
```

In Python, we use the colon : to mark the end of the statement and start of the block. This is true for any statement such as a class or function definition, conditional statements, loops, etc. Notice the : at the end of the `if` statement which marks the start of the `if` block. The code is indented inside the block to depict that it belongs to a particular block. To end the block, we just write the code without any indentation.

In the above statement, the code statement `position = 'Buy'` is said to be inside the if block. If the variable or logical condition itself tends to be boolean i.e. True or False, we can directly write the condition without comparing. We can re-write the above example as shown below:

```
# Here, buy_condition itself is a boolean variable
if buy_condition:
    position = 'Buy'
```

In another scenario we want to buy a stock when an indicator is below 20, we can use the `if` statement as depicted in the following example:

```
# Buy a stock when an indicator (RSI value) is less than
# or equal to 20
if rsi_indicator <= 20:
    position = 'Buy'
```

Additionally, two or more conditions can be combined using any logical operator such as `and`, `or`, etc. In a more refined scenario, we might want to go long on a stock only if two conditions are true. We can do so in the following way:

```
# Input
if buy_condition_1 == True and rsi_indicator <= 20:
    position = 'Buy'
```

Similar to the above scenario, we can compound the if condition to be as complex as we want it to be, using different combinations of logical operators.

7.1.2 The `elif` clause

The `elif` clause checks for new conditions and executes the code if they are held true after the conditions evaluated by the previous `if` statement weren't true. In a scenario with mutually exclusive conditions, we might want the code to execute when one set of condition/s fails and another holds true. Consider the following example:

```
# Input
if buy_condition_1 == True and rsi_indicator <= 20:
    position = 'Buy'
elif sell_condition_1 and rsi_indicator >= 80:
    position = 'Sell'
```

During the execution, the interpreter will first check whether the conditions listed by the `if` statement holds true or not. If they are true, the code within the `if` block will be executed. Otherwise, the interpreter will try to check the conditions listed by the `elif` statement and if they are true, the code within the `elif` block will be executed. And if they are false, the interpreter will execute the code following the `elif` block. It is also possible to have multiple `elif` blocks, and the interpreter will keep on checking the conditions listed by each `elif` clause and executes the code block wherever conditions will be held true.

7.1.3 The `else` clause

The `else` clause can be thought of as the last part of conditional statements. It does not evaluate any conditions. When defined it just executes the code within its block if all conditions defined by the `if` and `elif` statements are false. We can use the `else` clause in the following way.

```
# Input
if buy_condition_1 == True and rsi_indicator <= 20:
    position = 'Buy'
elif sell_condition_1 and rsi_indicator >= 80:
```

```
        position = 'Sell'
    else:
        position = 'None'
```

In the above example, if the conditions listed by the `if` and `elif` clauses are false, the code within the `else` block gets executed and the variable `position` will be assigned a value `'None'`.

7.2 Loops

Let us consider a scenario where we want to compare the value of the variable `rsi_indicator` multiple times. To address this situation, we need to update the variable each time manually and check it with the `if` statement. We keep repeating this until we check all the values that we are interested in. Another approach we can use is to write multiple `if` conditions for checking multiple values. The first approach is botched and cumbersome, whereas the latter is practically non-feasible.

The approach we are left with is to have a range of values that need to be logically compared, check each value and keep iterating over them. Python allows us to implement such approach using loops or more precisely the `while` and `for` statements.

7.2.1 The `while` statement

The `while` statement in Python is used to repeat execution of code or block of code that is controlled by a conditional expression. The syntax for a `while` loop is given below:

```
while conditional expression:
    code statement 1
    code statement 2
    ...
    code statement n
```

A `while` statement allows us to repeat code execution until a conditional expression becomes true. Consider the following while loop:

```
# Input
data_points = 6
count = 0

while count != data_points:
    print(count)
    count += 1
```

The `while` statement is an example of what is called a *looping* statement. The above loop will print 6 digits starting from 0 up to 5 and the output will be the following:

```
# Output
0
1
2
3
4
5
```

When the above code is run, the interpreter will first check the conditional expression laid by the `while` loop. If the expression is `false` and the condition is *not* met, it will enter the loop and executes the code statements within the loop. The interpreter will keep executing the code within the loop until the condition becomes true. Once the condition is true, the interpreter will stop executing the code within the loop and move to the next code statement. A `while` statement can have an optional `else` clause. Continuing the above example, we can add the `else` clause as shown in the below example:

```
# Input
data_points = 6
count = 0

while count != data_points:
    print(count)
    count += 1
else:
    print('The while loop is over.')
```

In the above example, the interpreter will execute the while loop as we discussed above. Additionally, when the condition becomes true, the interpreter will execute the `else` clause also and the output will be as follows:

```
# Output
0
1
2
3
4
5
The while loop is over.
```

7.2.2 The `for` statement

The `for` statement in Python is another looping technique which *iterates* over a sequence of objects. That is, it will go through each item in a sequence. A sequence may be either a list, tuple, dictionary, set or string. The syntax of a `for`loop is as follows:

```
for item in sequence:
    code statement 1
    code statement 2
    ...
    code statement n
```

The `for` statement is also known as `for..in` loop in Python. The `item` in the above syntax is the placeholder for each item in the sequence.

> The `for` loop in Python is different as compared to other programming languages. We shall now see some of its avatars below.

7.2.3 The `range()` function

Python provides the built-in function `range()` that is used to generate sequences or arithmetic progressions which in turn can be combined with the `for` loop to repeat the code.

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number. The syntax of range() is as follows:

```
range([start,] stop [, step])

Parameter Values:-
start : Optional. An integer specifying at which number to
        start. The default is 0.
stop  : Required. An integer specifying at which number to end.
step  : Optional. An integer specifying the incrementation.
        The default is 1.
```

The `range()` function can be used along with the `for` loop as follows:

```
# Input
for i in range(5):
    print(i)
```

Here, we have provided only *stop* parameter value as 5. Hence, the `range()` function will start with 0 and end at 5 providing us with a sequence of 5 numbers. The output of the above `for` loop will be the following:

```
# Output
0
1
2
3
4
```

In the above `for` loop, the variable *i* will take the value of 0 generated by the `range()` function for the first iteration and execute the code block following it. For the second iteration, the variable *i* will take the value of 1 and again execute the code block following it and such repetition will continue until the last value is yielded by the `range()` function.

It is also possible to use various combinations of the start, stop and step parameters in a `range()` function to generate any sort of sequence. Consider the following example:

```
# Input
for i in range(1, 10, 2):
    print(i)
```

The above `range()` function will generate the sequence starting from 1 up to 10 with an increment of 2 and the output will be the following:

```
# Output
1
3
5
7
9
```

7.2.4 Looping through lists

With the `for` loop we can execute a set of code, once for each item in a list. The `for` loop will execute the code for all elements in a list.

```
# Input
top_gainers = ['BHARTIARTL', 'EICHERMOT', 'HCLTECH',
               'BAJFINANCE', 'RELIANCE']

for gainer in top_gainers:
    print(str(top_gainers.index(gainer)) + ' : ' + gainer)
```

Here the `for` loop will iterate over the list `top_gainers` and it will print each item within it along with their corresponding index number. The output of the above `for` loop is shown below:

```
# Output
0 : BHARTIARTL
1 : EICHERMOT
2 : HCLTECH
3 : BAJFINANCE
4 : RELIANCE
```

7.2.5 Looping through strings

Strings in Python are iterable objects. In other words, strings are a sequence of characters. Hence, we can use a string as a sequence object in the `for` loop.

```
volume_name = 'Python'

for character in volume_name:
    print(character)
```

We initialize the string `volume_name` with the value 'Python' and provide it as an iterable object to the `for` loop. The `for` loop yields each character from the it and prints the respective character using the `print` statement. The output is shown below:

```
# Output
P
y
t
h
o
n
```

7.2.6 Looping through dictionaries

Another sequential data structure available at our disposal is *dictionary*. We learnt about dictionaries in detail in the previous section. Looping through dictionaries involves a different approach as compared to lists and strings. As dictionaries are not index based, we need to use its built-in `items()` method as depicted below:

```
dict = {'AAPL':193.53,
        'HP':24.16,
        'MSFT':108.29,
        'GOOG':1061.49}

for key, value in dict.items():
  print(f'Price of {key} is {value}')
```

If we execute the command `dict.items()` directly, Python will return us a collection of a dictionary items (in form of tuples). as shown below:

```
# Input
dict.items()

# Output
dict_items([('AAPL', 193.53), ('HP', 24.16),
            ('MSFT', 108.29), ('GOOG', 1061.49)])
```

As we are iterating over tuples, we need to fetch a key and value for each item in the `for` loop. We fetch the key and value of each item yielded by the `dict.items()` method in the `key` and `value` variables and the output is shown below:

```
# Output
Price of AAPL is 193.53
Price of HP is 24.16
Price of MSFT is 108.29
Price of GOOG is 1061.49
```

In Python version 2.x, we need to use the method `iteritems()` of the dictionary object to iterate over its items.

A `for` loop can also have an optional `else` statement which gets executed once the `for` loop completes iterating over all items in a sequence. Sample `for` loop with an optional `else` statement is shown below:

```
for item in range(1, 6):
  print(f'This is {item}.')
else:
  print('For loop is over!')
```

The above `for` loop prints five statements and once it completes iterating over the `range()` function, it will execute the `else` clause and the output will be the following:

```
# Output
This is 1.
```

```
This is 2.
This is 3.
This is 4.
This is 5.
For loop is over!
```

7.2.7 Nested loops

Often times it is required that we need to loop through multiple sequences simultaneously. Python allows the usage of one loop inside another loop. Consider a scenario where we want to generate multiplication tables of 1 up to 9 simultaneously. We can do so by using nested `for` loops as given below:

```
for table_value in range(1, 10):
  for multiplier in range(1, 11):
    answer = table_value * multiplier
    print(answer, end=' ')
  print()
else:
    print('For loop is over!')
```

The first `for` loop defines the range for table from 1 to 9. Similarly, the second or the inner `for` loop defines the multiplier value from 1 to 10. The `print()` in the inner loop has parameter `end=' '` which appends a space instead of default new line. Hence, answers for a particular table will appear in a single row. The output for the above nested loops is shown below:

```
# Output
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
For loop is over!
```

The same scenario can also be implemented using the `while` nested loops as given below and we will get the same output shown above.

```
# Input
table_value = 1

while table_value != 10:
  multiplier = 1
  while multiplier != 11:
    answer = table_value * multiplier
    print(answer, end=' ')
    multiplier += 1
  table_value += 1
  print()
else:
  print('While loop is over!')
```

In Python it is also possible to nest different loops together. That is, we can nest a `for` loop inside a `while` loop and vice versa.

7.3 Loop control statements

Python provides various ways to alter the code execution flow during the execution of loops. Three keywords to do so are `break`, `pass` and `continue`. Though we already got a glimpse of these keywords in the previous section, we will learn its usage in this section. These keywords can be used with any looping technique in Python. Here, we will learn its implementation using a `for` loop.

7.3.1 The `break` keyword

The `break` keyword is used to break the execution flow of a loop. When used inside a loop, this keyword stops executing the loop and the execution control shifts to the first statement outside the loop. For example, we can use this keyword to break the execution flow upon certain condition.

```
# Input
for item in range(1,10):
```

```
    print(f'This is {item}.')
    if item == 6:
      print('Exiting FOR loop.')
      break
print('Not in FOR loop.')
```

We define a `for` loop that iterates over a range of 1 to 9 in the above example. Python will try to execute the code block following the loop definition, where it will check if the item under consideration is 6. If true, the interpreter will break and exit the loop as soon as it encounters the `break` statement and starts executing the statement following the loop. The output of the above loop will be the following:

```
# Output
This is 1.
This is 2.
This is 3.
This is 4.
This is 5.
This is 6.
Exiting FOR loop.
Not in FOR loop.
```

7.3.2 The `continue` keyword

Similar to the `break` keyword discussed above, we have the `continue` keyword which will skip the current iteration and continue with the next iteration. Consider the below example:

```
# Input
for item in range(1,10):
  if item == 6:
    continue
    print('This statement will not be executed.')
  print(f'This is {item}.')
print('Not in FOR loop.')
```

Again, we define a `for` loop to iterate over a range of 1 to 9 and check whether the item under consideration is 6 or not? If it is, we skip that iteration and continues the loop. Python interpreter will not attempt to execute

any statement once it encounters the `continue` keyword. The output of the above `for` loop is shown below:

```
# Output
This is 1.
This is 2.
This is 3.
This is 4.
This is 5.
This is 6.
This is 7.
This is 8.
This is 9.
Not in FOR loop.
```

As seen in the output above, the interpreter didn't print anything once it encountered the `continue` keyword thereby skipping the iteration.

7.3.3 The `pass` keyword

Essentially the `pass` keyword is not used to alter the execution flow, but rather it is used merely as a placeholder. It is a null statement. The only difference between a comment and a `pass` statement in Python is that the interpreter will entirely ignore the comment whereas a `pass` statement is not ignored. However, nothing happens when `pass` is executed.

In Python, loops cannot have an empty body. Suppose we have a loop that is not implemented yet, but we want to implement it in the future, we can use the `pass` statement to construct a body that does nothing.

```
# Input
stocks = ['AAPL', 'HP', 'MSFT', 'GOOG']

for stock in stocks:
  pass
else:
  print('For loop is over!')
```

In the loop defined above, Python will just iterate over each item without producing any output and finally execute the `else` clause. The output will be as shown below:

```
# Output
For loop is over!
```

7.4 List comprehensions

List comprehension is an elegant way to define and create a list in Python. It is used to create a new list from another sequence, just like a mathematical set notation in a single line. Consider the following set notation:

```
{i^3: i is a natural number less than 10}
```

The output of the above set notation will be cubes of all natural numbers less than 10. Now let's look at the corresponding Python code implementing list comprehension.

```
[i**3 for i in range(0,10)]
```

As we see in the Python code above, list comprehension starts and ends with square brackets to help us remember that the output will be a list. If we look closely, it is a `for` loop embedded in the square bracket. In a general sense, a `for` loop works as follows:

```
for item in sequence:
    if condition:
        output expression
```

The same gets implemented in a simple list comprehension construct in a single line as:

```
[output expression for item in sequence if condition]
```

As shown above, the syntax for *list comprehension* starts with the opening square bracket `[` followed by `output expression`, `for` loop, and optional `if` condition. It has to be ended with the closing square bracket `]`.
The set defined above can also be implemented using the `for` loop in the following way:

```
# Input
cube_list = []

for i in range(0,10):
    cube_list.append(i**3)
```

The corresponding list comprehension is constructed in the following way:

```
# Input
[i**3 for i in range(0,10)]
```

The output for the `for` loop and the list comprehension defined above will be the same shown below:

```
# Output
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

We can filter the output produced by a list comprehension using the condition part in its construct. Consider the revised set notation given below:

```
{i^3: i is a whole number less than 20, i is even}
```

The set defined above contains cubes of all whole numbers which are less than 20 and even. It can be implemented using the `for` loop as given below:

```
# Input
cube_list = []

for i in range(1,20):
    if i%2==0:
        cube_list.append(i**3)

print(cube_list)

# Output
[8, 64, 216, 512, 1000, 1728, 2744, 4096, 5832]
```

The output we got is in line with the set defined above and the `for` loop defined above can be implemented in a single line using the LC construct.

```
# Input
[i**3 for i in range(1,20) if i%2==0]
```

With a list comprehension, it does not have to be a single condition. We can have multiple conditions to filter the output produced by it. Suppose, we want to have a list of all positive numbers less than 20 which are divisible by 2 and 3 both. Such a list can be generated as follows:

```
# Input
[i for i in range(0,20) if i%2==0 if i%3==0]

#Output
[0, 6, 12, 18]
```

Python provides a flexible way to integrate `if` conditions within a list comprehension. It also allows us to embed the `if...else` condition. Let us segregate a list of positive numbers into `Odd` and `Even` using a comprehension construct.

```
# Input
[str(i)+': Even' if i%2==0 else str(i)+': Odd' for i in
 range(0,6)]
```

In such a scenario, we need to put the `if` and `else` part of a condition before the `for` loop in the comprehension. The output of the above construct is as below:

```
# Output
['0: Even', '1: Odd', '2: Even', '3: Odd', '4: Even',
 '5: Odd']
```

Finally, we can use a list comprehension to write a nested `for` loop. We resort to normal `for` loop and implement the multiplication table of 7 using the nested `for` loops in the following example:

```
# Input
for i in range(7,8):
    for j in range(1,11):
        print(f'{i} * {j} = {i * j}')
```

```
# Output
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
```

Such nested `for` loops can be implemented using a comprehension in the following way:

```
# Input
[i * j for j in range(1,11) for i in range(7,8)]
```

Here, the output will only be the result of multiplication as shown below:

```
# Output
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

Is it possible to produce the exact output using comprehension as generated by nested `for` loops above? The answer is *yes*. Go ahead and give it a try. This brings us to an end of this section. Here we took a deep dive into a conditional statement and various looping techniques in Python. We learned about ways to implement `if` conditions within a `for` and `while` loop and explored list comprehensions and their applications.

7.5 Key Takeaways

1. Control flow statements are Python constructs that alter the flow of the execution.
2. The conditional `if` statement is used when code needs to be executed based on some condition.
3. The `if` statement can evaluate multiple conditions.

4. The `elif` statement can be used in case of multiple mutually exclusive conditions.
5. The `else` statement can be used when code needs to be executed if all previous conditions fail.
6. Loops are used to perform an iterative process. In other words, loops are used to execute the same code more than one time.
7. A loop can be implemented using: a `while` statement and a `for` statement.
8. A counter needs to be coded explicitly for a `while` loop, else it might run infinitely.
9. The `range()` function is used to generate sequences in Python.
10. A loop within a loop is known as a nested loop.
11. A `for` loop is used to iterate over data structures such as lists, tuples, dictionaries and string as well.
12. The `break` keyword is used to break the execution of a loop and directs the execution flow outside the loop.
13. The `continue` keyword is used to skip the current iteration of a loop and moves the execution flow to the next iteration.
14. In Python, loops cannot have an empty body.
15. The `pass` keyword is used as a placeholder in an empty loop.
16. A list comprehension returns list. It consists of square brackets containing an expression that gets executed for each element in the iteration over a loop.

Chapter 8

Iterators & Generators

In this section we will explore the natural world of iterators, objects that we have already encountered in the context of `for` loops without necessarily knowing it, followed by its easier implementation via a handy concept of generators. Let's begin.

8.1 Iterators

Iterators are everywhere in Python. They are elegantly implemented in `for` loop, comprehensions, etc. but they are simply hidden in plain sight. An iterator is an object that can be iterated upon and which will return data, one element at a time. It allows us to traverse through all elements of a collection, regardless of its specific implementation.

Technically, in Python, an iterator is an object which implements the iterator protocol, which in turn consists of the methods `__next__()` and `__iter__()`.

8.1.1 Iterables

An iterable is an object, not necessarily a data structure that can return an iterator. Its primary purpose is to return all of its elements. An object is known as iterable if we can get an iterator from it. Directly or indirectly it will define two methods:

- `__iter__()` method which returns the iterator object itself and is used while using the `for` and `in` keywords.
- `__next__()` method returns the next value. It also returns `StopIteration` error once all the objects have been traversed.

The Python Standard Library contains many iterables: lists, tuples, strings, dictionaries and even files and we can run a loop over them. It essentially means we have indirectly used the iterator in the previous section while implementing looping techniques.

All these objects have an `iter()` method which is used to get an iterator. Below code snippet returns an iterator from a tuple, and prints each value:

```
In []: stocks = ('AAPL', 'MSFT', 'AMZN')
In []: iterator = iter(stocks)

In []: next(iterator)
Out[]: 'AAPL'

In []: next(iterator)
Out[]: 'MSFT'

In []: iterator.__next__()
Out[]: 'AMZN'

In []: next(iterator)
Traceback (most recent call last):

File "<ipython-input-6>", line 1, in <module>
    next(iterator)

StopIteration
```

We use the `next()` function to iterate manually through all the items of an iterator. Also, the `next()` function will implicitly call the `__next__()` method of an iterator as seen in the above example. It will raise `StopIteration` error once we reach the end and there is no more data to be returned.

We can iterate manually through other iterables like strings and list, in the manner similar to one we used to iterate over the tuple int the above example. The more elegant and automated way is to use a `for` loop. The `for` loop actually creates an iterator object and executes the `next()` method for each loop.

We are now going to dive a bit deeper into the world of iterators and iterables by looking at some handy functions viz. the `enumerate()`, `zip()` and `unzip()` functions.

8.1.2 `enumerate()` function

The `enumerate()` function takes any iterable such as a list as an argument and returns a special *enumerate* object which consists of pairs containing an element of an original iterable along with their index within the iterable. We can use the `list()` function to convert the enumerate object into a list of tuples. Let's see this in practice.

```
In []: stocks = ['AAPL', 'MSFT', 'TSLA']

In []: en_object = enumerate(stocks)

In []: en_object
Out[]: <enumerate at 0x7833948>

In []: list(en_object)
Out[]: [(0, 'AAPL'), (1, 'MSFT'), (2, 'TSLA')]
```

The enumerate object itself is also iterable, and we can loop over while unpacking its elements using the following clause.

```
In []: for index, value in enumerate(stocks):
  ...:     print(index, value)

0 AAPL
1 MSFT
2 TSLA
```

It is the default behaviour to start an index with 0. We can alter this behaviour using the `start` parameter within the `enumerate()` function.

```
In []: for index, value in enumerate(stocks, start=10):
  ...:     print(index, value)

10 AAPL
11 MSFT
12 TSLA
```

Next, we have the `zip()` function.

8.1.3 The `zip()` function

The `zip()` function accepts an arbitrary number of iterables and returns a zip object which is an iterator of tuples. Consider the following example:

```
In []: company_names = ['Apple', 'Microsoft', 'Tesla']
In []: tickers = ['AAPL', 'MSFT', 'TSLA']

In []: z = zip(company_names, tickers)

In []: print(type(z))
<class 'zip'>
```

Here, we have two lists `company_names` and `tickers`. Zipping them together creates a zip object which can be then converted to list and looped over.

```
In []: z_list = list(z)

In []: z_list
Out[]: [('Apple', 'AAPL'), ('Microsoft', 'MSFT'),
        ('Tesla', 'TSLA')]
```

The first element of the `z_list` is a tuple which contains the first element of each list that was zipped. The second element in each tuple contains the corresponding element of each list that was zipped and so on. Alternatively, we could use a `for()` loop to iterate over a zip object print the tuples.

```
In []: for company, ticker in z_list:
  ...:     print(f'{ticker} = {company}')
```

```
AAPL = Apple
MSFT = Microsoft
TSLA = Tesla
```

We could also have used the splat operator(*) to print all the elements.

```
In []: print(*z)
('Apple', 'AAPL') ('Microsoft', 'MSFT') ('Tesla', 'TSLA')
```

8.1.4 Creating a custom iterator

Let's see how an iterator works internally to produce the next element in a sequence when asked for. Python iterator objects are required to support two methods while following the iterator protocol. They are `__iter__()` and `__next__()`. The custom iterator coded below returns a series of numbers:

```
class Counter(object):
    def __init__(self, start, end):
        """Initialize the object"""
        self.current = start
        self.end = end

    def __iter__(self):
        """Returns itself as an iterator object"""
        return self

    def __next__(self):
        """Returns the next element in the series"""
        if self.current > self.end:
            raise StopIteration
        else:
            self.current += 1
            return self.current -1
```

We created a `Counter` class which takes two arguments `start` (depicts the start of a counter) and `end` (the end of the counter). The `__init__()` method is a constructor method which initializes the object with the `start` and `end`

parameters received. The `__iter__()` method returns the iterator object and the `__next__()` method computes the next element within the series and returns it. Now we can use the above-defined iterator in our code as shown below:

```
# Creates a new instance of the class 'Counter' and
# initializes it with start and end values
counter = Counter(1, 5)

# Run a loop over the newly created object and print its
# values
for element in counter:
    print(element)
```

The output of the above `for` loop will be as follows:

```
# Output
1
2
3
4
5
```

Remember that an iterator object can be used only once. It means once we have traversed through all elements of an iterator, and it has raised `StopIteration`, it will keep raising the same exception. So, if we run the above `for` loop again, Python will not provide us with any output. Internally it will keep raising the `StopIteration` error. This can be verified using the `next()` method.

```
In []: next(counter)
Traceback (most recent call last):

File "<ipython-input-18>", line 21, in <module>
    next(counter)

File "<ipython-input-12>", line 11, in __next__
    raise StopIteration

StopIteration
```

8.2 Generators

Python generator gives us an easier way to create iterators. But before we make an attempt to learn what *generators* in Python are, let us recall the list comprehension we learned in the previous section. To create a list of the first 10 even digits, we can use the comprehension as shown below:

```
In []: [number*2 for number in range(10)]
Out[]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Now, if we replace the square brackets `[]` in the above list comprehension with the round parenthesis `()`, Python returns something called generator objects.

```
In []: (number*2 for number in range(10))
Out[]: <generator object <genexpr> at 0x000000000CE4E780>
```

But what are actually generator objects? Well, a generator object is like list comprehension except it does not store the list in memory; it does not construct the list but is an object we can iterate over to produce elements of the list as required. For example:

```
In []: numbers = (number for number in range(10))

In []: type(numbers)
Out[]: generator

In []: for nums in numbers:
  ...:     print(nums)
  ...:
0
1
2
3
4
5
6
7
8
9
```

Here we can see that looping over a generator object produces the elements of the analogous list. We can also pass the generator to the function list() to print the list.

```
In []: numbers = (number for number in range(10))

In []: list(numbers)
Out[]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Moreover, like any other iterator, we can pass a generator to the function `next()` to iterate through its elements.

```
In []: numbers = (number for number in range(5))

In []: next(numbers)
Out[]: 0

In []: next(numbers)
Out[]: 1
```

This is known as *lazy evaluation*, whereby the evaluation of the expression is delayed until its value is needed. This can help a great deal when we are working with extremely large sequences as we don't want to store the entire list in memory, which is what comprehensions do; we want to generate elements of the sequences on the fly.
We also have generator functions that produce generator objects when called. They are written with the syntax of any other user-defined function, however, instead of returning values using the keyword `return`, they yield sequences of values using the keyword `yield`. Let us see it in practice.

```
def counter(start, end):
    """Generate values from start to end."""
    while start <= end:
        yield start
        start += 1
```

In the above function, the `while` loop is true until `start` is less than or equal to `end` and then the generator ceases to yield values. Calling the above function will return a generator object.

```
In []: c = counter(1, 5)

In []: type(c)
Out[]: generator
```

And again, as seen above, we can call the `list()` function or run a loop over generator object to traverse through its elements. Here, we pass the object `c` to the `list()` function.

```
In []: list(c)
Out[]: [1, 2, 3, 4, 5]
```

This brings us to an end of this section. Iterators are a powerful and useful tool in Python and generators are a good approach to work with lots of data. If we don't want to load all the data in the memory, we can use a generator which will pass us each piece of data at a time. Using the generator implementation saves memory.

8.3 Key Takeaways

1. An iterator is an object which can be iterated upon and will return data, one element at a time. It implements the iterator protocol, that is, `__next__()` and `__iter__()` methods.
2. An iterable is an object that can return an iterator.
3. Lists, Tuples, Strings, Dictionaries, etc. are iterables in Python. Directly or indirectly they implement the above-mentioned two methods.
4. Iterables have an `iter()` method which returns an iterator.
5. The `next()` method is used to iterate manually through all the items of an iterator.
6. The `enumerate()` function takes an iterable as an input and returns the enumerate object containing a pair of index and elements.
7. The `zip()` function accepts an arbitrary number of iterables and returns zip object which can be iterated upon.
8. Generators provides an easy way to create and implement iterators.
9. The syntax for generators is very similar to list comprehension, except that it uses a parentheses `()`.
10. Generators do not store elements in the memory and often creates the elements on the fly as required.

11. The `list()` method is used to convert generators to lists.

Chapter 9

Functions in Python

Let's now explore this remarkably handy feature seen in almost all programming languages: functions. There are lots of fantastic in-built functions in Python and its ecosystem. However, often, we as a Python programmer need to write custom functions to solve problems that are unique to our needs. Here is the definition of a function.

A function is a block of code(that performs a specific task) which runs only when it is called.

From the definition, it can be inferred that writing such block of codes, i.e. functions, provides benefits such as

- *Reusability*: Code written within a function can be called as and when needed. Hence, the same code can be reused thereby reducing the overall number of lines of code.
- *Modular Approach*: Writing a function implicitly follows a modular approach. We can break down the entire problem that we are trying to solve into smaller chunks, and each chunk, in turn, is implemented via a function.

Functions can be thought of as building blocks while writing a program, and as our program keeps growing larger and more intricate, functions help make it organized and more manageable. They allow us to give a name to a block of code, allowing us to run that block using the given name anywhere in a program any number of times. This is referred to as

calling a function. For example, if we want to compute the length of a list, we call a built-in `len` function. Using any function means we are calling it to perform the task for which it is designed.

We need to provide an input to the `len` function while calling it. The input we provide to the function is called an *argument*. It can be a data structure, string, value or a variable referring to them. Depending upon the functionality, a function can take single or multiple arguments.

There are three types of functions in Python:

- Built-in functions such as `print` to print on the standard output device, `type` to check data type of an object, etc. These are the functions that Python provides to accomplish common tasks.
- User-Defined functions: As the name suggests these are custom functions to help/resolve/achieve a particular task.
- Anonymous functions, also known as lambda functions are custom-made without having any name identifier.

9.1 Recapping built-in functions

Built-in functions are the ones provided by Python. The very first built-in function we had learned was the `print` function to print a string given to it as an argument on the standard output device. They can be directly used within our code without importing any module and are always available to use.

In addition to `print` function, We have learned the following built-in functions until now in the previous sections.

- `type(object)` is used to check the data type of an *object*.
- `float([value])` returns a floating point number constructed from a number or string *value*.
- `int([value])` returns an integer object constructed from a float or string *value*, or return 0 if no arguments are given.
- `round(number[, ndigits])` is used to round a float *number* up to digits specified by *ndigits*.

- `abs(value)` returns the absolute value of a *value* provided as an argument.
- `format(value[, format_spec])` converts a *value* to a 'formatted' representation, as controlled by *format_spec*.
- `str([object])` returns a string version of *object*. If the object is not provided, returns the empty string.
- `bool([value])` return a Boolean value, i.e. one of `True` or `False`. *value* is converted using the standard truth testing procedure[1]. If the *value* is false or omitted, this returns `False`; otherwise, it returns `True`.
- `dir([object])` returns the list of names in the current local scope when an argument is not provided. With an argument, it attempts to return a list of valid attributes for that object.
- `len(object)` returns the length (the number of items) of an *object*. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

It is worth noting that almost all built-in functions take one or more arguments, perform the specific operation on it and return the output. We will keep learning about many more built-in functions as we progress through our Python learning journey. More information about various built-in functions can be obtained from Python official documentation[2].

9.2 User defined functions

Although Python provides a wide array of built-in functions, it does not suffice in tackling issues we would face while we develop programs and applications. As Python programmers, we might need to break down the programming challenge into smaller chunks and implement them in the form of custom or *user defined* functions. The concept of writing functions is probably an essential feature of any programming language.

Functions are defined using the `def` keyword, followed by an *identifier* name along with the parentheses, and by the final colon that ends the line. The

[1] https://docs.python.org/3/library/stdtypes.html#truth
[2] https://docs.python.org/3/library/functions.html

block of statements which forms the *body* of the function follows the *function definition*. Here's a simple example.

```
def greet():
    """Block of statement.
    or Body of function.
    """
    print(' Hello from inside the function!')
```

The above defined `greet` function can be called using its name as shown here.

```
# Calling the function
greet()
```

And the output will be

```
Hello from inside the function.
```

9.2.1 Functions with a single argument

A function can be called as many times as we want, and Python will execute the statements within its body. This function neither takes any input nor does it return any output. It just prints the statement written within it. If the function has to take any input, it goes within the parentheses as a *parameter* during the function definition. Parameters are the values we supply to the function so that the function can do something utilizing those values.

Note the terminology used here:

- *Parameters*: They are specified within parentheses in the function definition, separated by commas.
- *Arguments*: When we call a function, values that parameters take are to be given as arguments in a comma separated format.

The modified version of the above simple function explains these two terms:

```
# Here 'person_name' is a parameter.
def greet(person_name):
    """Prints greetings along with the value received
    via the parameter."""
    print('Hello ' + person_name + '!')
```

The above function definition defines `person_name` as a parameter to the function `greet`, and it can be called as shown below:

```
# Calling the function
greet('Amigo')
```

The above call to the function `greet` takes a string `Amigo` as an argument and the output will be as follows:

```
Hello Amigo!
```

9.2.2 Functions with multiple arguments and a `return` statement

Both versions of the `greet` functions defined above were actually straightforward in terms of functionality that they perform. One more functionality that functions are capable of performing is to return a value to the calling statement using the keyword `return`. Consider a function that takes several parameters, performs some mathematical calculation on it and returns the output. For example:

```
# Function with two parameters 'a' and 'b'
def add(a, b):
    """Computes the addition and returns the result.

    It does not implement the print statement.
    """
    result = a + b # Computes addition
    return result # Returns the result variable
```

This user defined function `add` takes two parameters `a` and `b`, sums them together and assigns its output to a variable `result` and ultimately returns the variable to calling statement as shown below:

```
# Calling the add function
x = 5
y = 6
print(f'The addition of {x} and {y} is {add(x, y)}.')
```

We call the function `add` with two arguments `x` and `y` (as the function definition has two parameters) initialized with `5` and `6` respectively, and the addition returned by the function gets printed via the `print` statement as shown below:

```
The addition of 5 and 6 is 11.
```

Similarly, functions can also return multiple values based on the implementation. The following function demonstrates the same.

```
# Function definition
def upper_lower(x):
    """
    Returns the upper and lower version of the string.

    The value must be a string, else it will result in
    an error.
    This function does not implement any error handling
    mechanism.
    """
    upper = x.upper() # Convert x to upper string
    lower = x.lower() # Convert x to lower string
    return upper, lower # Return both variables
```

The above `upper_lower` function takes one argument `x` (a string) and converts it to their upper and lower versions. Let us call it and see the output.

> **NOTE:** The function upper_lower implicitly assumes to have a string as a parameter. Providing an integer or float value as an argument while calling will result in an error.

```
# Calling the function
upper, lower = upper_lower('Python')
```

```
# Printing output
print(upper)
PYTHON

print(lower)
python
```

Here, the call to `upper_lower` function has been assigned to two variables `upper` and `lower` as the function returns two values which will be unpacked to each variable respectively and the same can be verified in the output shown above.

9.2.3 Functions with default arguments

Let us say, we are writing a function that takes multiple parameters. Often, there are common values for some of these parameters. In such cases, we would like to be able to call the function without explicitly specifying every parameter. In other words, we would like some parameters to have default values that will be used when they are not specified in the function call.

To define a function with a default argument value, we need to assign a value to the parameter of interest while defining a function.

```
def power(number, pow=2):
    """Returns the value of number to the power of pow."""
    return number**pow
```

Notice that the above function computes the first argument to the power of the second argument. The default value of the latter is 2. So now when we call the function `power` only with a single argument, it will be assigned to the `number` parameter and the return value will be obtained by squaring `number`.

```
# Calling the power function only with required argument
print(power(2))

# Output
4
```

In other words, the argument value to the second parameter `pow` became optional. If we want to calculate the number for a different power, we can obviously provide a value for it and the function will return the corresponding value.

```
# Calling the power function with both arguments
print(power(2, 5)

# Output
32
```

We can have any number of default value parameters in a function. Note however that they must follow non-default value parameters in the definition. Otherwise, Python will throw an error as shown below:

```
# Calling the power function that will throw an error
def power(pow=2, number):
    """Returns the raised number to the power of pow."""
    return number**pow

File "<ipython-input-57>", line 1
    def power(pow=2, number):
              ^
SyntaxError: non-default argument follows default argument
```

9.2.4 Functions with variable length arguments

Let's consider a scenario where we as developers aren't sure about how many arguments a user will want to pass while calling a function. For example, a function that takes floats or integers (irrespective of how many they are) as arguments and returns the sum of all of them. We can implement this scenario as shown below:

```
def sum_all(*args):
    """Sum all values in the *args."""

    # Initialize result to 0
    result = 0
```

```
    # Sum all values
    for i in args:
        result += i

    # Return the result
    return result
```

The flexible argument is written as * followed by the parameter name in the function definition. The parameter `args` preceded by * denotes that this parameter is of variable length. Python than unpacks it to a *tuple* of the same name `args` which will be available to use within the function. In the above example, we initialize the variable `result` to 0 which will hold the sum of all arguments. We then loop over the `args` to compute a sum and update the `result` with each iteration. Finally, we return the sum to the calling statement. The `sum_all` function can be called with any number of arguments and it will add them all up as follows:

```
# Calling the sum_all function with arbitrary number of
# arguments.
print(sum_all(1, 2, 3, 4, 5))

# Output
15

# Calling with different numbers of arguments.
print(sum_all(15, 20, 6))

# Output
41
```

Here, `*args` is used as the parameter name (the shorthand for *arguments*), but we can use any valid identifier as the parameter name. It justs needs to be preceded by * to make it flexible in length. On the same lines, Python provides another flavor of flexible arguments which are preceded by double asterisk marks. When used ,they are unpacked to *dictionaries* (with the same name) by the interpreter and are available to use within the function. For example:

```
def info(**kwargs):
    """Print out key-value pairs in **kwargs."""
```

```
    # Run for loop to prints dictionary items
    for key, value in kwargs.items():
        print(key + ': ' + value)
```

Here, the parameter `**kwargs` are known as *keywords arguments* which will be converted into a dictionary of the same name. We then loop over it and print all keys and values. Again, it is totally valid to use an identifier other than `kwargs` as the parameter name. The `info` function can be called as follows:

```
# Calling the function
print(info(ticker='AAPL', price='146.83',
name='Apple Inc.', country='US'))

# Output
ticker: AAPL
price: 146.83
name: Apple Inc.
country: US
```

That is all about the default and flexible arguments. We now attempt to head towards the documentation part of functions.

9.2.5 DocStrings

Python has a nifty feature called *documentation string*, usually referred to by its shorter name *docstrings*. This is an important but not required tool that should be used every time we write a program since it helps to document the program better and makes it easier to understand.

Docstrings are written within triple single/double quotes just after definition header. They are written on the first logical line of a function. Docstrings are not limited to functions only; they also apply to modules and classes. The convention followed for a docstring is a multi-line string where the first line starts with a capital letter and ends with a dot. The second line is blank followed by any detailed explanation starting from the third line. It is strongly advised to follow this convention for all docstrings. Let's see this in practice with the help of an example:

```
def power(x, y):
    """
    Equivalent to x**y or built-in pow() with two
    arguments.

    x and y should be numerical values else an appropriate
    error will be thrown for incompatible types.

    Parameters:
    x (int or float): Base value for the power operation.
    y (int or float): Power to which base value should be
                      raised.

    Returns:
    int or float: It returns x raised to the power of y.
    """

    try:
        return x ** y
    except Exception as e:
        print(e)
```

The function `power` defined above returns the raised value of the argument `x` powered to `y`. The thing of our interest is the docstring written within `'''` which documents the function. We can access a docstring of any function using the `__doc__` attribute (notice the *double underscores*) of that function. The docstring for the `power` function can be accessed with the following code:

```
print(power.__doc__)
```

And the output is shown below:

```
Equivalent to x**y or built-in pow() with two arguments.

x and y should be numerical values else an appropriate
error will be thrown for incompatible types.

Parameters:
```

```
x (int or float): Base value for the power operation.
y (int or float): Power to which base value should be
                  raised.

Returns:
int or float: It returns x raised to the power of y.
```

We have already seen the indirect usage of docstrings in previous sections. When we use a function `help` in Python, it will show up the docstring. What it does is fetch the `__doc__` attribute of that function and displays it in a neat manner. If we ask for the help on the user defined `power` using the `print(help(power))`, Python will return the same output as shown above that we got using the `print(power.__doc__)`.

9.2.6 Nested functions and non-local variable

A nested function is a function that is defined inside another function. The syntax for the nested function is the same as that of any other function. Though the applications of nested functions are complex in nature and limited at times, even in the quant domain, it is worth mentioning it, as we might encounter this out there in the wild. Below is an example which demonstrates the nested functions.

```
# Defining nested function
def outer():
    """This is an enclosing function"""
    def inner():
        """This is a nested function"""
        print('Got printed from the nested function.')

    print('Got printed from the outer function.')
    inner()
```

We define the function `outer` which nests another function `inner` within it. The `outer` function is referred to as an *enclosing* function and `inner` is known as *nested* function. They are also referred to as *inner* functions sometimes. Upon calling the `outer` function, Python will, in turn, call the `inner` function nested inside it and execute it. The output for the same is shown below:

```
# Calling the 'outer' function
outer()

# Output
Got printed from the outer function.
Got printed from the nested function.
```

The output we got here is intuitive. First, the `print` statement within the `outer` function got executed, followed by the `print` statement in the `inner` function. Additionally, nested functions can access variables of the enclosing functions. i.e. variables defined in the outer function can be accessed by the inner function. However, the inner or the nested function cannot modify the variables defined in the outer or enclosing function.

```
def outer(n):
    number = n

    def inner():
        print('Number =', number)

    inner()
```

A call to `outer` function will print the following

```
outer(5)

# Output
Number = 5
```

Though the variable `number` is not defined within `inner` function, it is able to access and print the `number`. This is possible because of scope mechanism that Python provided. We discuss more on this in the following section. Now consider, what if we want the nested function to modify the variable that is declared in the enclosing function. The default behavior of Python does not allow this. If we try to modify it, we will be presented with an error. To handle such a situation, the keyword `nonlocal` comes to the rescue.

In the nested function, we use the keyword `nonlocal` to create and change the variables defined in the enclosing function. In the example that follows, we alter the value of the variable `number`.

```
def outer(n):

    number = n
    def inner():
        nonlocal number
        number = number ** 2
        print('Square of number =', number)

    print('Number =', number)
    inner()
    print('Number =', number)
```

A call to the `outer` function will now print the number passed as an argument to it, the square of it and the newly updated number (which is nothing but the squared number only).

```
outer(3)

# Output
Number = 3
Square of number = 9
Number = 9
```

Remember, assigning a value to a variable will only create or change the variable within a particular function (or a scope) unless they are declared using the nonlocal statement.

9.3 Variable Namespace and Scope

If we read the *The Zen of Python* (try `import this` in Python console), the last line states *Namespaces are one honking great idea -- let's do more of those!* Let's try to understand what these mysterious namespaces are. However, before that, it will be worth spending some time understanding *names* in the context of Python.

9.3.1 Names in the Python world

A *name* (also known as an identifier) is simply a name given to an object. From Python basics, we know that everything in Python are objects. And a name is a way to access the underlying object. Let us create a new variable with a name `price` having a value 144, and check the *memory location identifier* accessible by the function `id`.

```
# Creating new variable
price = 144

# Case 1: Print memory id of the variable price
print(id(price))

# Case 1: Output
1948155424

# Case 2: Print memory id of the absolute value 144
print(id(144))

# Case 2: Output
1948155424
```

Interestingly we see that the memory location of both cases (the variable and its assigned value) is the same. In other words, both refer to the same integer object. If you would execute the above code on your workstation, memory location would almost certainly be different, but it would be the same for both the variable and value. Let's add more fun to it. Consider the following code:

```
# Assign price to old_price
old_price = price

# Assign new value to price
price = price + 1

# Print price
print(price)
```

```
# Output
145

# Print memory location of price and 145
print('Memory location of price:', id(price))
print('Memory location of 145:', id(145))

# Output
Memory location of price: 1948155456
Memory location of 145: 1948155456

# Print memory location of old_price and 144
print('Memory location of old_price:', id(old_price))
print('Memory location of 144:', id(144))

# Output
Memory location of old_price: 1948155424
Memory location of 144: 1948155424
```

We increased the value of a variable `price` by 1 unit and see that the memory location of it got changed. As you may have guessed, the memory location of an integer object `145` would also be the same as that of `price`. However, if we check the memory location of a variable `old_price`, it would point to the memory location of integer object `144`. This is efficient as Python does not need to create duplicate objects. This also makes Python powerful in a sense that a name could refer to any object, even functions. Note that functions are also objects in Python. Now that we are aware of the nitty-gritty of names in Python, we are ready to examine namespaces closely.

9.3.2 Namespace

Name conflicts happen all the time in real life. For example, we often see that there are multiple students with the same name X in a classroom. If someone has to call the student X, there would be a conflicting situation for determining which student X is actually being called. While calling, one might use the last name along with the student's first name to ensure that the call is made to the correct student X.

Similarly, such conflicts also arise in programming. It is easy and manageable to have unique names when programs are small without any external dependencies. Things start becoming complex when programs become larger and external modules are incorporated. It becomes difficult and wearisome to have unique names for all objects in the program when it spans hundreds of lines.

A namespace can be thought of a naming system to avoid ambiguity between names and ensures that all the names in a program are unique and can be used without any conflict. Most namespaces are implemented as a dictionary in Python. There is a name to object mapping, with names as keys and objects as values. Multiple namespaces can use the same name and map it to a different object. Namespaces are created at different moments and have different lifetimes. Examples of namespaces are:

- The set of built-in names: It includes built-in functions and built-in exception names.
- The global names in a module: It includes names from various modules imported in a program.
- The local names in a function: It includes names inside a function. It is created when a function is called and lasts until the function returns.

The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; that is, two different modules can contain a function `sum` without any conflict or confusion. However, they must be prefixed with the module name when used.

9.3.3 Scopes

Until now we've been using objects anywhere in a program. However, an important thing to note is not all objects are always accessible everywhere in a program. This is where the concept of scope comes into the picture. A *scope* is a region of a Python program where a namespace is directly accessible. That is when a reference to a name (lists, tuples, variables, etc.) is made, Python attempts to find the name in the namespace. The different types of scopes are:

Local scope: Names that are defined within a local scope means they are defined inside a function. They are accessible only within a function. Names

defined within a function cannot be accessed outside of it. Once the execution of a function is over, names within the local scope cease to exist. This is illustrated below:

```
# Defining a function
def print_number():
    # This is local scope
    n = 10
    # Printing number
    print('Within function: Number is', n)

print_number()

# This statement will cause error when executed
print('Outside function: Number is', n)

# Output
Within function: Number is 10

Traceback (most recent call last):

File "<ipython-input-2>", line 8, in <module>
    print('Outside function: Number is', n)

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

Enclosing scope: Names in the enclosing scope refer to the names defined within enclosing functions. When there is a reference to a name that is not available within the local scope, it will be searched within the enclosing scope. This is known as scope resolution. The following example helps us understand this better:

```
# This is enclosing / outer function
def outer():

    number = 10

    # This is nested / inner function
    def inner():
```

```
            print('Number is', number)

        inner()

    outer()


    # Output
    Number is 10
```

We try to print the variable `number` from within the `inner` function where it is not defined. Hence, Python tries to find the variable in the `outer` function which works as an enclosing function. What if the variable is not found within the enclosing scope as well? Python will try to find it in the *global* scope which we discuss next.

Global scope: Names in the global scope means they are defined within the main script of a program. They are accessible almost everywhere within the program. Consider the following example where we define a variable `n` before a function definition (that is, within global scope) and define another variable with the same name `n` within the function.

```
# Global variable
n = 3

def relu(val):
    # Local variable
    n = max(0, val)
    return n

print('First statement: ', relu(-3))
print('Second statement:', n)

# Output
First statement:  0
Second statement: 3
```

Here, the first print statement calls the `relu` function with a value of `-3` which evaluates the maximum number to 0 and assigns the maximum

number to the variable `n` which in turn gets returned thereby printing 0. Next, we attempt to print the `n` and Python prints 3. This is because Python now refers to the variable `n` defined outside the function (within the global scope). Hence, we got two different values of `n` as they reside in different scopes. This brings us to one obvious question, what if the variable is not defined within the local scope, but available in the globals scope and we try to access that global variable? The answer is intuitive, we will be able to access it within the function. However, it would be a read-only variable and hence we won't be able to modify it. An attempt to modify a global variable result in the error as shown below:

```
# Global variable
number = 5

# Function that updates the global variable
def update_number():
    number = number + 2
    print('Within function: Number is', number)

# Calling the function
update_number()

print('Outside function: Number is', number)


# Output
Traceback (most recent call last):

File "<ipython-input-8>", line 8, in <module>
    update_number()

File "<ipython-input-8>", line 4, in update_number
    number = number + 2

UnboundLocalError: local variable 'number' referenced
 before assignment
```

To handle such a situation which demands modification of a global name, we define the global name within the function followed by the `global` key-

word. The `global` keywords allow us to access the global name within the local scope. Let us run the above code, but with the `global` keyword.

```
# Global variable
number = 5

# Function that updates the global variable
def update_number():
    global number
    number = number + 2
    print('Within function: Number is', number)

# Calling the function
update_number()

print('Outside function: Number is', number)

# Output
Within function: Number is 7
Outside function: Number is 7
```

The `global` keyword allowed us to modify the global variable from the local scope without any issues. This is very similar to the keyword `non-local` which allows us to modify variables defined in the enclosing scope.

Built-in scope: This scope consists of names predefined within built-ins module in Python such as `sum`, `print`, `type`, etc. Though we neither define these functions anywhere in our program nor we import them from any external module they are always available to use.

To summarize, when executing a Python code, names are searched in various scopes in the following order:

1. Local
2. Enclosing
3. Global
4. Built-in

If they are not found in any scope, Python will throw an error.

9.4 Lambda functions

We have written functions above using the `def` keyword, function headers, DocStrings and function bodies. There's a quicker way to write on-the-fly functions in Python and they are known as lambda functions. They are also referred to as anonymous functions sometimes. We use the keyword `lambda` to write such functions. The syntax for lambda functions is as follows:

```
lambda arguments: expression
```

Firstly, the syntax shows that there is no function name. Secondly, *arguments* refers to parameters, and finally, *expression* depicts the function body. Let us create a function `square` which squares the argument provided to it and returns the result. We create this function using the `def` keyword.

```
# Function defnition
def square(arg):
    """
    Computes the square of an argument and returns the
    result.

    It does not implement the print statement."""
    """
    result = arg * arg
    return result

# Calling the function and printing its output
print(square(3))

# Output
9
```

The function `square` defined above can be re-written in a single line using the `lambda` keyword as shown below:

```
# Creating a lambda function and assigning it to square
square = lambda arg: arg * arg
```

```
# Calling the lambda function using the name 'square'
print(square(3))

# Outpuut
9
```

In the above lambda function, it takes one argument denoted by *arg* and returns its square. Lambda functions can have as many number of arguments as we want after the `lambda` keyword during its definition. We will restrict our discussion up to two arguments to understand how multiple arguments work. We create another lambda function to raise the first argument to the power of the second argument.

```
# Creating a lambda function to mimic 'raise to power'
# operation
power = lambda a, b: a ** b

# Calling the lambda function using the name 'power'
print(power(2, 3))

# Output
8
```

Lambda functions are extensively used along with built-in `map` and `filter` functions.

9.4.1 `map()` Function

The `map` function takes two arguments: a function and a sequence such as a list. This function makes an iterator that applies the function to each element of a sequence. We can pass lambda function to this `map` function without even naming it. In this case, we refer to lambda functions as an anonymous function. In the following example, we create a list `nums` consisting of numbers and pass it to a `map` function along with the lambda function which will square each element of the list.

```
# Creating a list of all numbers
nums = [1, 2, 3, 4, 5]
```

```
# Defining a lambda function to square each number and
# passing it as an argument to map function
squares = map(lambda num: num ** 2, nums)
```

The lambda function in the above example will square each element of the list `nums` and the map function will map each output to the corresponding elements in the original list. We then store the result into a variable called `squares`. If we print the `square` variable, Python will reveal us that it is a map object.

```
# Printing squares
print(squares)

# Output
<map object at 0x00000000074EAD68>
```

To see what this object contains, we need to cast it to list using the `list` function as shown below:

```
# Casting map object squares to a list and printing it
print(list(squares))

# Output
[1, 4, 9, 16, 25]
```

9.4.2 `filter()` Function

The `filter` function takes two arguments: a function or `None` and a sequence. This function offers a way to filter out elements from a list that don't satisfy certain criteria. Before we embed a lambda function with it, let's understand how it works.

```
# Creating a list of booleans
booleans = [False, True, True, False, True]

# Filtering 'booleans', casting it to a list, and finally
# printing it
print(list(filter(None, booleans)))
```

```
# Output
[True, True, True]
```

In the above example, we first create a list of random boolean values. Next, we pass it to the `filter` function along with the `None` which specifies to return the items that are true. Lastly, we cast the output of the `filter` function to a list as it outputs a filter object. In a more advanced scenario, we can embed a lambda function in the `filter` function. Consider that we have been given a scenario where we need to filter all strings whose length is greater than 3 from a given set of strings. We can use filter and lambda functions together to achieve this. This is illustrated below:

```
# Creating a pool of random strings
strings = ['one', 'two', 'three', 'four', 'five', 'six']

# Filtering strings using a lambda and filter functions
filtered_strings = filter(lambda string: len(string) > 3,
                          strings)

# Casting 'filtered_strings' to a list and printing it
print(list(filtered_strings))

# Output
['three', 'four', 'five']
```

In the above example, a lambda function is used within the `filter` function which checks for the length of each string in the `strings` list. And the `filter` function will then filter out the strings which match the criteria defined by the lambda function.

Apart from the `map` and `filter` functions discussed above, now we will learn another handy function `zip` which can be used for iterating through multiple sequences simultaneously.

9.4.3 `zip()` Function

As regular computer users, we often comes across a file with *.zip* extension aka zip files. Basically, these files are the files which have zipped other files

within them. In other words, zip files work as a container to hold other files.

In the Python world, the `zip` function works more or less as a container for iterables instead of real files. The syntax for the `zip` is shown below:

```
zip(*iterables)
```

It takes an iterable as an input and returns the iterator that aggregates elements from each of the iterable. The output contains the iterator of a tuple. The *i-th* element in the iterator is the tuple consisting the *i-th* element from each input. If the iterables in the input are of unequal sizes, the output iterator stops when the shortest input iterable is exhausted. With no input, it returns an empty iterator. Let us understand the working of `zip` with the help of an example.

```
# Defining iterables for the input
tickers = ['AAPL', 'MSFT', 'GOOG']
companies = ['Apple Inc', 'Microsoft Corporation',
             'Alphabet Inc']

# Zipping the above defined iterables using the 'zip'
zipped = zip(tickers, companies)
```

We define two lists `tickers` and `companies` which are used as an input to the `zip`. The `zipped` object is the iterator of type `zip` and hence we can iterate either over it using a looping technique to print its content:

```
# Iterating over a zipped object
for ticker, company in zipped:
    print('Ticker name of {} is {}.'.format(ticker,
    company))

# Output
Ticker name of AAPL is Apple Inc.
Ticker name of MSFT is Microsoft Corporation.
Ticker name of GOOG is Alphabet Inc.
```

or cast it to sequential data structures such as list or tuple easily.

```
# Casting the zip object to a list and printing it
print(list(zipped))

# Output
[('AAPL', 'Apple Inc.'),
 ('MSFT', 'Microsoft Corporation'),
 ('GOOG', 'Alphabet Inc.')]
```

As we should expect, the zipped object contains a sequence of tuples where elements are from the corresponding inputs. A `zip` object in conjunction with `*` unzips the elements as they were before. For example:

```
# Unzipping the zipped object
new_tickers, new_companies = zip(*zipped)

# Printing new unzipped sequences
print(new_tickers)
('AAPL', 'MSFT', 'GOOG')

print(new_companies)
('Apple Inc.', 'Microsoft Corporation', 'Alphabet Inc.')
```

We unzip the zip object `zipped` to two sequences `new_tickers` and `new_companies`. By printing these sequences, we can see that the operation got successful and elements got unzipped successfully into respective tuples.

9.5 Key Takeaways

1. A function is a block of statements that can be reused as and when required.
2. There are three types of functions available in Python: Built-in functions, User-defined functions and anonymous functions using the `lambda` keyword.
3. Python provides various built-in functions to perform common programming tasks such as `print()`, `len()`, `dir()`, `type()`, etc
4. User-defined functions are defined using the keyword `def`.
5. Functions may take zero or more arguments. Arguments are specified while defining a function within parentheses `()`.

6. It is possible that arguments take some default value in the function definition. Such arguments are called default arguments.
7. Functions can return a value using the keyword `return`.
8. Functions can have variable-length arguments. There are two types of such arguments:
   (a) An argument that is preceded by `*` in the function definition can have a flexible number of values within it. And gets unpacked to a tuple inside a function.
   (b) An argument that is preceded by `**` in the function definition can have a flexible number of key-value pairs and gets unpacked to a dictionary inside a function.
9. A docstring is used to document a function and is written within triple single/double quotes. It can be accessed by the `__doc__` attribute on the function name.
10. Docstrings can be used to document modules and classes as well.
11. A namespace is a naming system in Python to avoid ambiguity between names (variable names, object names, module names, class names, etc.).
12. A scope is a region of a Python program where a namespace is directly accessible.
13. The `global` keyword is used when a variable that is defined outside a function and needs to be accessed from within a function.
14. The `lambda` keyword is used to create anonymous functions. Such functions are created during the run time and do not have any name associated with them.
15. `map()`, `filter()` and `zip()` functions are often used with anonymous functions.

Chapter 10

NumPy Module

NumPy, an acronym for *Numerical Python*, is a package to perform scientific computing in Python efficiently. It includes random number generation capabilities, functions for basic linear algebra, Fourier transforms as well as a tool for integrating Fortran and C/C++ code along with a bunch of other functionalities.

NumPy is an open-source project and a successor to two earlier scientific Python libraries: *Numeric* and *Numarray*.

It can be used as an efficient multi-dimensional container of generic data. This allows NumPy to integrate with a wide variety of databases seamlessly. It also features a collection of routines for processing single and multidimensional vectors known as arrays in programming parlance.

NumPy is not a part of the Python Standard Library and hence, as with any other such library or module, it needs to be installed on a workstation before it can be used. Based on the Python distribution one uses, it can be installed via a command prompt, conda prompt, or terminal using the following command. *One point to note is that if we use the Anaconda distribution to install Python, most of the libraries (like NumPy, pandas, scikit-learn, matplotlib, etc.) used in the scientific Python ecosystem come pre-installed.*

```
pip install numpy
```

NOTE: If we use the Python or iPython console to install the

NumPy library, the command to install it would be preceded by the character `!`.

Once installed we can use it by importing into our program by using the `import` statement. The de facto way of importing is shown below:

```
import numpy as np
```

Here, the NumPy library is imported with an alias of `np` so that any functionality within it can be used with convenience. We will be using this form of alias for all examples in this section.

10.1 NumPy Arrays

A Python list is a pretty powerful sequential data structure with some nifty features. For example, it can hold elements of various data types which can be added, changed or removed as required. Also, it allows index subsetting and traversal. But lists lack an important feature that is needed while performing data analysis tasks. We often want to carry out operations over an entire collection of elements, and we expect Python to perform this fast. With lists executing such operations over all elements efficiently is a problem. For example, let's consider a case where we calculate PCR (Put Call Ratio) for the previous 5 days. Say, we have put and call options volume (in Lacs) stored in lists `call_vol` and `put_vol` respectively. We then compute the PCR by dividing put volume by call volume as illustrated in the below script:

```
# Put volume in lacs
In []: put_vol = [52.89, 45.14, 63.84, 77.1, 74.6]

# Call volume in lacs
In []: call_vol = [49.51, 50.45, 59.11, 80.49, 65.11]

# Computing Put Call Ratio (PCR)
In []: put_vol / call_vol
Traceback (most recent call last):

File "<ipython-input-12>", line 1, in <module>
```

```
    put_vol / call_vol

TypeError: unsupported operand type(s) for /: 'list' and
'list'
```

Unfortunately, Python threw an error while calculating PCR values as it has no idea on how to do calculations on lists. We can do this by iterating over each item in lists and calculating the PCR for each day separately. However, doing so is inefficient and tiresome too. A way more elegant solution is to use NumPy arrays, an alternative to the regular Python list.

The NumPy array is pretty similar to the list, but has one useful feature: we can perform operations over entire arrays(all elements in arrays). It's easy as well as super fast. Let us start by creating a NumPy array. To do this, we use `array()` function from the NumPy package and create the NumPy version of `put_vol` and `call_vol` lists.

```
# Importing NumPy library
In []: import numpy as np

# Creating arrays
In []: n_put_vol = np.array(put_vol)

In []: n_call_vol = np.array(call_vol)

In []: n_put_vol
Out[]: array([52.89, 45.14, 63.84, 77.1 , 74.6 ])

In []: n_call_vol
Out[]: array([49.51, 50.45, 59.11, 80.49, 65.11])
```

Here, we have two arrays `n_put_vol` and `n_call_vol` which holds put and call volume respectively. Now, we can calculate PCR in one line:

```
# Computing Put Call Ratio (PCR)
In []: pcr = n_put_vol / n_call_vol

In []: pcr
Out[]: array([1.06826904, 0.89474727, 1.0800203,
              0.95788297, 1.14575334])
```

This time it worked, and calculations were performed element-wise. The first observation in `pcr` array was calculated by dividing the first element in `n_put_vol` by the first element in `n_call_vol` array. The second element in `pcr` was computed using the second element in the respective arrays and so on.

First, when we tried to compute PCR with regular lists, we got an error, because Python cannot do calculations with lists like we want it to. Then we converted these regular lists to NumPy arrays and the same operation worked without any problem. NumPy work with arrays as if they are scalars. But we need to pay attention here. NumPy can do this easily because it assumes that array can only contain values of a single type. It's either an array of integers, floats or booleans and so on. If we try to create an array of different types like the one mentioned below, the resulting NumPy array will contain a single type only. String in the below case:

```
In []: np.array([1, 'Python', True])
Out[]: array(['1', 'Python', 'True'], dtype='<U11')
```

NOTE: NumPy arrays are made to be created as homogeneous arrays, considering the mathematical operations that can be performed on them. It would not be possible with heterogeneous data sets.

In the example given above, an integer and a boolean were both converted to strings. NumPy array is a new type of data structure type like the Python list type that we have seen before. This also means that it comes with its own methods, which will behave differently from other types. Let us implement the + operation on the Python list and NumPy arrays and see how they differ.

```
# Creating lists
In []: list_1 = [1, 2, 3]

In []: list_2 = [5, 6, 4]

# Adding two lists
In []: list_1 + list_2
Out[]: [1, 2, 3, 5, 6, 4]
```

```
# Creating arrays
In []: arr_1 = np.array([1, 2, 3])

In []: arr_2 = np.array([5, 6, 4])

# Adding two arrays
In []: arr_1 + arr_2
Out[]: array([6, 8, 7])
```

As can be seen in the above example, performing the + operation with `list_1` and `list_2`, the list elements are pasted together, generating a list with 6 elements. On the other hand, if we do this with NumPy arrays, Python will do an element-wise sum of the arrays.

10.1.1 N-dimensional arrays

Until now we have worked with two arrays: `n_put_vol` and `n_call_vol`. If we are to check its type using `type()`, Python tells us that they are of type `numpy.ndarray` as shown below:

```
# Checking array type
In []: type(n_put_vol)
Out[]: numpy.ndarray
```

Based on the output we got, it can be inferred that they are of data type `ndarray` which stands for *n-dimensional array* within NumPy. These arrays are one-dimensional arrays, but NumPy also allows us to create two dimensional, three dimensional and so on. We will stick to two dimensional for our learning purpose in this module. We can create a 2D (two dimensional) NumPy array from a regular Python list of lists. Let us create one array for all put and call volumes.

```
# Recalling put and call volumes lists
In []: put_vol
Out[]: [52.89, 45.14, 63.84, 77.1, 74.6]

In []: call_vol
Out[]: [49.51, 50.45, 59.11, 80.49, 65.11]
```

```
# Creating a two-dimensional array
In []: n_2d = np.array([put_vol, call_vol])

In []: n_2d
Out[]:
array([[52.89, 45.14, 63.84, 77.1 , 74.6 ],
       [49.51, 50.45, 59.11, 80.49, 65.11]])
```

We see that `n_2d` array is a rectangular data structure. Each list provided in the `np.array` creation function corresponds to a row in the two-dimensional NumPy array. Also for 2D arrays, the NumPy rule applies: an array can only contain a single type. If we change one float value in the above array definition, all the array elements will be coerced to strings, to end up with a homogeneous array. We can think of a 2D array as an advanced version of lists of a list. We can perform element-wise operation with 2D as we had seen for a single dimensional array.

10.2 Array creation using built-in functions

An explicit input has been provided while creating `n_call_vol` and `n_put_vol` arrays. In contrast, NumPy provides various built-in functions to create arrays and input to them will be produced by NumPy. Below we discuss a handful of such functions:

- `zeros(shape, dtype=float)` returns an array of a given shape and type, filled with zeros. If the *dtype* is not provided as an input, the default type for the array would be float.

```
# Creating a one-dimensional array
In []: np.zeros(5)
Out[]: array([0., 0., 0., 0., 0.])

# Creating a two-dimensional array
In []: np.zeros((3, 5))
Out[]:
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
```

```
       [0., 0., 0., 0., 0.]])

# Creating a one-dimensional array of integer type
In []: np.zeros(5, dtype=int)
Out[]: array([0, 0, 0, 0, 0])
```

- `ones(shape, dtype=float)` returns an array of a given shape and type, filled with ones. If the *dtype* is not provided as an input, the default type for the array would be float.

```
# Creating a one-dimensional array
In []: np.ones(5)
Out[]: array([1., 1., 1., 1., 1.])

# Creating a one-dimensional array of integer type
In []: np.ones(5, dtype=int)
Out[]: array([1, 1, 1, 1, 1])
```

- `full(shape, fill_value, dtype=None)` returns an array of a given shape and type, fill with *fill_value* given in input parameters.

```
# Creating a one-dimensional array with value as 12
In []: np.full(5, 12)
Out[]: array([12, 12, 12, 12, 12])

# Creating a two-dimensional array with value as 9
In []: np.full((2, 3), 9)
Out[]:
array([[9, 9, 9],
       [9, 9, 9]])
```

- `arange([start, ]stop, [step])` returns an array with evenly spaced values within a given interval. Here the *start* and *step* parameters are optional. If they are provided NumPy will consider them while computing the output. Otherwise, range computation starts from 0. For all cases, stop value will be excluded in the output.

```
# Creating an array with only stop argument
In []: np.arange(5)
```

```
Out[]: array([0, 1, 2, 3, 4])

# Creating an array with start and stop arguments
In []: np.arange(3, 8)
Out[]: array([3, 4, 5, 6, 7])

# Creating an array with given interval and step value
# as 0.5
In []: np.arange(3, 8, 0.5)
Out[]: array([3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5, 7. ,
              7.5])
```

- `linspace(start, stop, num=50, endpoint=True)` returns evenly spaced numbers over a specified interval. The number of samples to be returned is specified by the *num* parameter. The endpoint of the interval can optionally be excluded.

```
# Creating an evenly spaced array with five numbers within
# interval 2 to 3
In []: np.linspace(2.0, 3.0, num=5)
Out[]: array([2.  , 2.25, 2.5 , 2.75, 3.  ])

# Creating an array excluding end value
In []: np.linspace(2.0, 3.0, num=5, endpoint=False)
Out[]: array([2. , 2.2, 2.4, 2.6, 2.8])

# Creating an array with ten values within the specified
# interval
In []: np.linspace(11, 20, num=10)
Out[]: array([11., 12., 13., 14., 15., 16., 17., 18., 19.,
              20.])
```

10.3 Random Sampling in NumPy

In addition to built-in functions discussed above, we have a `random` sub-module within the NumPy that provides handy functions to generate data randomly and draw samples from various distributions. Some of the widely used such functions are discussed here.

- `rand([d0, d1, ..., dn])` is used to create an array of a given shape and populate it with random samples from a *uniform distribution* over `[0, 1)`. It takes only positive arguments. If no argument is provided, a single float value is returned.

```
# Generating single random number
In []: np.random.rand()
Out[]: 0.1380210268817208

# Generating a one-dimensional array with four random
# values
In []: np.random.rand(4)
Out[]: array([0.24694323, 0.83698849, 0.0578015,
              0.42668907])

# Generating a two-dimensional array
In []: np.random.rand(2, 3)
Out[]:
array([[0.79364317, 0.15883039, 0.75798628],
       [0.82658529, 0.12216677, 0.78431111]])
```

- `randn([d0, d1, ..., dn])` is used to create an array of the given shape and populate it with random samples from a *standard normal* distributions. It takes only positive arguments and generates an array of shape `(d0, d1, ..., dn)` filled with random floats sampled from a univariate normal distribution of mean 0 and variance 1. If no argument is provided, a single float randomly sampled from the distribution is returned.

```
# Generating a random sample
In []: np.random.randn()
Out[]: 0.5569441449249491

# Generating a two-dimensional array over N(0, 1)
In []: np.random.randn(2, 3)
Out[]:
array([[ 0.43363995, -1.04734652, -0.29569917],
       [ 0.31077962, -0.49519421,  0.29426536]])
```

```
# Generating a two-dimensional array over N(3, 2.25)
In []: 1.5 * np.random.randn(2, 3) + 3
Out[]:
array([[1.75071139, 2.81267831, 1.08075029],
       [3.35670489, 3.96981281, 1.7714606 ]])
```

- `randint(low, high=None, size=None)` returns a random integer from a discrete uniform distribution with limits of *low* (inclusive) and *high* (exclusive). If *high* is None (the default), then results are from 0 to *low*. If the *size* is specified, it returns an array of the specified size.

```
# Generating a random integer between 0 and 6
In []: np.random.randint(6)
Out[]: 2

# Generating a random integer between 6 and 9
In []: np.random.randint(6, 9)
Out[]: 7

# Generating a one-dimensional array with values between 3
# and 9
In []: np.random.randint(3, 9, size=5)
Out[]: array([6, 7, 8, 8, 5])

# Generating a two-dimensional array with values between 3
# and 9
In []: np.random.randint(3, 9, size=(2, 5))
Out[]:
array([[5, 7, 4, 6, 4],
       [6, 8, 8, 5, 3]])
```

- `random(size=None)` returns a random float value between 0 and 1 which is drawn from the *continuous uniform* distribution.

```
# Generating a random float
In []: np.random.random()
Out[]: 0.6013749764953444

# Generating a one-dimensional array
```

```
In []: np.random.random(3)
Out[]: array([0.69929315, 0.61152299, 0.91313813])

# Generating a two-dimensional array
In []: np.random.random((3, 2))
Out[]:
array([[0.55779547, 0.6822698 ],
       [0.75476145, 0.224952  ],
       [0.99264158, 0.02755453]])
```

- `binomial(n, p, size=None)` returns samples drawn from a binomial distribution with n trials and p probability of success where n is greater than 0 and p is in the interval of 0 and 1.

```
# Number of trials, probability of each trial
In []: n, p = 1, .5

# Flipping a coin 1 time for 50 times
In []: samples = np.random.binomial(n, p, 50)

In []: samples
Out[]:
array([1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0,
       0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1,
       0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0])
```

- `normal(mu=0.0, sigma=1.0, size=None)` draws random samples from a normal (Gaussian) distribution. If no arguments provided, a sample will be drawn from N(0, 1).

```
# Initialize mu and sigma
In []: mu, sigma = 0, 0.1

# Drawing 5 samples in a one-dimensional array
In []: np.random.normal(mu, sigma, 5)
Out[]: array([ 0.06790522, 0.0092956, 0.07063545,
               0.28022021, -0.13597963])

# Drawing 10 samples in a two-dimensional array of
```

```
# shape (2, 5)
In []: np.random.normal(mu, sigma, (2, 5))
Out[]:
array([[-0.10696306, -0.0147926, -0.07027478, 0.04399432,
        -0.03861839],
       [-0.02004485, 0.08760261, 0.18348247, -0.09351321,
        -0.19487115]])
```

- `uniform(low=0.0, high=1.0, size=None)` draws samples from a uniform distribution over the interval 0 (including) and 1 (excluding), if no arguments are provided. In other words, any value drawn is equally likely within the interval.

```
# Creating a one-dimensional array with samples drawn
# within [-1, 0)
In []: np.random.uniform(-1, 0, 10)
Out[]:
array([-0.7910379, -0.64144624, -0.64691011, -0.03817127,
       -0.24480339, -0.82549031, -0.37500955, -0.88304322,
       -0.35196588, -0.51377252])

# Creating a two-dimensional array with samples drawn
# within [0, 1)
In []: np.random.uniform(size=(5, 2))
Out[]:
array([[0.43155784, 0.41360889],
       [0.81813931, 0.70115211],
       [0.40003811, 0.2114227 ],
       [0.95487774, 0.92251769],
       [0.91042434, 0.98697917]])
```

In addition to functions shown above, we can draw samples from various other distributions such as Poisson, Gamma, Exponential, etc. using NumPy.

10.4 Array Attributes and Methods

We now have some idea about the working of NumPy arrays. Let us now explore the functionalities provided by them. As with any Python object,

NumPy arrays also have a rich set of attributes and methods which simplifies the data analysis process to a great extent. Following are the most useful array attributes. For illustration purpose, we will be using previously defined arrays.

- `ndim` attribute displays the number of dimensions of an array. Using this attribute on `n_call_vol` and `pcr`, we expect dimensions to be 1 and 2 respectively. Let's check.

```
# Checking dimensions for n_call_vol array
In []: np_call_vol.ndim
Out[]: 1

In []: n_2d.ndim
Out[]: 2
```

- `shape` returns a tuple with the dimensions of the array. It may also be used to reshape the array in-place by assigning a tuple of array dimensions to it.

```
# Checking the shape of the one-dimensional array
In []: n_put_vol.shape
Out[]: (5,)

# Checking shape of the two-dimensional array
In []: n_2d.shape
Out[]: (2, 5) # It shows 2 rows and 5 columns

# Printing n_2d with 2 rows and 5 columns
In []: n_2d
Out[]:
array([[52.89, 45.14, 63.84, 77.1 , 74.6 ],
       [49.51, 50.45, 59.11, 80.49, 65.11]])

# Reshaping n_2d using the shape attribute
In []: n_2d.shape = (5, 2)

# Printing reshaped array
In []: n_2d
```

```
Out[]:
array([[52.89, 45.14],
       [63.84, 77.1 ],
       [74.6 , 49.51],
       [50.45, 59.11],
       [80.49, 65.11]])
```

- `size` returns the number of elements in the array.

```
In []: n_call_vol.size
Out[]: 5

In []: n_2d.size
Out[]: 10
```

- `dtype` returns the data-type of the array's elements. As we learned above, NumPy comes with its own data type just like regular built-in data types such as int, float, str, etc.

```
In []: n_put_vol.dtype
Out[]: dtype('float64')
```

A typical first step in analyzing a data is getting to the data in the first place. In an ideal data analysis process, we generally have thousands of numbers which need to be analyzed. Simply staring at these numbers won't provide us with any insights. Instead, what we can do is generate summary statistics of the data. Among many useful features, NumPy also provides various statistical functions which are good to perform such statistics on arrays.

Let us create a `samples` array and populate it with samples drawn from a normal distribution with a mean of 5 and standard deviation of 1.5 and compute various statistics on it.

```
# Creating a one-dimensional array with 1000 samples drawn
# from a normal distribution
In []: samples = np.random.normal(5, 1.5, 1000)

# Creating a two-dimensional array with 25 samples
```

```
# drawn from a normal distribution
In []: samples_2d = np.random.normal(5, 1.5, size=(5, 5))

In []: samples_2d
Out[]:
array([[5.30338102, 6.29371936, 2.74075451, 3.45505812,
        7.24391809],
       [5.20554917, 5.33264245, 6.08886915, 5.06753721,
        6.36235494],
       [5.86023616, 5.54254211, 5.38921487, 6.77609903,
        7.79595902],
       [5.81532883, 0.76402556, 5.01475416, 5.20297957,
        7.57517601],
       [5.76591337, 1.79107751, 5.03874984, 5.05631362,
        2.16099478]])
```

- `mean(a, axis=None)` returns the average of the array elements. The average is computed over the flattened array by default, otherwise over the specified axis.

- `average(a, axis=None)` returns the average of the array elements and works similar to that of `mean()`.

```
# Computing mean
In []: np.mean(samples)
Out[]: 5.009649198007546

In []: np.average(samples)
Out[]: 5.009649198007546

# Computing mean with axis=1 (over each row)
In []: np.mean(samples_2d, axis=1)
Out[]: array([5.00736622, 5.61139058, 6.27281024,
              4.87445283, 3.96260983])

In []: np.average(samples_2d, axis=1)
Out[]: array([5.00736622, 5.61139058, 6.27281024,
              4.87445283, 3.96260983])
```

- `max(a, axis=None)` returns the maximum of an array or maximum along an axis.

```
In []: np.max(samples)
Out[]: 9.626572532562523

In []: np.max(samples_2d, axis=1)
Out[]: array([7.24391809, 6.36235494, 7.79595902,
              7.57517601, 5.76591337])
```

- `median(a, axis=None)` returns the median along the specified axis.

```
In []: np.median(samples)
Out[]: 5.0074934668143865

In []: np.median(samples_2d)
Out[]: 5.332642448141249
```

- `min(a, axis=None)` returns the minimum of an array or minimum along an axis.

```
In []: np.min(samples)
Out[]: 0.1551821703754115

In []: np.min(samples_2d, axis=1)
Out[]: array([2.74075451, 5.06753721, 5.38921487,
              0.76402556, 1.79107751])
```

- `var(a, axis=None)` returns the variance of an array or along the specified axis.

```
In []: np.var(samples)
Out[]: 2.2967299389550466

In []: np.var(samples_2d)
Out[]: 2.93390175942658

# The variance is computed over each column of numbers
In []: np.var(samples_2d, axis=0)
Out[]: array([0.07693981, 4.95043105, 1.26742732,
              1.10560727, 4.37281009])
```

- `std(a, axis=None)` returns the standard deviation of an array or along the specified axis.

```
In [ ]: np.std(samples)
Out[ ]: 1.5154965981337756

In [ ]: np.std(samples_2d)
Out[ ]: 1.7128636137844075
```

- `sum(a, axis=None)` returns the sum of array elements.

```
# Recalling the array n_put_vol
In [ ]: n_put_vol
Out[ ]: array([52.89, 45.14, 63.84, 77.1 , 74.6 ])

# Computing sum of all elements within n_put_vol
In [ ]: np.sum(n_put_vol)
Out[ ]: 313.57

# Computing sum of all array over each row
In [ ]: np.sum(samples_2d, axis=1)
Out[ ]: array([25.03683109, 28.05695291, 31.36405118,
               24.37226413, 19.81304913])
```

- `cumsum(a, axis=None)` returns the cumulative sum of the elements along a given axis.

```
In [ ]: np.cumsum(n_put_vol)
Out[ ]: array([ 52.89,  98.03, 161.87, 238.97, 313.57])
```

The methods discussed above can also be directly called upon NumPy objects such as `samples`, `n_put_vol`, `samples_2d`, etc. instead of using the `np.` format as shown below. The output will be the same in both cases.

```
# Using np. format to compute the sum
In [ ]: np.sum(samples)
Out[ ]: 5009.649198007546

# Calling sum() directly on a NumPy object
In [ ]: samples.sum()
Out[ ]: 5009.649198007546
```

10.5 Array Manipulation

NumPy defines a new data type called `ndarray` for the array object it creates. This also means that various operators such as arithmetic operators, logical operator, boolean operators, etc. work in ways unique to it as we've seen so far. There's a flexible and useful array manipulation technique that NumPy provides to use on its data structure using *broadcasting*.

The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations (with certain constraints). The smaller array is *'broadcast'* across the larger array so that they have compatible shapes. It also provides a mean of vectorizing array operations.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape as in the following example.

```
In []: a = np.array([1, 2, 3])

In []: b = np.array([3, 3, 3])

In []: a * b
Out[]: array([3, 6, 9])
```

NumPy's broadcasting rule relaxes this constraint when the array's shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in operation as depicted below:

```
In []: a = np.array([1, 2, 3])

In []: b = 3

In []: a * b
Out[]: array([3, 6, 9])
```

The result is equivalent to the previous example where `b` was an array. We can think of the scalar `b` in the above example being *stretched* during the arithmetic operation into an array with the same shape as `a`. The new elements in `b` are simply copies of the original scalar. Here, the

stretching analogy is only conceptual. NumPy is smart enough to use the original scalar value without actually making copies so that broadcasting operations are as memory and computationally efficient as possible.

The code in the last example is more efficient because broadcasting moves less memory around during the multiplication than that of its counterpart defined above it. Along with efficient number processing capabilities, NumPy also provides various methods for array manipulation thereby proving versatility. We discuss some of them here.

- `exp(*args)` returns the exponential of all elements in the input array. The numbers will be raised to e also known as Euler's number.

```
# Computing exponentials for the array 'a'
In []: np.exp(a)
Out[]: array([ 2.71828183, 7.3890561, 20.08553692])
```

- `sqrt(*args)` returns the positive square-root of an array, element-wise.

```
# Computing square roots of a given array
In []: np.sqrt([1, 4, 9, 16, 25])
Out[]: array([1., 2., 3., 4., 5.])
```

- `reshape(new_shape)` gives a new shape to an array without changing its data.

```
# Creating a one-dimensional array with 12 elements
In []: res = np.arange(12)

In []: res
Out[]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

# Reshaping the 'res' array to 2-dimensional array
In []: np.reshape(res, (3, 4))
Out[]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
# Reshaping the dimensions from (3, 4) to (2, 6)
In []: np.reshape(res, (2, 6))
Out[]:
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

- `resize(a, new_shape)` return a new array with the specified shape. If the new array is larger than the original array, then the new array is filled with repeated copies of *a*.

```
# Creating a one-dimensional array
In []: demo = np.arange(4)

In []: demo
Out[]: array([0, 1, 2, 3])

# Resizing a 'demo' array to (2, 2)
In []: np.resize(demo, (2, 2))
Out[]:
array([[0, 1],
       [2, 3]])

# Resizing a 'demo' greater than its size.
In []: np.resize(demo, (4, 2))
Out[]:
array([[0, 1],
       [2, 3],
       [0, 1],
       [2, 3]])
```

- `round(a, decimals=0)` round an array to the given number of decimals. If decimals are not given, elements will be rounded to the whole number.

```
# Creating a one-dimensional array
In []: a = np.random.rand(5)

# Printing array
```

```
In []: a
Out[]: array([0.71056952, 0.58306487, 0.13270092,
              0.38583513, 0.7912277 ])

# Rounding to 0 decimals
In []: a.round()
Out[]: array([1., 1., 0., 0., 1.])

# Rounding to 0 decimals using the np.round syntax
In []: np.round(a)
Out[]: array([1., 1., 0., 0., 1.])

# Rounding to 2 decimals
In []: a.round(2)
Out[]: array([0.71, 0.58, 0.13, 0.39, 0.79])

# Rounding to 3 decimals using the np.round syntax
In []: np.round(a, 3)
Out[]: array([0.711, 0.583, 0.133, 0.386, 0.791])
```

- `sort(a, kind='quicksort')` returns a sorted copy of an array. The default sorting algorithm used is *quicksort*. Other available options are *mergesort* and *heapsort*.

```
In []: np.sort(n_put_vol)
Out[]: array([45.14, 52.89, 63.84, 74.6 , 77.1 ])

In []: np.sort(samples_2d)
Out[]:
array([[2.74075451, 3.45505812, 5.30338102, 6.29371936,
        7.24391809],
       [5.06753721, 5.20554917, 5.33264245, 6.08886915,
        6.36235494],
       [5.38921487, 5.54254211, 5.86023616, 6.77609903,
        7.79595902],
       [0.76402556, 5.01475416, 5.20297957, 5.81532883,
        7.57517601],
       [1.79107751, 2.16099478, 5.03874984, 5.05631362,
        5.76591337]])
```

- `vstack(tup)` stacks arrays provided via *tup* in sequence vertically (row wise).
- `hstack(tup)` stacks arrays provided via *tup* in sequence horizontally (column wise).
- `column_stack(tup)` stacks 1-dimensional arrays as column into a 2-dimensional array. It takes a sequence of 1-D arrays and stacks them as columns to make a single 2-D array.

```
# Creating sample arrays
In []: a = np.array([1, 2, 3])

In []: b = np.array([4, 5, 6])

In []: c = np.array([7, 8, 9])

# Stacking arrays vertically
In []: np.vstack((a, b, c))
Out[]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

# Stacking arrays horizontally
In []: np.hstack((a, b, c))
Out[]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Stacking two arrays together
In []: np.column_stack((a, b))
Out[]:
array([[1, 4],
       [2, 5],
       [3, 6]])

# Stacking three arrays together
In []: np.column_stack((a, b, c))
Out[]:
array([[1, 4, 7],
```

```
       [2, 5, 8],
       [3, 6, 9]])
```

- `transpose()` permutes the dimensions of an array.

```
# Creating a two-dimensional array with shape (2, 4)
In []: a = np.arange(8).reshape(2, 4)

# Printing it
In []: a
Out[]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])

# Transposing the array
In []: a.transpose()
Out[]:
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

10.6 Array Indexing and Iterating

NumPy is an excellent library for efficient number crunching along with ease of use. It seamlessly integrates with Python and its syntax. Following this attribute, NumPy provides subsetting and iterating techniques very similar to lists. We can use square brackets to subset NumPy arrays, Python built-in constructs to iterate, and other built-in methods to slice them.

10.6.1 Indexing and Subsetting

NumPy arrays follow indexing structure similar to Python lists. Index starts with 0 and each element in an array is associated with a unique index. Below table shows NumPy indexing for a one-dimensional array.

Index	**0**	**1**	**2**	**3**	**4**
np.array	52	88	41	63	94

The index structure for a two-dimensional array with a shape of (3, 3) is shown below.

Index	0	1	2
0	a	b	c
1	d	e	f
2	g	h	i

The two arrays `arr_1d` and `arr_2d` which depicts the above-shown structure have been created below:

```
# Creating one-dimensional array
In []: arr_1d = np.array([52, 88, 41, 63, 94])

# Creating two-dimensional array
In []: arr_2d = np.array([['a', 'b', 'c'],
  ...:                    ['d', 'e', 'f'],
  ...:                    ['g', 'h', 'i']])
```

We use square brackets `[]` to subset each element from NumPy arrays. Let us subset arrays created above using indexing.

```
# Slicing the element at index 0
In []: arr_1d[0]
Out[]: 52

# Slicing the last element using negative index
In []: arr_1d[-1]
Out[]: 94

# Slicing elements from position 2 (inclusive) to
# 5 (exclusive)
In []: arr_1d[2:5]
Out[]: array([41, 63, 94])
```

In the above examples, we sliced a one-dimensional array. Similarly, square brackets also allow slicing two-dimensional using the syntax `[r, c]` where *r* is a row and *c* is a column.

```
# Slicing the element at position (0, 1)
In []: arr_2d[0, 1]
Out[]: 'b'

# Slicing the element at position (1, 2)
In []: arr_2d[1, 2]
Out[]: 'f'

# Slicing the element at position (2, 0)
In []: arr_2d[2, 0]
Out[]: 'g'

# Slicing the first row
In []: arr_2d[0, ]
Out[]: array(['a', 'b', 'c'], dtype='<U1')

# Slicing the last column
In []: arr_2d[:, 2]
Out[]: array(['c', 'f', 'i'], dtype='<U1')
```

Notice the syntax in the last example where we slice the last column. The : has been provided as an input which denotes all elements and then filtering the last column. Using only : would return us all elements in the array.

```
In []: arr_2d[:]
Out[]:
array([['a', 'b', 'c'],
       ['d', 'e', 'f'],
       ['g', 'h', 'i']], dtype='<U1')
```

10.6.2 Boolean Indexing

NumPy arrays can be indexed with other arrays (or lists). The arrays used for indexing other arrays are known as *index arrays*. Mostly, it is a simple array which is used to subset other arrays. The use of index arrays ranges from simple, straightforward cases to complex and hard to understand cases. When an array is indexed using another array, a copy of the original data is returned, not a view as one gets for slices. To illustrate:

```
# Creating an array
In []: arr = np.arange(1, 10)

# Printing the array
In []: arr
Out[]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Subsetting the array `arr` using an anonymous array
In []: arr[np.array([2, 5, 5, 1])]
Out[]: array([3, 6, 6, 2])
```

We create an array `arr` with ten elements in the above example. Then we try to subset it using an anonymous index array. The index array consisting of the values 2, 5, 5 and 1 correspondingly create an array of length 4, i.e. same as the length index array. Values in the index array work as an index to subset (in the above-given operation) and it simply returns the corresponding values from the `arr`.

Extending this concept, an array can be indexed with itself. Using logical operators, NumPy arrays can be filtered as desired. Consider a scenario, where we need to filter array values which are greater than a certain threshold. This is shown below:

```
# Creating an random array with 20 values
In []: rand_arr = np.random.randint(1, 50, 20)

# Printing the array
In []: rand_arr
Out[]:
array([14, 25, 39, 18, 40, 10, 33, 36, 29, 25, 27,  4, 28,
       43, 43, 19, 30,29, 47, 41])

# Filtering the array values which are greater than 30
In []: rand_arr[rand_arr > 30]
Out[]: array([39, 40, 33, 36, 43, 43, 47, 41])
```

Here, we create an array with the name `rand_arr` with 20 random values. We then try to subset it with values which are greater than 30 using the logical operator >. When an array is being sliced using the logical operator,

NumPy generates an anonymous array of `True` and `False` values which is then used to subset the array. To illustrate this, let us execute the code used to subset the `rand_arr`, i.e. code written within the square brackets.

```
In []: filter_ = rand_arr > 30

In []: filter_
Out[]:
array([False, False,  True, False,  True, False,  True,
       True, False, False, False, False, False,  True,
       True, False, False, False, True,  True])
```

It returned a boolean array with only `True` and `False` values. Here, `True` appears wherever the logical condition holds true. NumPy uses this outputted array to subset the original array and returns only those values where it is `True`.

Apart from this approach, NumPy provides a `where` method using which we can achieve the same filtered output. We pass a logical condition within `where` condition, and it will return an array with all values for which conditions stands true. We filter out all values greater than 30 from the `rand_arr` using the `where` condition in the following example:

```
# Filtering an array using np.where method
In []: rand_arr[np.where(rand_arr > 30)]
Out[]: array([39, 40, 33, 36, 43, 43, 47, 41])
```

We got the same result by executing `rand_arr[rand_arr > 30]` and `rand_arr[np.where(rand_arr > 30)]`. However, the `where` method provided by NumPy just do not filter values. Instead, it can be used for more versatile scenarios. Below given is the official syntax:

```
np.where[condition[, x, y]]
```

It returns the elements, either from `x` or `y`, depending on `condition`. As these parameters, `x` and `y` are optional, `condition` when true yields `x` if given or boolean `True`, otherwise `y` or boolean `False`.

Below we create an array `heights` that contains 20 elements with height ranging from 150 to 160 and look at various uses of `where` method.

```
# Creating an array heights
In []: heights = np.random.uniform(150, 160, size=20)

In []: heights
Out[]:
array([153.69911134, 154.12173942, 150.35772942,
       151.53160722, 153.27900307, 154.42448961,
       153.25276742, 151.08520803, 154.13922276,
       159.71336708, 151.45302507, 155.01280829,
       156.9504274 , 154.40626961, 155.46637317,
       156.36825413,151.5096344 , 156.75707004,
       151.14597394, 153.03848597])
```

Usage 1: Without `x` and `y` parameters. Using the `where` method without the optional parameter as illustrated in the following example would return the index values of the original array where the condition is true.

```
In []: np.where(heights > 153)
Out[]:
(array([ 0, 1, 4, 5, 6, 8, 9, 11, 12, 13, 14, 15, 17, 19],
       dtype=int64),)
```

The above codes returned index values of the `heights` array where values are greater than 153. This scenario is very similar to the one we have seen above with the random array `rand_arr` where we tried to filter values above 30. Here, the output is merely the index values. If we want the original values, we need to subset the `heights` array using the output that we obtained.

Usage 2: With `x` as `True` and `y` as `False`. Having these optional parameters in place would return either of the parameters based on the condition. This is shown in the below example:

```
In []: np.where(heights > 153, True, False)
Out[]:
array([True, True, False, False, True, True, True, False,
       True, True, False, True, True, True, True, True,
       False, True,False, True])
```

The output in the *Usage 2* provides either `True` or `False` for all the elements in the `heights` array in contrast to the *Usage 1* where it returned index values of only those elements where the condition was true. The optional parameters can also be array like elements instead of scalars or static value such as `True` or `False`.

Usage 3: With `x` and `y` being arrays. Now that we have quite a good understanding of how the `where` method works, it is fairly easy to guess the output. The output will contain values from either `x` array or `y` array based on the condition in the first argument. For example:

```
# Creating an array 'x_array'
In []: x_array = np.arange(11, 31, 1)

In []: x_array
Out[]:
array([11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
       24, 25, 26, 27, 28, 29, 30])

# Creating an array 'y_array'
In []: y_array = np.arange(111, 131, 1)

In []: y_array
Out[]:
array([111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
       121, 122, 123, 124, 125, 126, 127, 128, 129, 130])

In []: np.where(heights > 153, x_array, y_array)
Out[]:
array([ 11,  12, 113, 114,  15,  16,  17, 118,  19,  20,
       121,  22,  23, 24,  25,  26, 127,  28, 129,  30])
```

As expected, the output of the above code snippet contains values from the array `x_array` when the value in the `heights` array is greater than 153, otherwise, the value from the `y_array` will be outputted.

Having understood the working of `where` method provided by the NumPy library, let us now see how it is useful in back-testing strategies. Consider a scenario where we have all the required data for generating trading signals.

Data that we have for this hypothetical example is the close price of a stock for 20 periods and its average price.

```
# Hypothetical close prices for 20 periods
In []: close_price = np.random.randint(132, 140, 20)

# Printing close_price
In []: close_price
Out[]:
array([137, 138, 133, 132, 134, 139, 132, 138, 137, 135,
       136, 134, 134, 139, 135, 133, 136, 139, 132, 134])
```

We are to generate trading signals based on the buy condition given to us. i.e. we go long or buy the stock when the closing price is greater than the average price of 135.45. It can be easily computed using the `where` method as shown below:

```
# Average close price
In []: avg_price = 135.45

# Computing trading signals with 1 being 'buy' and 0
# represents 'no signal'
In []: signals = np.where(close_price > avg_price, 1, 0)

# Printing signals
In []: signals
Out[]: array([1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0,
              0, 1, 1, 0, 0])
```

The `signals` array contains the trading signals where `1` represents the `buy` and `0` represents no trading signal.

10.6.3 Iterating Over Arrays

NumPy arrays are iterable objects in Python which means that we can directly iterate over them using the `iter()` and `next()` methods as with any other iterable. This also implies that we can use built-in looping constructs to iterate over them. The following examples show iterating NumPy arrays using a `for` loop.

```
# Looping over a one-dimensional array
In []: for element in arr_1d:
  ...:     print(element)

# Output
52
88
41
63
94
```

Looping over a one-dimensional array is easy and straight forward. But, if we are to execute the `for` loop with `arr_2d`, it will be traverse all rows and provide that as the output. It is demonstrated in the following example.

```
In []: for element in arr_2d:
  ...:     print(element)

# Output
['a' 'b' 'c']
['d' 'e' 'f']
['g' 'h' 'i']
```

To iterate over two-dimensional arrays, we need to take care of both axes. A separate `for` loop can be used in a nested format to traverse through each element as shown below:

```
In []: for element in arr_2d:
  ...:     for e in element:
  ...:         print(e)

# Output
a
b
c
d
e
f
g
```

```
h
i
```

The output that we got can also be achieved using `nditer()` method of NumPy, and it works for irrespective of dimensions.

```
In []: for element in np.nditer(arr_2d):
  ...:     print(element)

# Output
a
b
c
d
e
f
g
h
i
```

This brings us to the end of a journey with the NumPy module. The examples provided above depicts only a minimal set of NumPy functionalities. Though not comprehensive, it should give us a pretty good feel about what is NumPy and why we should be using it.

10.7 Key Takeaways

1. NumPy library is used to perform scientific computing in Python.
2. It is not a part of the Python Standard Library and needs to be installed explicitly before it can be used in a program.
3. It allows creating n-dimensional arrays of the type `ndarray`.
4. NumPy arrays can hold elements of single data type only.
5. They can be created using any sequential data structures such as lists or tuples, or using built-in NumPy functions.
6. The random module of the NumPy library allows generating samples from various data distributions.
7. NumPy supports for element-wise operation using broadcast functionality.

8. Similar to lists, NumPy arrays can also be sliced using square brackets `[]` and starts indexing with `0`.
9. It is also possible to slice NumPy arrays based on logical conditions. The resultant array would be an array of boolean True or False based on which other arrays are sliced or filtered. This is known as boolean indexing.

Chapter 11

Pandas Module

Pandas is a Python library to deal with sequential and tabular data. It includes many tools to manage, analyze and manipulate data in a convenient and efficient manner. We can think of its data structures as akin to database tables or spreadsheets.

Pandas is built on top of the Numpy library and has two primary data structures viz. Series (1-dimensional) and DataFrame (2- dimensional). It can handle both homogeneous and heterogeneous data, and some of its many capabilities are:

- ETL tools (Extraction, Transformation and Load tools)
- Dealing with missing data (NaN)
- Dealing with data files (csv, xls, db, hdf5, etc.)
- Time-series manipulation tools

In the Python ecosystem, Pandas is the best choice to retrieve, manipulate, analyze and transform financial data.

11.1 Pandas Installation

The official documentation[1] has a detailed explanation that spans over several pages on installing Pandas. We summarize it below.

[1] https://pandas.pydata.org/pandas-docs/stable/install.html

11.1.1 Installing with `pip`

The simplest way to install Pandas is from PyPI.
In a terminal window, run the following command.

```
pip install pandas
```

In your code, you can use the escape character '**!**' to install pandas directly from your Python console.

```
!pip install pandas
```

Pip is a useful tool to manage Python's packages and it is worth investing some time in knowing it better.

```
pip help
```

11.1.2 Installing with Conda environments

For advanced users, who like to work with Python environments for each project, you can create a new environment and install pandas as shown below.

```
conda create -n EPAT python
source activate EPAT
conda install pandas
```

11.1.3 Testing Pandas installation

To check the installation, Pandas comes with a test suite to test almost all of the codebase and verify that everything is working.

```
import pandas as pd
pd.test()
```

11.2 What problem does Pandas solve?

Pandas works with homogeneous data series (1-Dimension) and heterogeneous tabular data series (2-Dimensions). It includes a multitude of tools to work with these data types, such as:

- Indexes and labels.
- Searching of elements.
- Insertion, deletion and modification of elements.
- Apply set techniques, such as grouping, joining, selecting, etc.
- Data processing and cleaning.
- Work with time series.
- Make statistical calculations
- Draw graphics
- Connectors for multiple data file formats, such as, csv, xlsx, hdf5, etc.

11.3 Pandas Series

The first data structure in Pandas that we are going to see is the Series. They are homogeneous one-dimensional objects, that is, all data are of the same type and are implicitly labeled with an index.

For example, we can have a Series of integers, real numbers, characters, strings, dictionaries, etc. We can conveniently manipulate these series performing operations like adding, deleting, ordering, joining, filtering, vectorized operations, statistical analysis, plotting, etc.

Let's see some examples of how to create and manipulate a Pandas Series:

- We will start by creating an empty Pandas Series:

```
import pandas as pd
s = pd.Series()
print(s)

Out[]: Series([], dtype: float64)
```

- Let's create a Pandas Series of integers and print it:

```
import pandas as pd
s = pd.Series([1, 2, 3, 4, 5, 6, 7])
print(s)

Out[]:  0    1
```

```
1    2
2    3
3    4
4    5
5    6
6    7
dtype: int64
```

- Let's create a Pandas Series of characters:

```
import pandas as pd
s = pd.Series(['a', 'b', 'c', 'd', 'e'])
print(s)

Out[]:  0    1
        1    2
        2    3
        3    4
        4    5
        5    6
        6    7
        dtype: int64
```

- Let's create a random Pandas Series of float numbers:

```
import pandas as pd
import numpy as np
s = pd.Series(np.random.randn(5))
print(s)

Out[]:  0    0.383567
        1    0.869761
        2    1.100957
        3   -0.259689
        4    0.704537
        dtype: float64
```

In all these examples, we have allowed the index label to appear by default (without explicitly programming it). It starts at 0, and we can check the index as:

```
In []: s.index

Out[]: RangeIndex(start=0, stop=5, step=1)
```

But we can also specify the index we need, for example:

```
In []: s = pd.Series(np.random.randn(5),
                     index=['a', 'b', 'c', 'd', 'e'])

Out[]: a    1.392051
       b    0.515690
       c   -0.432243
       d   -0.803225
       e    0.832119
       dtype: float64
```

- Let's create a Pandas Series from a dictionary:

```
import pandas as pd
dictionary = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5}
s = pd.Series(dictionary)
print(s)

Out[]: a    1
       b    2
       c    3
       d    4
       e    5
       dtype: int64
```

In this case, the Pandas Series is created with the dictonary keys as index unless we specify any other index.

11.3.1 Simple operations with Pandas Series

When we have a Pandas Series, we can perform several simple operations on it. For example, let's create two Series. One from a dictionary and the other from an array of integers:

```
In []: import pandas as pd
       dictionary = {'a' : 1, 'b' : 2, 'c' : 3, 'd': 4,
                     'e': 5}
       s1 = pd.Series(dictionary)

       array = [1, 2, 3, 4, 5]
       s2 = pd.Series(array)

Out[]: a    1
       b    2
       c    3
       d    4
       e    5
       dtype: int64

       0    1
       1    2
       2    3
       3    4
       4    5
       dtype: int64
```

We can perform operations similar to Numpy arrays:

- Selecting one item from the Pandas Series by means of its index:

```
In []: s1[0] # Select the first element
Out[]: 1

In []: s1['a']
Out[]: 1

In []: s2[0]
Out[]: 1
```

- Selecting several items from the Pandas Series by means of its index:

```
In []: s1[[1, 4]]
Out[]: b    2
```

```
        e    5
        dtype: int64

In []: s1[['b', 'e']]
Out[]: b    2
        e    5
         dtype: int64

In []: s2[[1, 4]]
Out[]: b    2
        e    5
         dtype: int64
```

- Get the series starting from an element:

```
In []: s1[2:]
Out[]: c    3
        d    4
        e    5
        dtype: int64

In []: s2[2:]
Out[]: 2    3
        3    4
        4    5
        dtype: int64
```

- Get the series up to one element:

```
In []: s1[:2]
Out[]: c    3
        d    4
        e    5
        dtype: int64

In []: s2[:2]
Out[]: 2    3
        3    4
        4    5
        dtype: int64
```

We can perform operations like a dictionary:

- Assign a value:

```
In []: s1[1] = 99
       s1['a'] = 99

Out[]: a     1
       b    99
       c     3
       d     4
       e     5
       dtype: int64

In []: s2[1] = 99
       print(s2)

Out[]: 0     1
       1    99
       2     3
       3     4
       4     5
       dtype: int64
```

- Get a value by index (like dictionary key):

```
In []: s.get('b')
Out[]: 2
```

Here are some powerful vectorized operations that let us perform quickly calculations, for example:

- Add, subtract, multiply, divide, power, and almost any NumPy function that accepts NumPy arrays.

```
s1 + 2
s1 - 2
s1 * 2
s1 / 2
s1 ** 2
np.exp(s1)
```

- We can perform the same operations over two Pandas Series although these must be aligned, that is, to have the same index, in other case, perform a Union operation.

```
In []: s1 + s1 # The indices are aligned
Out[]: a     2
       b     4
       c     6
       d     8
       e    10
       dtype: int64

In []: s1 + s2 # The indices are unaligned
Out[]: a   NaN
       b   NaN
       c   NaN
       d   NaN
       e   NaN
       0   NaN
       1   NaN
       2   NaN
       3   NaN
       4   NaN
       dtype: float64
```

11.4 Pandas DataFrame

The second data structure in Pandas that we are going to see is the DataFrame.

Pandas DataFrame is a heterogeneous two-dimensional object, that is, the data are of the same type within each column but it could be a different data type for each column and are implicitly or explicitly labeled with an index.

We can think of a DataFrame as a database table, in which we store heterogeneous data. For example, a DataFrame with one column for the first name, another for the last name and a third column for the phone

number, or a dataframe with columns to store the opening price, close price, high, low, volume, and so on.

The index can be implicit, starting with zero or we can specify it ourselves, even working with dates and times as indexes as well. Let's see some examples of how to create and manipulate a Pandas DataFrame.

- Creating an empty DataFrame:

```
In []: import pandas as pd
       s = pd.DataFrame()
       print(s)

Out[]: Empty DataFrame
       Columns: []
       Index: []
```

- Creating an empty structure DataFrame:

```
In []: import pandas as pd
       s = pd.DataFrame(columns=['A', 'B', 'C', 'D', 'E'])
       print(s)

Out[]: Empty DataFrame
       Columns: [A, B, C, D, E]
       Index: []

In []: import pandas as pd
       s = pd.DataFrame(columns=['A', 'B', 'C', 'D', 'E'],
                        index=range(1, 6))
       print(s)

Out[]:      A    B    C    D    E
       1  NaN  NaN  NaN  NaN  NaN
       2  NaN  NaN  NaN  NaN  NaN
       3  NaN  NaN  NaN  NaN  NaN
       4  NaN  NaN  NaN  NaN  NaN
       5  NaN  NaN  NaN  NaN  NaN
```

- Creating a DataFrame passing a NumPy array:

```
In []: array =  {'A' : [1, 2, 3, 4],
                 'B' : [4, 3, 2, 1]}

       pd.DataFrame(array)

Out[]:   A   B
       0 1   4
       1 2   3
       2 3   2
       3 4   1
```

- Creating a DataFrame passing a NumPy array, with datetime index:

```
In []: import pandas as pd
       array = {'A':[1, 2, 3, 4], 'B':[4, 3, 2, 1]}
       index = pd.DatetimeIndex(['2018-12-01',
                                 '2018-12-02',
                                 '2018-12-03',
                                 '2018-12-04'])
       pd.DataFrame(array, index=index)

Out[]:              A   B
       2018-12-01   1   4
       2018-12-02   2   3
       2018-12-03   3   2
       2018-12-04   4   1
```

- Creating a DataFrame passing a Dictionary:

```
In []: import pandas as pd
       dictionary = {'a':1, 'b':2, 'c':3, 'd':4, 'e': 5}
       pd.DataFrame([dictionary])

Out[]:   a   b   c   d   e
       0 1   2   3   4   5
```

- Viewing a DataFrame: We can use some methods to explore the Pandas DataFrame:

First, we go to create a Pandas DataFrame to work with it.

```
In []: import pandas as pd
       pd.DataFrame({'A':np.random.randn(10),
                     'B':np.random.randn(10),
                     'C':np.random.randn(10)})
```

```
Out[]:      A           B           C
       0   0.164358    1.689183    1.745963
       1  -1.830385    0.035618    0.047832
       2   1.304339    2.236809    0.920484
       3   0.365616    1.877610   -0.287531
       4  -0.741372   -1.443922   -1.566839
       5  -0.119836   -1.249112   -0.134560
       6  -0.848425   -0.569149   -1.222911
       7  -1.172688    0.515443    1.492492
       8   0.765836    0.307303    0.788815
       9   0.761520   -0.409206    1.298350
```

- Get the first three rows:

```
In []: import pandas as pd
       df=pd.DataFrame({'A':np.random.randn(10),
                        'B':np.random.randn(10),
                        'C':np.random.randn(10)})
       df.head(3)
```

```
Out[]:      A           B           C
       0   0.164358    1.689183    1.745963
       1  -1.830385    0.035618    0.047832
       2   1.304339    2.236809    0.920484
```

- Get the last three rows:

```
In []: import pandas as pd
       df=pd.DataFrame({'A':np.random.randn(10),
                        'B':np.random.randn(10),
                        'C':np.random.randn(10)})
       df.tail(3)
```

```
Out[]:      A          B          C
        7  -1.172688   0.515443   1.492492
        8   0.765836   0.307303   0.788815
        9   0.761520  -0.409206   1.298350
```

- Get the DataFrame's index:

```
In []: import pandas as pd
       df=pd.DataFrame({'A':np.random.randn(10),
                        'B':np.random.randn(10),
                        'C':np.random.randn(10)})
       df.index

Out[]: RangeIndex(start=0, stop=10, step=1)
```

- Get the DataFrame's columns:

```
In []: import pandas as pd
       df=pd.DataFrame({'A':np.random.randn(10),
                        'B':np.random.randn(10),
                        'C':np.random.randn(10)})
       df.columns

Out[]: Index(['A', 'B', 'C'], dtype='object')
```

- Get the DataFrame's values:

```
In []: import pandas as pd
       df=pd.DataFrame({'A':np.random.randn(10),
                        'B':np.random.randn(10),
                        'C':np.random.randn(10)})
       df.values

Out[]: array([[ 0.6612966 , -0.60985049,  1.11955054],
              [-0.74105636,  1.42532491, -0.74883362],
              [ 0.10406892,  0.5511436 ,  2.63730671],
              [-0.73027121, -0.11088373, -0.19143175],
              [ 0.11676573,  0.27582786, -0.38271609],
              [ 0.51073858, -0.3313141 ,  0.20516165],
```

```
[ 0.23917755,  0.55362   , -0.62717194],
[ 0.25565784, -1.4960713 ,  0.58886377],
[ 1.20284041,  0.21173483,  2.0331718 ],
[ 0.62247283,  2.18407105,  0.02431867]])
```

11.5 Importing data in Pandas

Pandas DataFrame is able to read several data formats, some of the most used are: CSV, JSON, Excel, HDF5, SQL, etc.

11.5.1 Importing data from CSV file

One of the most useful functions is `read_csv` that allows us to read csv files with almost any format and load it into our DataFrame to work with it. Let's see how to work with csv files:

```
import pandas as pd
df=pd.read_csv('Filename.csv')
type(df)

Out[]: pandas.core.frame.DataFrame
```

This simple operation, loads the csv file into the Pandas DataFrame after which we can explore it as we have seen before.

11.5.2 Customizing pandas import

Sometimes the format of the csv file come with a particular separator or we need specific columns or rows. We will now see some ways to deal with this.

In this example, we want to load a csv file with blank space as separator:

```
import pandas as pd
df=pd.read_csv('Filename.csv', sep=' ')
```

In this example, we want to load columns from 0 and 5 and the first 100 rows:

```
import pandas as pd
df=pd.read_csv('Filename.csv', usecols=[0, 1, 2, 3, 4, 5],
               nrows=100)
```

It's possible to customize the headers, convert the columns or rows names and carry out a good number of other operations.

11.5.3 Importing data from Excel files

In the same way that we have worked with csv files, we can work with Excel file with the `read_excel` function, let's see some examples:
In this example, we want to load the sheet 1 from an Excel file:

```
import pandas as pd
df=pd.read_excel('Filename.xls', sheet_name='Sheet1')
```

This simple operation, loads the Sheet 1 from the Excel file into the Pandas DataFrame.

11.6 Indexing and Subsetting

Once we have the Pandas DataFrame prepared, independent of the source of our data (csv, Excel, hdf5, etc.) we can work with it, as if it were a database table, selecting the elements that interest us. We will work with some some examples on how to index and extract subsets of data.

Let's begin with loading a csv file having details of a market instrument.

```
In []: import pandas as pd
       df=pd.read_csv('MSFT.csv',
                      usecols=[0, 1, 2, 3, 4])
       df.head()
       df.shape

Out[]:     Date        Open    High    Low     Close
       0   2008-12-29  19.15   19.21   18.64   18.96
       1   2008-12-30  19.01   19.49   19.00   19.34
       2   2008-12-31  19.31   19.68   19.27   19.44
```

```
3   2009-01-02  19.53   20.40   19.37   20.33
4   2009-01-05  20.20   20.67   20.06   20.52

(1000, 5)
```

Here, we have read a csv file, of which we only need the columns of date, opening, closing, high and low (the first 5 columns) and we check the form of the DataFrame that has 1000 rows and 5 columns.

11.6.1 Selecting a single column

In the previous code, we have read directly the first 5 columns from the csv file. This is a filter that we applied, because we were only interested in those columns.

We can apply selection filters to the DataFrame itself, to select one column to work with. For example, we could need the `Close` column:

```
In []: close=df['Close']
       close.head()

Out[]:     Close
       0   18.96
       1   19.34
       2   19.44
       3   20.33
       4   20.52
```

11.6.2 Selecting multiple columns

We can select multiple columns too:

```
In []: closevol=df[['Close', 'Volume']]
       closevol.head()

Out[]:     Close    Volume
       0   18.96   58512800.0
       1   19.34   43224100.0
       2   19.44   46419000.0
```

```
3   20.33   50084000.0
4   20.52   61475200.0
```

11.6.3 Selecting rows via []

We can select a set of rows by index:

```
In []: import pandas as pd
       df=pd.read_csv('TSLA.csv' )
       df[100:110]
Out[]:
```

	Date	Open	...	AdjVolume	Name
100	2017-10-30	319.18	...	4236029.0	TSLA
101	2017-10-27	319.75	...	6942493.0	TSLA
102	2017-10-26	327.78	...	4980316.0	TSLA
103	2017-10-25	336.70	...	8547764.0	TSLA
104	2017-10-24	338.80	...	4463807.0	TSLA
105	2017-10-23	349.88	...	5715817.0	TSLA
106	2017-10-20	352.69	...	4888221.0	TSLA
107	2017-10-19	355.56	...	5032884.0	TSLA
108	2017-10-18	355.97	...	4898808.0	TSLA
109	2017-10-17	350.91	...	3280670.0	TSLA

Or we can select a set of rows and columns:

```
In []: df[100:110][['Close', 'Volume']]

Out[]:
        Close   Volume
100 320.08  4236029.0
101 320.87  6942493.0
102 326.17  4980316.0
103 325.84  8547764.0
104 337.34  4463807.0
105 337.02  5715817.0
106 345.10  4888221.0
```

```
107 351.81  5032884.0
108 359.65  4898808.0
109 355.75  3280670.0
```

11.6.4 Selecting via `.loc[]` (By label)

With `df.loc` we can do the same selections using labels:
To select a set of rows, we can code the following using the index number as label:

```
In []: df.loc[100:110]
Out[]:
```

	Date	Open	...	AdjVolume	Name
100	2017-10-30	319.18	...	4236029.0	TSLA
101	2017-10-27	319.75	...	6942493.0	TSLA
102	2017-10-26	327.78	...	4980316.0	TSLA
103	2017-10-25	336.70	...	8547764.0	TSLA
104	2017-10-24	338.80	...	4463807.0	TSLA
105	2017-10-23	349.88	...	5715817.0	TSLA
106	2017-10-20	352.69	...	4888221.0	TSLA
107	2017-10-19	355.56	...	5032884.0	TSLA
108	2017-10-18	355.97	...	4898808.0	TSLA
109	2017-10-17	350.91	...	3280670.0	TSLA

Or we can select a set of rows and columns like before:

```
In []: df.loc[100:110, ['Close', 'Volume']]
Out[]:
        Close   Volume
100 320.08  4236029.0
101 320.87  6942493.0
102 326.17  4980316.0
103 325.84  8547764.0
104 337.34  4463807.0
105 337.02  5715817.0
106 345.10  4888221.0
```

```
107 351.81  5032884.0
108 359.65  4898808.0
109 355.75  3280670.0
110 350.60  5353262.0
```

11.6.5 Selecting via `.iloc[]` (By position)

With `df.iloc` we can do the same selections using integer position:

```
In []: df.iloc[100:110]
Out[]:
```

	Date	Open	...	AdjVolume	Name
100	2017-10-30	319.18	...	4236029.0	TSLA
101	2017-10-27	319.75	...	6942493.0	TSLA
102	2017-10-26	327.78	...	4980316.0	TSLA
103	2017-10-25	336.70	...	8547764.0	TSLA
104	2017-10-24	338.80	...	4463807.0	TSLA
105	2017-10-23	349.88	...	5715817.0	TSLA
106	2017-10-20	352.69	...	4888221.0	TSLA
107	2017-10-19	355.56	...	5032884.0	TSLA
108	2017-10-18	355.97	...	4898808.0	TSLA
109	2017-10-17	350.91	...	3280670.0	TSLA

In the last example, we used the index as an integer position rather than by label.
We can select a set of rows and columns like before:

```
In []: df.iloc[100:110, [3, 4]]
Out[]:
           Low     Close
100 317.25  320.08
101 316.66  320.87
102 323.20  326.17
103 323.56  325.84
104 336.16  337.34
105 336.25  337.02
```

```
106 344.34  345.10
107 348.20  351.81
108 354.13  359.65
109 350.07  355.75
```

11.6.6 Boolean indexing

So far, we have sliced subsets of data by label or by position. Now let's see how to select data that meet some criteria. We do this with Boolean indexing. We can use the same criteria similar to what we have seen with Numpy arrays. We show you just two illustrative examples here. This is by no means enough to get comfortable with it and so would encourage you to check the documentation and further readings at the end of this chapter to learn more.

- We can filter data that is greater (less) than a number.

```
In []: df[df.Close > 110]
Out[]:
```

	Date	Open	...	Close	...	Name
0	2017-03-27	304.00	...	279.18	...	TSLA
1	2017-03-26	307.34	...	304.18	...	TSLA
2	2017-03-23	311.25	...	301.54	...	TSLA
	...	...	...	...	...	...
1080	2017-12-09	137.00	...	141.60	...	TSLA
1081	2017-12-06	141.51	...	137.36	...	TSLA
1082	2013-12-05	140.51	...	140.48	...	TSLA

```
1083 rows × 14 columns

In []: df[(df['Close'] > 110) | (df['Close'] < 120)]
Out[]:
```

	Date	Open	...	Close	...	Name
0	2017-03-27	304.00	...	279.18	...	TSLA

	Date	Open	...	Close	...	Name
1	2017-03-26	307.34	...	304.18	...	TSLA
2	2017-03-23	311.25	...	301.54	...	TSLA
	...	...	...	...	...	...
1080	2017-12-09	137.00	...	141.60	...	TSLA
1081	2017-12-06	141.51	...	137.36	...	TSLA
1082	2013-12-05	140.51	...	140.48	...	TSLA

```
1083 rows × 14 columns
```

11.7 Manipulating a DataFrame

When we are working with data, the most common structure is the DataFrame. Until now we have seen how to create them, make selections and find data. We are now going to see how to manipulate the DataFrame to transform it into another DataFrame that has the form that our problem requires.

We'll see how to sort it, re-index it, eliminate unwanted (or spurious) data, add or remove columns and update values.

11.7.1 Transpose using `.T`

The Pandas DataFrame transpose function `T` allows us to transpose the rows as columns, and logically the columns as rows:

```
In []: import pandas as pd
       df=pd.read_csv('TSLA.csv' )
       df2=df[100:110][['Close', 'Volume']]

       df2.T
Out[]:
```

	100	101	102	...	109
Close	320.08	320.87	326.17	...	355.75

	100	101	102	...	109
Volume	4236029.00	6942493.00	4980316.00	...	3280670.00

11.7.2 The `.sort_index()` method

When we are working with Pandas Dataframe it is usual to add or remove rows, order by columns, etc. That's why it's important to have a function that allows us to easily and comfortably sort the DataFrame by its index. We do this with the `sort_index` function of Pandas DataFrame.

```
In []: df.sort_index()
Out[]:
```

	Date	Open	High	Low	Close	...	Name
0	2017-03-27	304.00	304.27	277.18	279.18	...	TSLA
1	2017-03-26	307.34	307.59	291.36	304.18	...	TSLA
2	2017-03-23	311.25	311.61	300.45	301.54	...	TSLA
	...	...	...	...	...	...	...

11.7.3 The `.sort_values()` method

Sometimes, we may be interested in sorting the DataFrame by some column or even with several columns as criteria. For example, sort the column by first names and the second criterion by last names. We do this with the `sort_values` function of Pandas DataFrame.

```
In []: df.sort_values(by='Close')
Out[]:
```

	Date	Open	High	Low	Close	...	Name
1081	2013-12-06	141.51	142.49	136.30	137.36	...	TSLA
1057	2014-01-13	145.78	147.00	137.82	139.34	...	TSLA
1078	2013-12-11	141.88	143.05	139.49	139.65	...	TSLA
	...	...	...	...	...	...	...

```
In []: df.sort_values(by=['Open', 'Close'])
Out[]:
```

	Date	Open	High	Low	Close	...	Name
1080	2013-12-09	137.00	141.70	134.21	141.60	...	TSLA
1077	2014-12-12	139.70	148.24	138.53	147.47	...	TSLA
1079	2013-12-10	140.05	145.87	139.86	142.19	...	TSLA
	...	...	...	...	...	...	...

11.7.4 The `.reindex()` function

The Pandas' reindex funtion let us to realign the index of the Series or DataFrame, it's useful when we need to reorganize the index to meet some criteria. For example, we can play with the Series or DataFrame that we create before to alter the original index. For example, when the index is a label, we can reorganize as we need:

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5),
                       index=['a', 'b', 'c', 'd', 'e'])
       df

Out[]:
```

	0
a	-0.134133
b	-0.586051
c	1.179358
d	0.433142
e	-0.365686

Now, we can reorganize the index as follows:

```
In []: df.reindex(['b', 'a', 'd', 'c', 'e'])
Out[]:
```

	0
b	-0.586051
a	-0.134133
d	0.433142
c	1.179358
e	-0.365686

When the index is numeric we can use the same function to order by hand the index:

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5))
       df.reindex([4,3,2,1,0])
Out[]:
```

	0
4	1.058589
3	1.194400
2	-0.645806
1	0.836606
0	1.288102

Later in this section, we'll see how to work and reorganize date and time indices.

11.7.5 Adding a new column

Another interesting feature of DataFrames is the possibility of adding new columns to an existing DataFrame.
For example, we can add a new column to the random DataFrame that we have created before:

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5))
Out[]:
```

	0
0	0.238304
1	2.068558
2	1.015650
3	0.506208
4	0.214760

To add a new column, we only need to include the new column name in the DataFrame and assign a initialization value, or assign to the new column a Pandas Series or another column from other DataFrame.

```
In []: df['new']=1
       df
Out[]:
```

	0	new
0	0.238304	1
1	2.068558	1
2	1.015650	1
3	0.506208	1
4	0.214760	1

11.7.6 Delete an existing column

Likewise, we can remove one or more columns from the DataFrame. Let's create a DataFrame with 5 rows and 4 columns with random values to delete one column.

```
In []: import pandas as pd
       import numpy as np
       df = pd.DataFrame(np.random.randn(5, 4))
       df
Out[]:
```

	0	1	2	3
0	-1.171562	-0.086348	-1.971855	1.168017
1	-0.408317	-0.061397	-0.542212	-1.412755
2	-0.365539	-0.587147	1.494690	1.756105
3	0.642882	0.924202	0.517975	-0.914366
4	0.777869	-0.431151	-0.401093	0.145646

Now, we can delete the column that we specify by index or by label if any:

```
In [ ]: del df[0]
Out[ ]:
```

	1	2	3
0	-0.086348	-1.971855	1.168017
1	-0.061397	-0.542212	-1.412755
2	-0.587147	1.494690	1.756105
3	0.924202	0.517975	-0.914366
4	-0.431151	-0.401093	0.145646

```
In [ ]: df['new']=1
        df
Out[ ]:
```

	0	new
0	0.238304	1
1	2.068558	1
2	1.015650	1
3	0.506208	1

```
In [ ]: del df['new']
Out[ ]:
```

	0
0	0.238304
1	2.068558
2	1.015650
3	0.506208

11.7.7 The `.at[]` (By label)

With `at` we can to locate a specific value by row and column labels as follows:

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5,4),
                       index=['a', 'b', 'c', 'd', 'e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	0.996496	-0.165002	0.727912	0.564858
b	-0.388169	1.171039	-0.231934	-1.124595
c	-1.385129	0.299195	0.573570	-1.736860
d	1.222447	-0.312667	0.957139	-0.054156
e	1.188335	0.679921	1.508681	-0.677776

```
In []: df.at['a', 'A']
Out[]: 0.9964957014209125
```

It is possible to assign a new value with the same funcion too:

```
In []: df.at['a', 'A'] = 0
Out[]:
```

	A	B	C	D
a	0.000000	-0.165002	0.727912	0.564858
b	-0.388169	1.171039	-0.231934	-1.124595
c	-1.385129	0.299195	0.573570	-1.736860
d	1.222447	-0.312667	0.957139	-0.054156
e	1.188335	0.679921	1.508681	-0.677776

11.7.8 The `.iat[]` (By position)

With `iat` we can to locate a specific value by row and column index as follow:

```
In []: import pandas as pd
              import numpy as np
       df=pd.DataFrame(np.random.randn(5, 4),
                       index=['a', 'b', 'c', 'd', 'e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)
       df.iat[0, 0]

Out[]: 0.996496
```

It is possible to assign a new value with the same funcion too:

```
In []: df.iat[0, 0] = 0
Out[]:
```

	A	B	C	D
a	0.000000	-0.165002	0.727912	0.564858
b	-0.388169	1.171039	-0.231934	-1.124595
c	-1.385129	0.299195	0.573570	-1.736860
d	1.222447	-0.312667	0.957139	-0.054156
e	1.188335	0.679921	1.508681	-0.677776

11.7.9 Conditional updating of values

Another useful function is to update values that meet some criteria, for example, update values whose values are greater than 0:

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5, 4),
                       index=['a','b','c','d','e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)

       df[df > 0] = 1
       df

Out[]:
```

	A	B	C	D
a	1.000000	-0.082466	1.000000	-0.728372
b	-0.784404	-0.663096	-0.595112	1.000000
c	-1.460702	-1.072931	-0.761314	1.000000
d	1.000000	1.000000	1.000000	-0.302310
e	-0.488556	1.000000	-0.798716	-0.590920

We can also update the values of a specific column that meet some criteria, or even work with several columns as criteria and update a specific column.

```
In []: df['A'][df['A'] < 0] = 1
       print(df)

Out[]:
```

	A	B	C	D
a	1.0	-0.082466	1.000000	-0.728372
b	1.0	-0.663096	-0.595112	1.000000
c	1.0	-1.072931	-0.761314	1.000000

	A	B	C	D
d	1.0	1.000000	1.000000	-0.302310
e	1.0	1.000000	-0.798716	-0.590920

```
In []: df['A'][(df['B'] < 0 ) & (df['C'] < 0)] = 9
       print(df)
Out[]:
```

	A	B	C	D
a	1.0	-0.082466	1.000000	-0.728372
b	9.0	-0.663096	-0.595112	1.000000
c	9.0	-1.072931	-0.761314	1.000000
d	1.0	1.000000	1.000000	-0.302310
e	1.0	1.000000	-0.798716	-0.590920

11.7.10 The `.dropna()` method

Occasionally, we may have a DataFrame that, for whatever reason, includes NA values. This type of values is usually problematic when we are making calculations or operations and must be treated properly before proceeding with them. The easiest way to eliminate NA values is to remove the row that contains it.

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5, 4),
                       index=['a','b','c','d','e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)

Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332

	A	B	C	D
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	0.312319	-0.765930	-1.641234	-1.388924

```
In []: df['A'][(df['B'] < 0 ) & (df['C'] < 0)] = np.nan
       print(df)
```

```
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	NaN	-0.765930	-1.641234	-1.388924

```
In []: df=df.dropna()
        print(df)
```

```
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177

Here we are deleting the whole row that has, in any of its columns, a NaN value, but we can also specify that it deletes the column that any of its values is NaN:

```
df=df.dropna(axis=1)
print(df)
```

We can specify if a single NaN value is enough to delete the row or column, or if the whole row or column must have NaN to delete it.
`pythonpython df=df.dropna(how='all') print(df) "'`

11.7.11 The `.fillna()` method

With the previous function we have seen how to eliminate a complete row or column that contains one or all the values to NaN, this operation can be a little drastic if we have valid values in the row or column.
For this, it is interesting to use the `fillna` function that substitutes the NaN values with some fixed value.

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5, 4),
                       index=['a','b','c','d','e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	0.312319	-0.765930	-1.641234	-1.388924

```
In []: df['A'][(df['B'] < 0 ) & (df['C'] < 0)] = np.nan
       print(df)
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177

	A	B	C	D
e	NaN	-0.765930	-1.641234	-1.388924

```
In []: df=df.fillna(999)
       print(df)
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	999	-0.765930	-1.641234	-1.388924

11.7.12 The `.apply()` method

The `apply` is a very useful way to use functions or methods in a DataFrame without having to loop through it. We can apply the `apply` method to a Series or DataFrame to apply a function to all rows or columns of the DataFrame. Let's see some examples.

Suppose we are working with the randomly generated DataFrame and need to apply a function. In this example, for simplicity's sake, we're going to create a custom function to square a number.

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5, 4),
                       index=['a','b','c','d','e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742

	A	B	C	D
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
def square_number(number):
  return number**2

# Test the function
In []: square_number(2)
Out[]: 4
```

Now, let's use the custom function through Apply:

```
In []: df.apply(square_number, axis=1)
Out[]:
```

	A	B	C	D
a	0.401005	7.285074	0.329536	0.426073
b	0.003636	0.022658	0.022238	0.491704
c	0.002758	0.220412	0.808524	0.370161
d	1.830372	0.010671	0.209652	3.599253
e	0.007793	0.174989	1.216586	0.339254

This method `apply` the funcion `square_number` to all rows of the DataFrame.

11.7.13 The `.shift()` function

The `shift` function allows us to move a row to the right or left and/or to move a column up or down. Let's look at some examples.

First, we are going to move the values of a column downwards:

```
In []: import pandas as pd
       import numpy as np
```

```
df=pd.DataFrame(np.random.randn(5, 4),
                index=['a','b','c','d','e'],
                columns=['A', 'B', 'C', 'D'])
print(df)
```

Out[]:

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df['D'].shift(1)
```

Out[]:

	A	B	C	D
a	-0.633249	-2.699088	0.574052	NaN
b	0.060295	-0.150527	0.149123	0.652742
c	-0.052515	0.469481	0.899180	-0.701216
d	-1.352912	0.103302	0.457878	-0.608409
e	0.088279	0.418317	-1.102989	-1.897170

We are going to move the values of a column upwards

```
In []: df['shift'] = df['D'].shift(-1)
```

Out[]:

	A	B	C	D
a	-0.633249	-2.699088	0.574052	-0.701216
b	0.060295	-0.150527	0.149123	-0.608409
c	-0.052515	0.469481	0.899180	-1.897170
d	-1.352912	0.103302	0.457878	0.582455
e	0.088279	0.418317	-1.102989	NaN

This is very useful for comparing the current value with the previous value.

11.8 Statistical Exploratory data analysis

Pandas DataFrame allows us to make some descriptive statistics calculations, which are very useful to make a first analysis of the data we are handling. Let's see some useful functions.

11.8.1 The `info()` function

It is a good practice to know the structure and format of our DataFrame, the `Info` function offers us just that:

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5, 4),
                       index=['a','b','c','d','e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df.info()

Out[]: <class 'pandas.core.frame.DataFrame'>
       Index: 5 entries, a to e
       Data columns (total 5 columns):
       A        5 non-null float64
       B        5 non-null float64
       C        5 non-null float64
```

```
D         5 non-null float64
shift     4 non-null float64
dtypes: float64(5)
memory usage: 240.0+ bytes
```

11.8.2 The `describe()` function

We can obtain a statistical overview of the DataFrame with the 'describe function, which gives us the mean, median, standard deviation, maximum, minimum, quartiles, etc. of each DataFrame column.

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5, 4),
                       index=['a','b','c','d','e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df.describe()
Out[]:
```

	A	B	C	D
count	5.000000	5.000000	5.000000	5.000000
mean	-0.378020	-0.371703	0.195449	-0.394319
std	0.618681	1.325046	0.773876	1.054633
min	-1.352912	-2.699088	-1.102989	-1.897170
25%	-0.633249	-0.150527	0.149123	-0.701216
50%	-0.052515	0.103302	0.457878	-0.608409

	A	B	C	D
75%	0.060295	0.418317	0.574052	0.582455
max	0.088279	0.469481	0.899180	0.652742

11.8.3 The `value_counts()` function

The function `value_counts` counts the repeated values of the specified column:

```
In []: df['A'].value_counts()
Out[]: 0.088279    1
      -0.052515    1
       0.060295    1
      -0.633249    1
      -1.352912    1
      Name: A, dtype: int64
```

11.8.4 The `mean()` function

We can obtain the mean of a specific column or row by means of the `mean` function.

```
In []: df['A'].mean() # Specifying a column
Out[]: -0.3780203497252693

In []: df.mean() # By column
       df.mean(axis=0) # By column

Out[]:  A       -0.378020
        B       -0.371703
        C        0.195449
        D       -0.394319
        shift   -0.638513
        dtype: float64

In []: df.mean(axis=1) # By row

Out[]:  a   -0.526386
```

```
b    0.002084
c    0.001304
d   -0.659462
e   -0.382222
dtype: float64
```

11.8.5 The `std()` function

We can obtain the standard deviation of a specific column or row by means of the `std` function.

```
In []: df['A'].std() # Specifying a column
Out[]: 0.6186812554819784

In []: df.std() # By column
       df.std(axis=0) # By column

Out[]: A        0.618681
       B        1.325046
       C        0.773876
       D        1.054633
       shift    1.041857
       dtype: float64

In []: df.std(axis=1) # By row

Out[]: a    1.563475
       b    0.491499
       c    0.688032
       d    0.980517
       e    1.073244
       dtype: float64
```

11.9 Filtering Pandas DataFrame

We have already seen how to filter data in a DataFrame, including logical statements to filter rows or columns with some logical criteria. For example, we will filter rows whose column 'A' is greater than zero:

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5, 4),
                       index=['a','b','c','d','e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df_filtered = df[df['A'] > 0]
       print(df_filtered)
Out[]:
```

	A	B	C	D
b	0.060295	-0.150527	0.149123	-0.701216
e	0.088279	0.418317	-1.102989	0.582455

We can also combine logical statements, we will filter all rows whose column 'A' and 'B' have their values greater than zero.

```
In []: df_filtered = df[(df['A'] > 0) & (df['B'] > 0)]
       print(df_filtered)
Out[]:
```

	A	B	C	D
e	0.088279	0.418317	-1.102989	0.582455

11.10 Iterating Pandas DataFrame

We can go through the DataFrame row by row to do operations in each iteration, let's see some examples.

```
In []: for item in df.iterrows():
           print(item)
Out[]:

('a', A       -0.633249
B       -2.699088
C        0.574052
D        0.652742
shift         NaN
Name: a, dtype: float64)
('b', A        0.060295
B       -0.150527
C        0.149123
D       -0.701216
shift    0.652742
Name: b, dtype: float64)
('c', A       -0.052515
B        0.469481
C        0.899180
D       -0.608409
shift   -0.701216
Name: c, dtype: float64)
('d', A       -1.352912
B        0.103302
C        0.457878
D       -1.897170
shift   -0.608409
Name: d, dtype: float64)
('e', A        0.088279
B        0.418317
C       -1.102989
D        0.582455
shift   -1.897170
Name: e, dtype: float64)
```

11.11 Merge, Append and Concat Pandas DataFrame

Another interesting feature of DataFrames is that we can merge, concatenate them and add new values, let's see how to do each of these operations.

- `merge` function allows us to merge two DataFrame by rows:

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5, 4),
                       index=['a','b','c','d','e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334

```
In []: import pandas as pd
       import numpy as np
       df=pd.DataFrame(np.random.randn(5, 4),
                       index=['a','b','c','d','e'],
                       columns=['A', 'B', 'C', 'D'])
       print(df)
Out[]:
```

	A	B	C	D
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

```
In []: df3 = pd.merge(df1, df2)
       print(df3)

Out[]: Empty DataFrame
       Columns: [A, B, C, D]
       Index: []
```

- `append` function allows us to append rows from one DataFrame to another DataFrame by rows:

```
In []: import pandas as pd
       import numpy as np
       df1=pd.DataFrame(np.random.randn(5, 4),
                        index=['a','b','c','d','e'],
                        columns=['A', 'B', 'C', 'D'])

In []: df2=pd.DataFrame(np.random.randn(5, 4),
                        index=['a','b','c','d','e'],
                        columns=['A', 'B', 'C', 'D'])

In []: df3 = df1.append(df2)
       print(df3)
Out[]:
```

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

- `concat` function allows us to merge two DataFrame by rows or

columns:

```
In []: import pandas as pd
       import numpy as np
       df1=pd.DataFrame(np.random.randn(5, 4),
                        index=['a','b','c','d','e'],
                        columns=['A', 'B', 'C', 'D'])

In []: df2=pd.DataFrame(np.random.randn(5, 4),
                        index=['a','b','c','d','e'],
                        columns=['A', 'B', 'C', 'D'])

In []: df3 = pd.concat([df1, df2]) # Concat by row
       print(df3)
Out[]:
```

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

```
In []: df3 = pd.concat([df1, df2], axis=0) # Concat by row
       print(df3)
Out[]:
```

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751

	A	B	C	D
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

```
# Concat by column
In []: df3 = pd.concat([df1, df2], axis=1)
       print(df3)
Out[]:
```

	A	B	...	D	A	...	D
a	1.179924	-1.512124	...	0.019265	2.030462	...	-0.757323
b	0.019969	-1.351649	...	-0.989025	0.475807	...	-0.843963
c	0.351921	-0.792914	...	0.170751	0.948164	...	1.319999
d	-0.150499	0.151942	...	-0.347300	1.433736	...	-1.051454
e	-1.307590	0.185759	...	-0.170334	0.565345	...	-0.227519

11.12 TimeSeries in Pandas

Pandas TimeSeries includes a set of tools to work with Series or DataFrames indexed in time. Usually, the series of financial data are of this type and therefore, knowing these tools will make our work much more comfortable. We are going to start creating time series from scratch and then we will see how to manipulate them and convert them to different frequencies.

11.12.1 Indexing Pandas TimeSeries

With `date_range` Panda's method, we can create a time range with a certain frequency. For example, create a range starting in December 1st, 2018, with 30 occurrences with an hourly frequency.

```
In []: rng = pd.date_range('12/1/2018', periods=30,
                          freq='H')
       print(rng)

Out[]: DatetimeIndex([
       '2018-12-01 00:00:00', '2018-12-01 01:00:00',
       '2018-12-01 02:00:00', '2018-12-01 03:00:00',
       '2018-12-01 04:00:00', '2018-12-01 05:00:00',
       '2018-12-01 06:00:00', '2018-12-01 07:00:00',
       '2018-12-01 08:00:00', '2018-12-01 09:00:00',
       '2018-12-01 10:00:00', '2018-12-01 11:00:00',
       '2018-12-01 12:00:00', '2018-12-01 13:00:00',
       '2018-12-01 14:00:00', '2018-12-01 15:00:00',
       '2018-12-01 16:00:00', '2018-12-01 17:00:00',
       '2018-12-01 18:00:00', '2018-12-01 19:00:00',
       '2018-12-01 20:00:00', '2018-12-01 21:00:00',
       '2018-12-01 22:00:00', '2018-12-01 23:00:00',
       '2018-12-02 00:00:00', '2018-12-02 01:00:00',
       '2018-12-02 02:00:00', '2018-12-02 03:00:00',
       '2018-12-02 04:00:00', '2018-12-02 05:00:00'],
       dtype='datetime64[ns]', freq='H')
```

We can do the same to get a daily frequency[2] (or any other, as per our requirement). We can use the `freq` parameter to adjust this.

```
In []: rng = pd.date_range('12/1/2018', periods=10,
                          freq='D')
       print(rng)

Out[]: DatetimeIndex(['2018-12-01', '2018-12-02',
        '2018-12-03', '2018-12-04', '2018-12-05',
        '2018-12-06', '2018-12-07', '2018-12-08',
        '2018-12-09', '2018-12-10'],
        dtype='datetime64[ns]', freq='D')
```

Now, we have a `DateTimeIndex` in the `rng` object and we can use it to create a Series or DataFrame:

[2] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandasrange.html

```
In []: ts = pd.DataFrame(np.random.randn(len(rng), 4),
       index=rng, columns=['A', 'B', 'C', 'D'])
       print(ts)
Out[]:
```

	A	B	C	D
2018-12-01	0.048603	0.968522	0.408213	0.921774
2018-12-02	-2.301373	-2.310408	-0.559381	-0.652291
2018-12-03	-2.337844	0.329954	0.289221	0.259132
2018-12-04	1.357521	0.969808	1.341875	0.767797
2018-12-05	-1.212355	-0.077457	-0.529564	0.375572
2018-12-06	-0.673065	0.527754	0.006344	-0.533316
2018-12-07	0.226145	0.235027	0.945678	-1.766167
2018-12-08	1.735185	-0.604229	0.274809	0.841128

```
In []: ts = pd.Series(np.random.randn(len(rng)), index=rng)
       print(ts)

Out[]: 2018-12-01    0.349234
       2018-12-02   -1.807753
       2018-12-03    0.112777
       2018-12-04    0.421516
       2018-12-05   -0.992449
       2018-12-06    1.254999
       2018-12-07   -0.311152
       2018-12-08    0.331584
       2018-12-09    0.196904
       2018-12-10   -1.619186
       2018-12-11    0.478510
       2018-12-12   -1.036074
```

Sometimes, we read the data from internet sources or from csv files and we need to convert the date column into the index to work properly with the Series or DataFrame.

```
In []: import pandas as pd
       df=pd.read_csv('TSLA.csv')
```

```
        df.tail()
Out[]:
```

	Date	Open	High	Low	Close	...	Name
1078	2013-12-11	141.88	143.05	139.49	139.65	...	TSLA
1079	2013-12-10	140.05	145.87	139.86	142.19	...	TSLA
1080	2013-12-09	137.00	141.70	134.21	141.60	...	TSLA
1081	2013-12-06	141.51	142.49	136.30	137.36	...	TSLA
1082	2013-12-05	140.15	143.35	139.50	140.48	...	TSLA

Here, we can see the index as numeric and a Date column, let's convert this column into the index to indexing our DataFrame, read from a csv file, in time. For this, we are going to use the Pandas `set_index` method

```
In []: df = df.set_index('Date')
       df.tail()
Out[]:
```

Date	Open	High	Low	Close	...	Name
2013-12-11	141.88	143.05	139.49	139.65	...	TSLA
2013-12-10	140.05	145.87	139.86	142.19	...	TSLA
2013-12-09	137.00	141.70	134.21	141.60	...	TSLA
2013-12-06	141.51	142.49	136.30	137.36	...	TSLA
2013-12-05	140.15	143.35	139.50	140.48	...	TSLA

Now, we have Pandas TimeSeries ready to work.

11.12.2 Resampling Pandas TimeSeries

A very useful feature of Pandas TimeSeries is the resample capacity, this allows us to pass the current frequency to another higher frequency (we can't pass to lower frequencies, because we don't know the data).

As it can be supposed, when we pass from one frequency to another data could be lost, for this, we must use some function that treat the values of

each frequency interval, for example, if we pass from an hourly frequency to daily, we must specify what we want to do with the group of data that fall inside each frequency, we can do a mean, a sum, we can get the maximum or the minimum, etc.

```
In []: rng = pd.date_range('12/1/2018', periods=30,
              freq='H')
       ts = pd.DataFrame(np.random.randn(len(rng), 4),
            index=rng, columns=['A', 'B', 'C', 'D'])
       print(ts)
Out[]:
```

	A	B	C	D
2018-12-01 00:00:00	0.048603	0.968522	0.408213	0.921774
2018-12-01 01:00:00	-2.301373	-2.310408	-0.559381	-0.652291
2018-12-01 02:00:00	-2.337844	0.329954	0.289221	0.259132
2018-12-01 03:00:00	1.357521	0.969808	1.341875	0.767797
2018-12-01 04:00:00	-1.212355	-0.077457	-0.529564	0.375572
2018-12-01 05:00:00	-0.673065	0.527754	0.006344	-0.533316

```
In []: ts = ts.resample("1D").mean()
       print(ts)
Out[]:
```

	A	B	C	D
2018-12-01	0.449050	0.127412	-0.154179	-0.358324
2018-12-02	-0.539007	-0.855894	0.000010	0.454623

11.12.3 Manipulating TimeSeries

We can manipulate the Pandas TimeSeries in the same way that we have done until now, since they offer us the same capacity that the Pandas Series and the Pandas DataFrames. Additionally, we can work comfortably with all jobs related to handling dates. For example, to obtain all the data from a date, to obtain the data in a range of dates, etc.

```
In []: rng = pd.date_range('12/1/2018', periods=30,
             freq='D')
       ts = pd.DataFrame(np.random.randn(len(rng), 4),
            index=rng, columns=['A', 'B', 'C', 'D'])
       print(ts)
Out[]:
```

	A	B	C	D
2018-12-01	0.048603	0.968522	0.408213	0.921774
2018-12-02	-2.301373	-2.310408	-0.559381	-0.652291
2018-12-03	-2.337844	0.329954	0.289221	0.259132
2018-12-04	1.357521	0.969808	1.341875	0.767797
2018-12-05	-1.212355	-0.077457	-0.529564	0.375572
2018-12-06	-0.673065	0.527754	0.006344	-0.533316

Getting all values from a specific date:

```
In []: ts['2018-12-15':]
Out[]:
```

	A	B	C	D
2018-12-02	0.324689	-0.413723	0.019163	0.385233
2018-12-03	-2.198937	0.536600	-0.540934	-0.603858
2018-12-04	-1.195148	2.191311	-0.981604	-0.942440
2018-12-05	0.621298	-1.435266	-0.761886	-1.787730
2018-12-06	0.635679	0.683265	0.351140	-1.451903

Getting all values inside a date range:

```
In []: ts['2018-12-15':'2018-12-20']
Out[]:
```

	A	B	C	D
2018-12-15	0.605576	0.584369	-1.520749	-0.242630

	A	B	C	D
2018-12-16	-0.105561	-0.092124	0.385085	0.918222
2018-12-17	0.337416	-1.367549	0.738320	2.413522
2018-12-18	-0.011610	-0.339228	-0.218382	-0.070349
2018-12-19	0.027808	-0.422975	-0.622777	0.730926
2018-12-20	0.188822	-1.016637	0.470874	0.674052

11.13 Key Takeaways

1. Pandas DataFrame and Pandas Series are some of the most important data structures. It is a must to acquire fluency in its handling because we will find them in practically all the problems that we handle.
2. A DataFrame is a data structure formed by rows and columns and has an index.
3. We must think of them as if they were data tables (for the Array with a single column) with which we can select, filter, sort, add and delete elements, either by rows or columns.
4. Help in ETL processes (Extraction, Transformation and Loading)
5. We can select, insert, delete and update elements with simple functions.
6. We can perform computations by rows or columns.
7. Has the ability to run vectorized computations.
8. We can work with several DataFrames at the same time.
9. Indexing and subsetting are the most important features from Pandas.
10. Facilitates the statistical exploration of the data.
11. It offers us a variety of options for handling NaN data.
12. Another additional advantage is the ability to read & write multiple data formats (CSV, Excel, HDF5, etc.).
13. Retrieve data from external sources (Yahoo, Google, Quandl, etc.)
14. Finally, it has the ability to work with date and time indexes and offers us a set of functions to work with dates.

Chapter 12

Data Visualization with Matplotlib

Matplotlib is a popular Python library that can be used to create data visualizations quite easily. It is probably the single most used Python package for 2D-graphics along with limited support for 3D-graphics. It provides both, a very quick way to visualize data from Python and publication-quality figures in many formats. Also, It was designed from the beginning to serve two purposes:

1. Allow for interactive, cross-platform control of figures and plots
2. Make it easy to produce static vector graphics files without the need for any GUIs.

Much like Python itself, Matplotlib gives the developer complete control over the appearance of their plots. It tries to make easy things easy and hard things possible. We can generate plots, histograms, power spectra, bar charts, error charts, scatter plots, etc. with just a few lines of code. For simple plotting, the `pyplot` module within `matplotlib` package provides a MATLAB-like interface to the underlying object-oriented plotting library. It implicitly and automatically creates figures and axes to achieve the desired plot.

To get started with Matplotlib, we first *import* the package. It is a common practice to import `matplotlib.pyplot` using the alias as `plt`. The `pyplot` being the sub-package within Matplotlib provides the common charting

functionality. Also, if we are working in a Jupyter Notebook, the line `%matplotlib inline` becomes important, as it makes sure that the plots are embedded inside the notebook. This is demonstrated in the example below:

```
import matplotlib.pyplot as plt

%matplotlib inline
```

NOTE: `Matplotlib` does not fall under the Python Standard Library and hence, like any other third party library, it needs to be installed before it can be used. It can be installed using the command `pip install matplotlib`.

12.1 Basic Concepts

Matplotlib allows creating a wide variety of plots and graphs. It is a humongous project and can seem daunting at first. However, we will break it down into bite-sized components and so learning it should be easier.
Different sources use 'plot' to mean different things. So let us begin by defining specific terminology used across the domain.

- `Figure` is the top-level container in the hierarchy. It is the overall window where everything is drawn. We can have multiple independent figures, and each figure can have multiple `Axes`. It can be created using the `figure` method of `pyplot` module.
- `Axes` is where the plotting occurs. The axes are effectively the area that we plot data on. Each `Axes` has an `X-Axis` and a `Y-Axis`.

The below mentioned example illustrates the use of the above-mentioned terms:

```
fig = plt.figure()
<Figure size 432x288 with 0 Axes>
```

Upon running the above example, nothing happens really. It only creates a figure of size 432 x 288 with 0 Axes. Also, Matplotlib will not show anything until told to do so. Python will wait for a call to `show` method to display the

plot. This is because we might want to add some extra features to the plot before displaying it, such as title and label customization. Hence, we need to call `plt.show()` method to show the figure as shown below:

```
plt.show()
```

As there is nothing to plot, there will be no output. While we are on the topic, we can control the size of the figure through the `figsize` argument, which expects a tuple of (width, height) in inches.

```
fig = plt.figure(figsize=(8, 4))
<Figure size 576x288 with 0 Axes>
plt.show()
```

12.1.1 Axes

All plotting happens with respect to an `Axes`. An Axes is made up of `Axis` objects and many other things. An *Axes* object must belong to a `Figure`. Most commands that we will ever issue will be with respect to this *Axes* object. Typically, we will set up a `Figure`, and then add `Axes` on to it. We can use `fig.add_axes` but in most cases, we find that adding a subplot fits our need perfectly. A subplot is an axes on a grid system.

- `add_subplot` method adds an `Axes` to the figure as part of a subplot arrangement.

```
# -Example 1-
# Creating figure
fig = plt.figure()

# Creating subplot
# Sub plot with 1 row and 1 column at the index 1
ax = fig.add_subplot(111)
plt.show()
```

The above code adds a single plot to the figure `fig` with the help of `add_subplot()` method. The output we get is a blank plot with axes ranging from 0 to 1 as shown in *figure 1*.

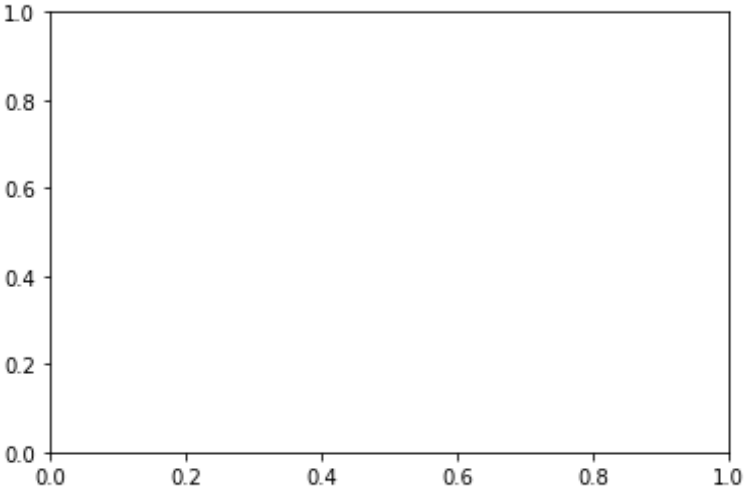

Figure 1: Empty plot added on axes

We can customize the plot using a few more built-in methods. Let us add the title, X-axis label, Y-axis label, and set limit range on both axes. This is illustrated in the below code snippet.

```
# -Example 2-
fig = plt.figure()

# Creating subplot/axes
ax = fig.add_subplot(111)

# Setting axes/plot title
ax.set_title('An Axes Title')

# Setting X-axis and Y-axis limits
ax.set_xlim([0.5, 4.5])
ax.set_ylim([-3, 7])

# Setting X-axis and Y-axis labels
ax.set_ylabel('Y-Axis Label')
ax.set_xlabel('X-Axis Label')
```

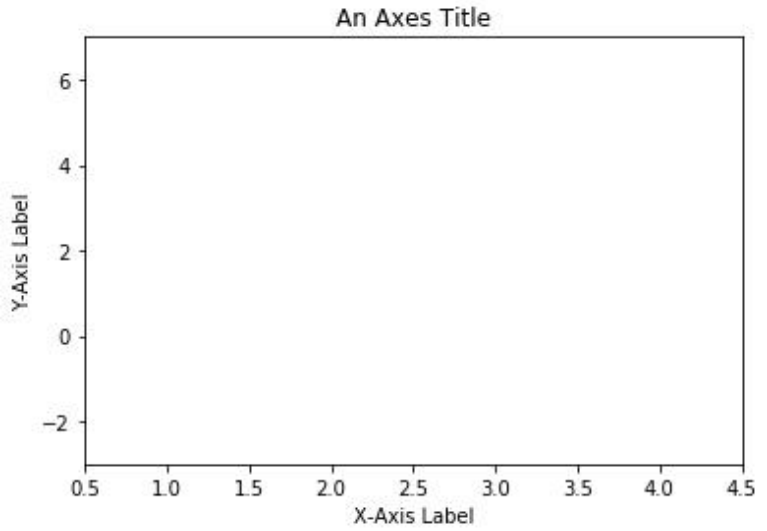

Figure 2: An empty plot with title, labels and custom axis limits

```
# Showing the plot
plt.show()
```

The output of the above code is shown in *figure 2*. Matplotlib's objects typically have lots of *explicit setters*, i.e. methods that start with `set_<something>` and control a particular option. Setting each option using explicit setters becomes repetitive, and hence we can set all required parameters directly on the axes using the `set` method as illustrated below:

```
# -Example 2 using the set method-
fig = plt.figure()

# Creating subplot/axes
ax = fig.add_subplot(111)

# Setting title and axes properties
ax.set(title='An Axes Title', xlim=[0.5, 4.5],
       ylim=[-3, 7], ylabel='Y-Axis Label',
       xlabel='X-Axis Label')

plt.show()
```

NOTE: The `set` method does not just apply to `Axes`; it applies to more-or-less all matplotlib objects.

The above code snippet gives the same output as *figure 2*. Using the `set` method when all required parameters are passed as arguments.

12.1.2 Axes method v/s `pyplot`

Interestingly, almost all methods of axes objects exist as a method in the `pyplot` module. For example, we can call `plt.xlabel('X-Axis Label')` to set label of X-axis (`plt` being an alias for `pyplot`), which in turn calls `ax.set_xlabel('X-Axis Label')` on whichever axes is *current*.

```
# -Example 3-
# Creating subplots, setting title and axes labels
# using `pyplot`
plt.subplots()
plt.title('Plot using pyplot')
plt.xlabel('X-Axis Label')
plt.ylabel('Y-Axis Label')
plt.show()
```

The code above is more intuitive and has fewer variables to construct a plot. The output for the same is shown in *figure 3*. It uses implicit calls to axes method for plotting. However, if we take a look at *"The Zen of Python"* (try `import this`), it says:

```
"Explicit is better than implicit."
```

While very simple plots, with short scripts, would benefit from the conciseness of the `pyplot` implicit approach, when doing more complicated plots, or working within larger scripts, we will want to explicitly pass around the axes and/or figure object to operate upon. We will be using both approaches here wherever it deems appropriate.

Anytime we see something like below:

```
fig = plt.figure()
ax = fig.add_subplot(111)
```

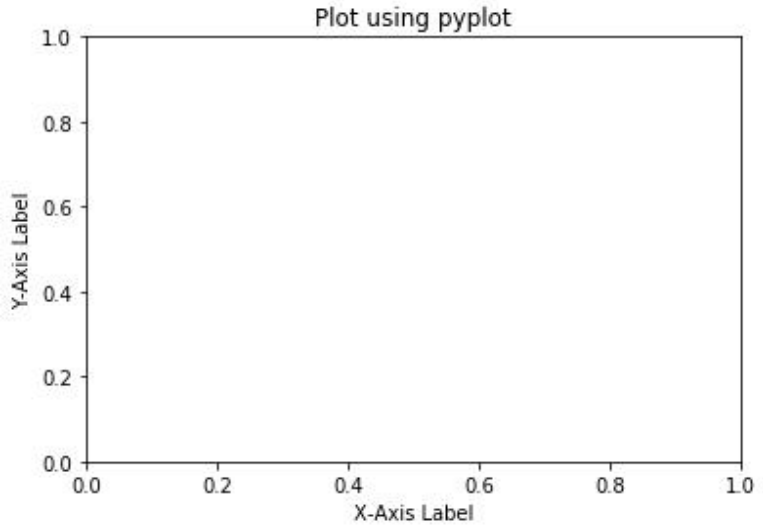

Figure 3: An empty plot using `pyplot`

can almost always be replaced with the following code:

```
fig, ax = plt.subplots()
```

Both versions of code produce the same output. However, the latter version is cleaner.

12.1.3 Multiple Axes

A figure can have more than one `Axes` on it. The easiest way is to use `plt.subplots()` call to create a figure and add the axes to it automatically. Axes will be on a regular grid system. For example,

```
# -Example 4-
# Creating subplots with 2 rows and 2 columns
fig, axes = plt.subplots(nrows=2, ncols=2)
plt.show()
```

Upon running the above code, Matplotlib would generate a figure with four subplots arranged with two rows and two columns as shown in *figure 4*.

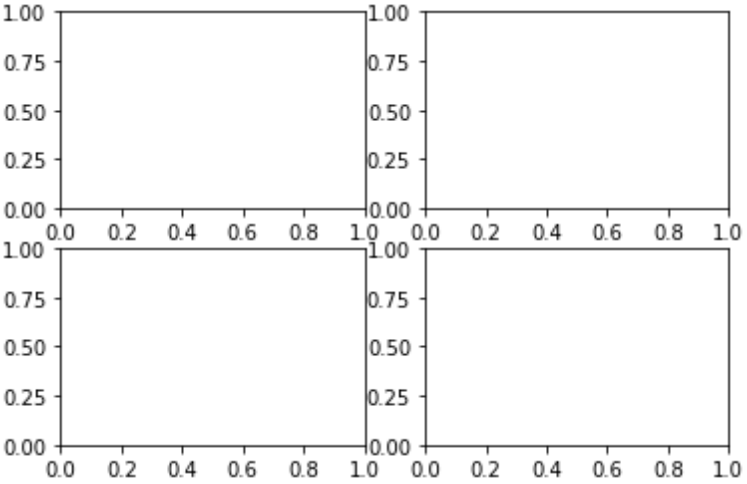

Figure 4: Figure with multiple axes

The `axes` object that was returned here would be a 2D-NumPy array, and each item in the array is one of the subplots. Therefore, when we want to work with one of these axes, we can index it and use that item's methods. Let us add the title to each subplot using the axes methods.

```
# -Example 5-
# Create a figure with four subplots and shared axes
fig, axes = plt.subplots(nrows=2, ncols=2, sharex=True,
sharey=True)
axes[0, 0].set(title='Upper Left')
axes[0, 1].set(title='Upper Right')
axes[1, 0].set(title='Lower Left')
axes[1, 1].set(title='Lower Right')
plt.show()
```

The above code generates a figure with four subplots and shared X and Y axes. Axes are shared among subplots in row wise and column-wise manner. We then set a title to each subplot using the `set` method for each subplot. Subplots are arranged in a clockwise fashion with each subplot having a unique index. The output is shown in *figure 5*.

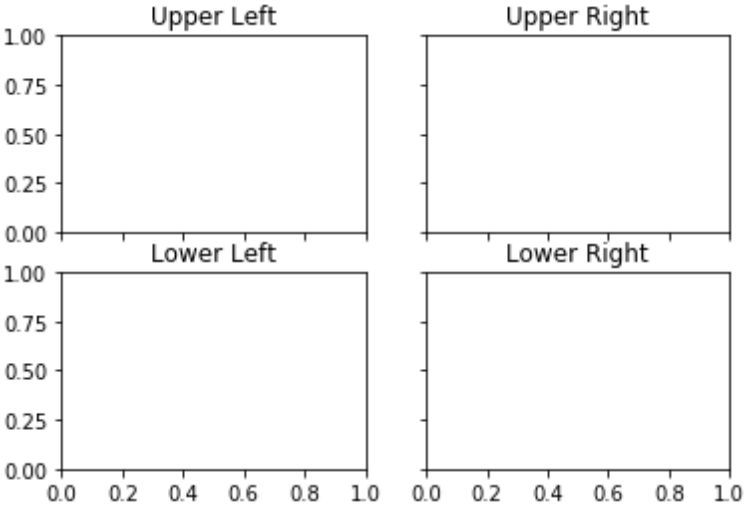

Figure 5: Subplots with the share axes

12.2 Plotting

We have discussed a lot about laying things out, but we haven't really discussed anything about plotting data yet. Matplotlib has various plotting functions. Many more than we will discuss and cover here. However, a full list or gallery[1] can be a bit overwhelming at first. Hence, we will condense it down and attempt to start with simpler plotting and than move towards more complex plotting. The `plot` method of `pyplot` is one of the most widely used methods in Matplotlib to plot the data. The syntax to call the `plot` method is shown below:

```
plot([x], y, [fmt], data=None, **kwargs)
```

The coordinates of the points or line nodes are given by x and y. The optional parameter *fmt* is a convenient way of defining basic formatting like color, market, and style. The `plot` method is used to plot almost any kind of data in Python. It tells Python what to plot and how to plot it, and also allows customization of the plot being generated such as color, type, etc.

[1] https://matplotlib.org/gallery/index.html

12.2.1 Line Plot

A line plot can be plotted using the `plot` method. It plots Y versus X as lines and/or markers. Below we discuss a few scenarios for plotting line. To plot a line, we provide coordinates to be plotted along X and Y axes separately as shown in the below code snippet.

```
# -Example 6-
# Defining coordinates to be plotted on X and Y axes
# respectively
x = [1.3, 2.9, 3.1, 4.7, 5.6, 6.5, 7.4, 8.8, 9.2, 10]
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]

# Plot lists 'x' and 'y'
plt.plot(x, y)

# Plot axes labels and show the plot
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.show()
```

The above code plots values in the list `x` along the X-axis and values in the list `y` along the Y-axis as shown in *figure 6*.

The call to `plot` takes minimal arguments possible, i.e. values for Y-axis only. In such a case, Matplotlib will implicitly consider the index of elements in list `y` as the input to the X-axis as demonstrated in the below example:

```
# -Example 7-
# Defining 'y' coordinates
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]

# Plot list 'y'
plt.plot(y)

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in List Y')
plt.show()
```

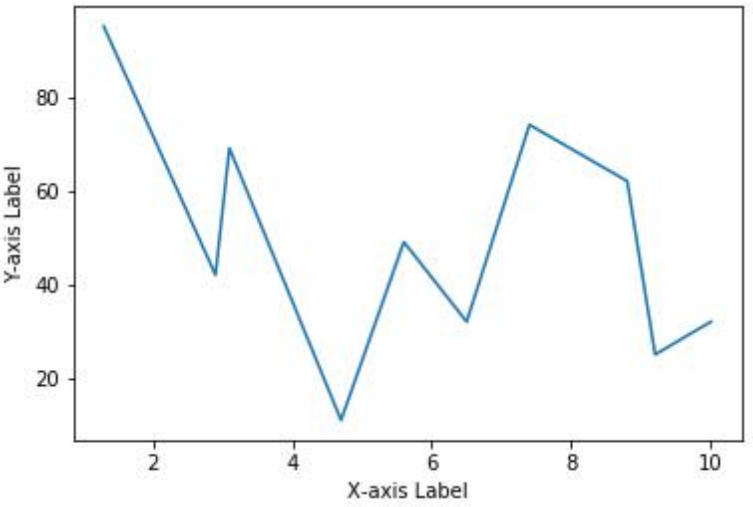

Figure 6: Line plot with `x` *and* `y` *as data*

Here, we define a list called `y` that contains values to be plotted on Y-axis. The output for the same is shown in *figure 7*.

The plots created uses *the default* line style and color. The optional parameter *fmt* in the `plot` method is a convenient way for defining basic formatting like color, marker, and line-style. It is a shortcut string notation consisting of color, marker, and line:

```
fmt = '[color][marker][line]'
```

Each of them is optional. If not provided, the value from the style cycle[2] is used. We use this notation in the below example to change the line color:

```
# -Example 8-
# Plot line with green color
plt.plot(y, 'g')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
```

[2]https://matplotlib.org/tutorials/intermediate/color_cycle.html

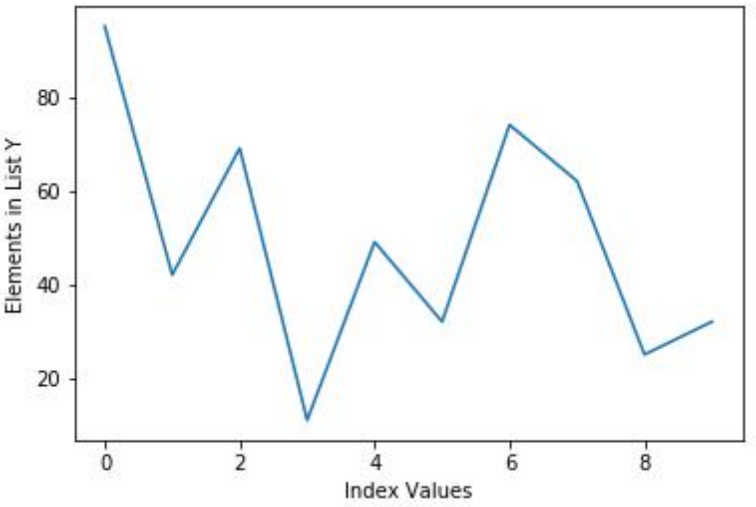

Figure 7: Line plot with `y` as the only data

```
plt.ylabel('Elements in List Y')
plt.show()
```

Following the *fmt* string notation, we changed the color of a line to green using the character `g` which refers to the line color. This generates the plot with green line as shown in *figure 8*. Likewise, markers are added using the same notation as shown below:

```
# -Example 9-
# Plot continuous green line with circle markers
plt.plot(y, 'go-')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in List Y')
plt.show()
```

Here, the *fmt* parameters: `g` refers to the green color, `o` refers to circle markers and `-` refers to a continuous line to be plotted as shown in *figure 9*. This formatting technique allows us to format a line plot in virtually any way

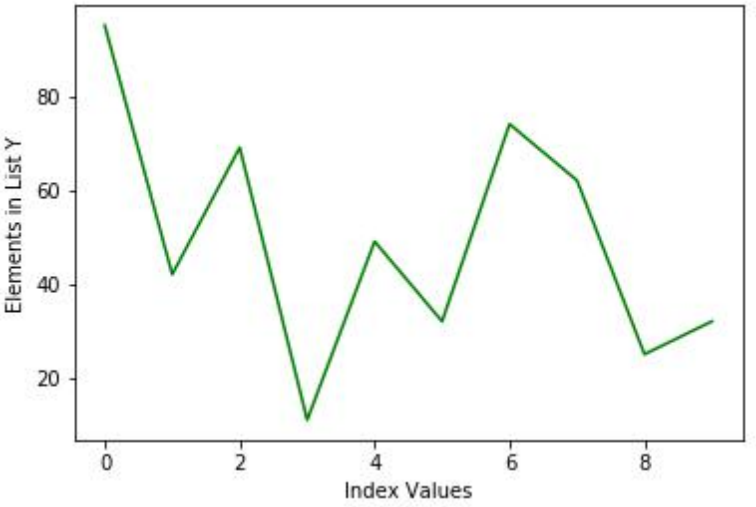

Figure 8: Line plot with green line

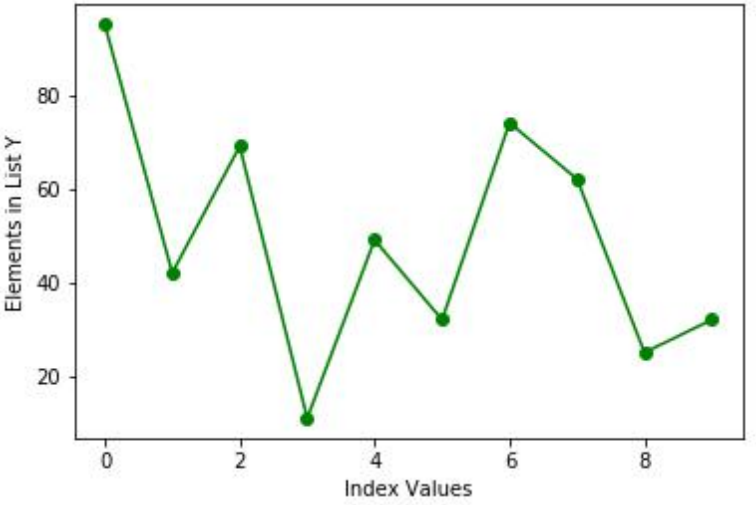

Figure 9: Line plot with circle markers

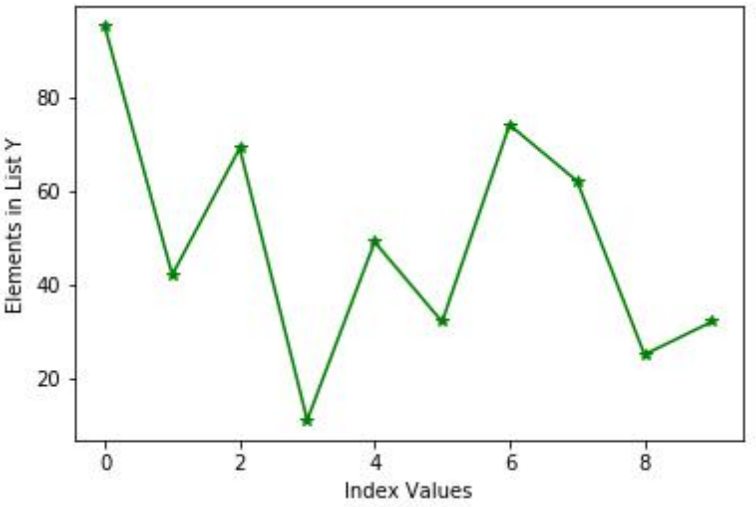

Figure 10: Line chart with asterisk markers

we like. It is also possible to change marker style by tweaking marker parameter in *fmt* string as shown below:

```
# -Example 10-
# Plot continuous green line with asterisk markers
plt.plot(y, 'g*-')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in List Y')
plt.show()
```

The output of the above code is *figure 10* where the line and markers share the same color, i.e. green specified by the *fmt* string. If we are to plot line and markers with different colors, we can use multiple `plot` methods to achieve the same.

```
# -Example 11-
# Plot list 'y'
plt.plot(y, 'g')
```

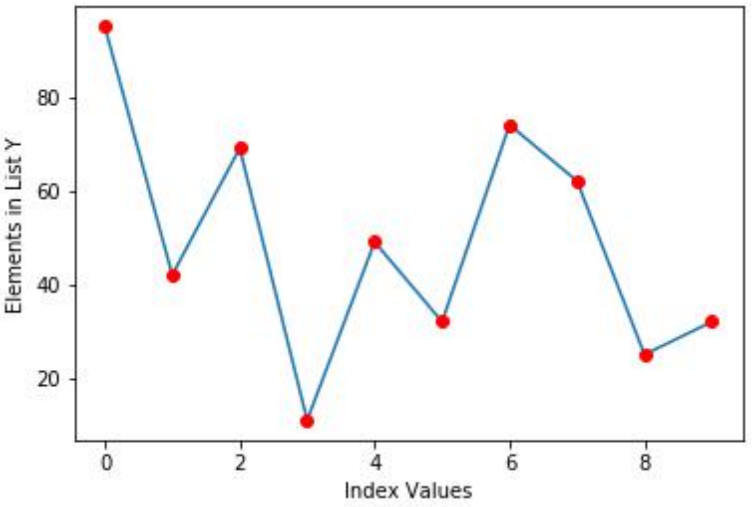

Figure 11: Plot with blue line and red markers

```
# Plot red circle markers
plt.plot(y, 'ro')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in List Y')
plt.show()
```

The above code plots line along with *red circle markers* as seen in *figure 11*. Here, we first plot the line with the default style and then attempt to plot *markers* with attributes `r` referring to red color and `o` referring to circle. On the same lines, we can plot multiple sets of data using the same technique. The example given below plots two lists on the same plot.

```
# -Example 12: Technique 1-
# Define two lists
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]
y2 = [35, 52, 96, 77, 36, 66, 50, 12, 35, 63]

# Plot lists and show them
plt.plot(y, 'go-')
```

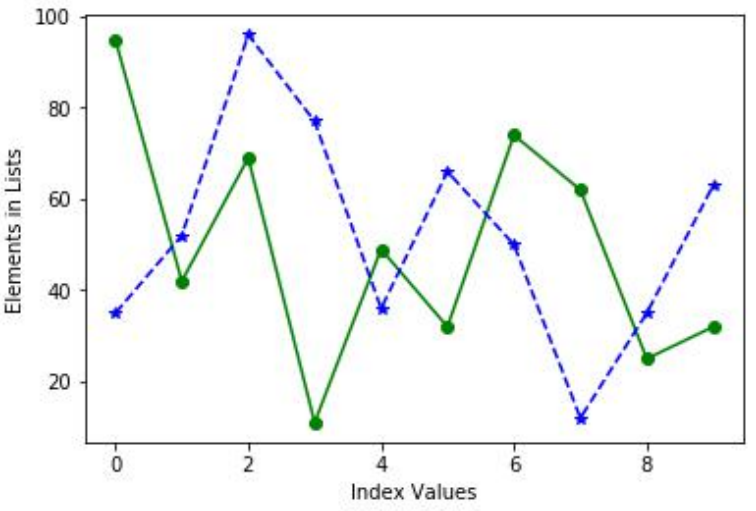

Figure 12: Line plot with two lines

```
plt.plot(y2, 'b*--')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in Lists')
plt.show()
```

The output can be seen in *figure 12* where both green and blue lines are drawn on the same plot. We can achieve the same result as shown above using the different technique as shown below:

```
# -Example 12: Technique 2-
# Plot lists and show them
plt.plot(y, 'go-', y2, 'b*--')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in Lists')
plt.show()
```

Essentially, the `plot` method makes it very easy to plot sequential data

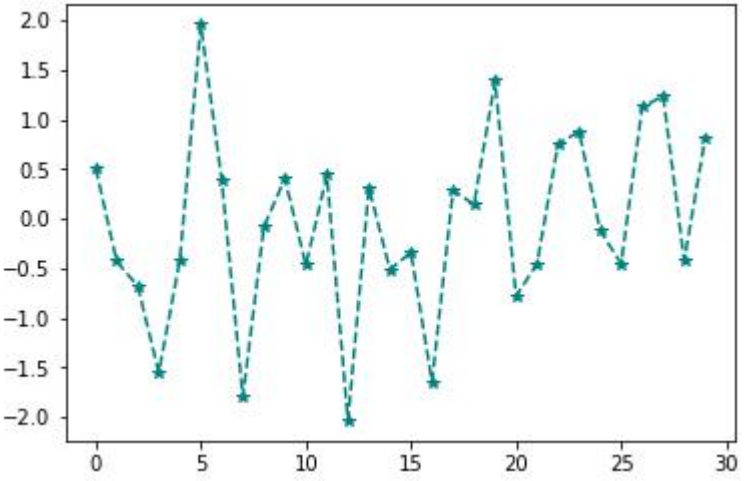

Figure 13: Line plot from a NumPy array

structure such as list, NumPy arrays, pandas series, etc. Similar to plotting lists, we can plot NumPy arrays directly via the `plot` method. Let us plot NumPy one dimensional array. As we are executing codes directly in IPython console, calling the `plt.show()` is not required and hence, we will not be calling the same in subsequent examples. However, remember, it is absolutely necessary to call it while writing Python code in order to show a plot.

```
# -Example 13-
# Importing NumPy library
import numpy as np

# Drawing 30 samples from a standard normal distribution
# into an array 'arr'
arr = np.random.normal(size=30)

# Plotting 'arr' with dashed line-style and * markers
plt.plot(arr, color='teal', marker='*', linestyle='dashed')
```

In the above example, we draw thirty samples from a normal distribution into an array `arr` which in turn gets plotted as a *dashed* line along with

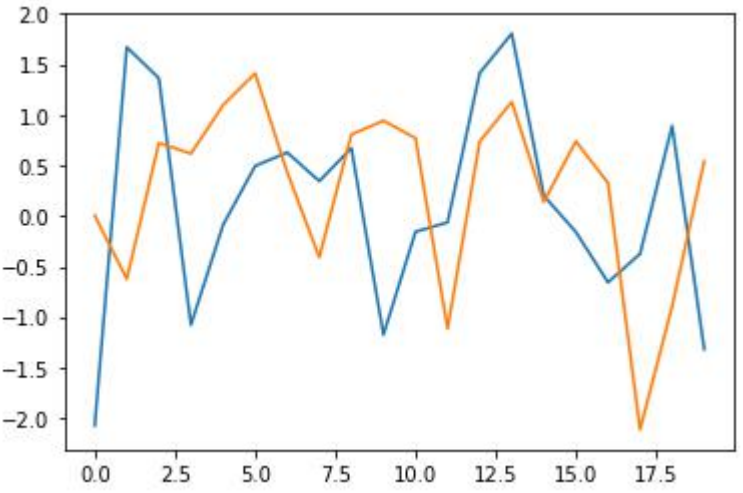

Figure_14: Line plot from 2-D NumPy array

asterisk markers as seen in the *figure 13*.

Plotting two-dimensional arrays follows the same pattern. We provide a 2-D array to a `plot` method to plot it. The below code shows the example of this whose output is shown in *figure 14*.

```
# -Example 14-
# Creating a two dimensional array 'arr_2d' with 40 samples
# and shape of (20, 2)
arr_2d = np.random.normal(size=40).reshape(20, 2)

# Plotting the array
plt.plot(arr_2d)
```

Let us now move our focus to plot pandas data structures. The pandas library uses the standard convention as the matplotlib for plotting directly from its data structures. The pandas also provide a `plot` method which is equivalent to the one provided by matplotlib. Hence, the `plot` method can be called directly from pandas Series and DataFrame objects. The `plot` method on Series and DataFrame is just a simple wrapper around `plt.plot()`. The below example illustrates plotting pandas Series object:

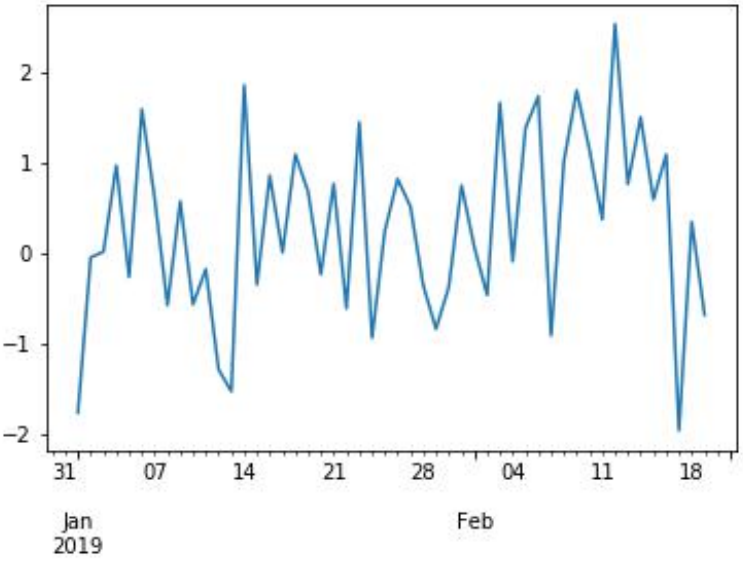

Figure 15: Line plot from a pandas series

```
# -Example 15-
# Importing necessary libraries
import pandas as pd
import numpy as np

# Creating pandas Series with 50 samples drawn from normal
# distribution
ts = pd.Series(np.random.normal(size=50),
               index=pd.date_range(start='1/1/2019',
               periods=50))

# Plotting pandas Series
ts.plot()
```

In the above example, we call the `plot` method directly on pandas Series object `ts` which outputs the plot as shown in *figure 15*. Alternatively, we could have called `plt.plot(ts)`. Calling `ts.plot()` is equivalent to calling `plt.plot(ts)` and both calls would result in almost the same output as shown above. Additionally, the `plot()` method on pandas object sup-

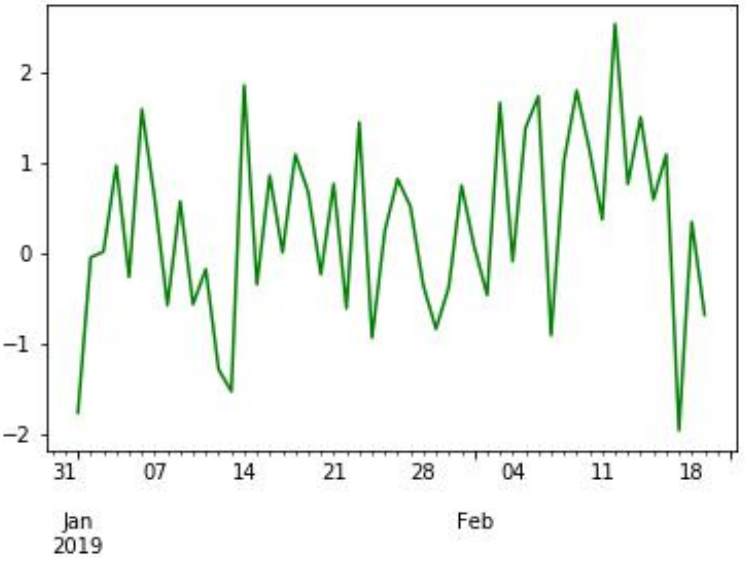

Figure 16: Line plot from a pandas series in green color

ports almost every attribute that `plt.plot()` supports for formatting. For example, calling the `plot` method on pandas objects with a `color` attribute would result in a plot with color mentioned by its value. This is shown below:

```
# -Example 16-
# Plotting pandas Series in green color
ts.plot(color='green')
```

The output of the above code is shown in *figure 16*.

Moving forward, the same notation is followed by pandas DataFrame object and visualizing data within a dataframe becomes more intuitive and less quirky. Before we attempt to plot data directly from a dataframe, let us create a new dataframe and populate it. We fetch the stock data of AAPL ticker that we will be using for illustration purposes throughout the remaining chapter.

```
# -Script to fetch AAPL data from a web resource-
# Import libraries
```

```
import pandas as pd

# Fetch data
data = pd.read_csv('https://bit.ly/2WcsJE7', index_col=0,
                   parse_dates=True)
```

The dataframe `data` will contain stock data with dates being the index. The excerpt of the downloaded data is shown below:

Date	Open	High	Low	Close	Volume	...
2018-03-27	173.68	175.15	166.92	168.340	38962893.0	...
2018-03-26	168.07	173.10	166.44	172.770	36272617.0	...
2018-03-23	168.39	169.92	164.94	164.940	40248954.0	...

Now we can plot any column of a `data` dataframe by calling `plot` method on it. In the example given below, we plot the recent 100 data points from the `Volume` column of the dataframe:

```
# -Example 17-
# Plot volume column
data.Volume.iloc[:100].plot()
```

The output of the above code is shown in *figure 17*. With a dataframe, `plot` method is a convenience to plot all of the columns with labels. In other words, if we plot multiple columns, it would plot labels of each column as well. In the below example, we plot `AdjOpen` and `AdjClose` columns together and the output for the same is shown in *figure 18*.

```
# -Example 18-
data[['AdjOpen', 'AdjClose']][:50].plot()
```

The `plot` method generates a line plot by default when called on pandas data structures. However, it can also produce a variety of other charts as we will see later in this chapter. Having said that, lets head forward to plot scatter plots.

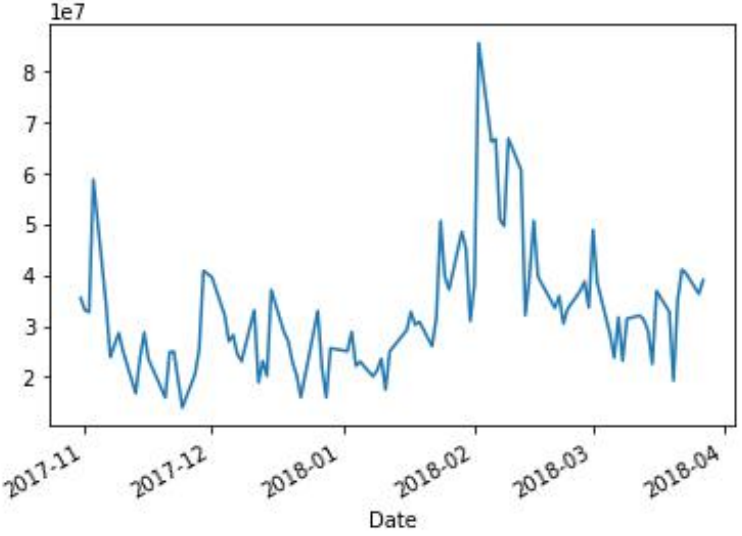

Figure 17: Line plot of a volume column

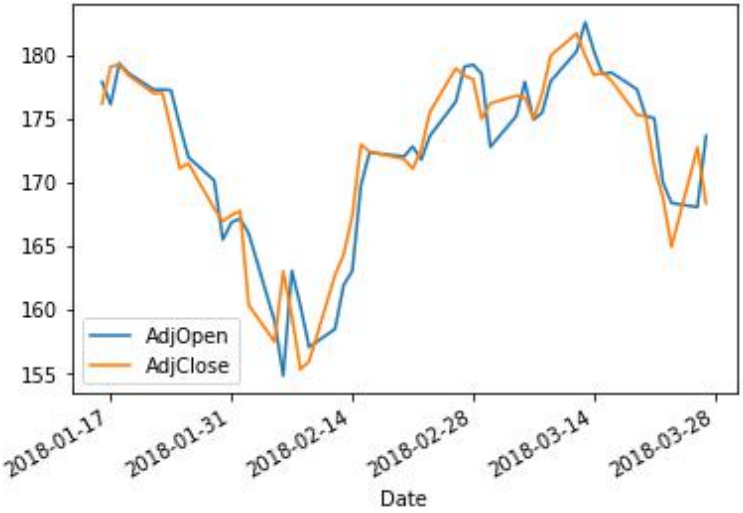

Figure 18: Line plot of two columns

12.2.2 Scatter Plot

Scatter plots are used to visualize the relationship between two different data sets. Matplotlib provides the `scatter` method within `pyplot` sub-module using which scatter plots can be generated.

- `plt.scatter` generates scatter plot of y vs x with varying marker size and/or color.

The x and y parameters are data positions and it can be array-like sequential data structures. There are some instances where we have data in the format that lets us access particular variables with string. For example, Python dictionary or pandas dataframe. Matplotlib allows us to provide such an object with the `data` keyword argument to the `scatter` method to directly plot from it. The following example illustrates this using a dictionary.

```
# -Example 19-
# Creating a dictionary with three key-value pairs
dictionary = {'a': np.linspace(1, 100, 50),
              'c': np.random.randint(0, 50, 50),
              'd': np.abs(np.random.randn(50)) * 100}

# Creating a new dictionary key-value pair
dictionary['b'] = dictionary['a'] * np.random.rand(50)

# Plotting a scatter plot using argument 'data'
plt.scatter('a', 'b', c='c', s='d', data=dictionary)

# Labeling the plot and showing it
plt.xlabel('A Data Points')
plt.ylabel('B Data Points')
plt.show()
```

In the above code, we created a dictionary `dictionary` with four key-value pairs. Values in key `a` and `b` contain fifty random values to be plotted on a scatter plot. Key `c` contains fifty random integers and key `d` contains fifty positive floats which represents color and size respectively for each scatter data point. Then, a call to `plt.scatter` is made along with all keys and the `dictionary` as the value to `data`. The argument `c` within the call refers

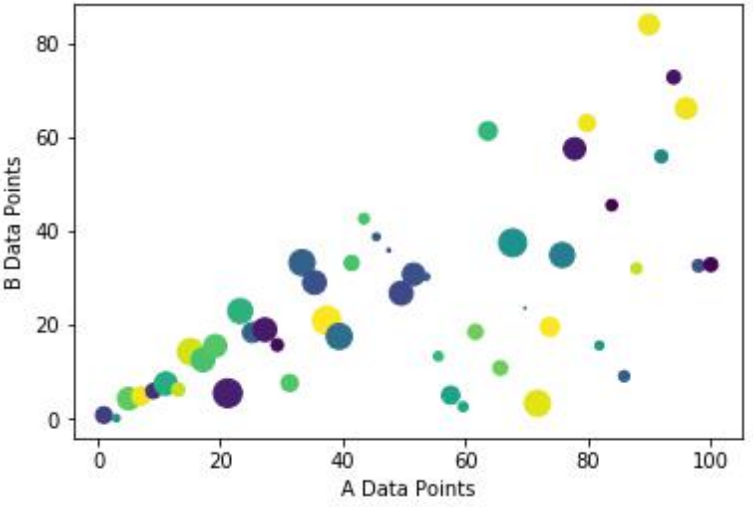

Figure 19: Scatter plot with different size and color

to color to be used and the argument `s` represents the size of a data point. These arguments `c` and `s` are optional. The output we get is a scatter plot with different size and color as shown in *figure 19*. A simple scatter plot with the same color and size gets plotted when we omit these optional arguments as shown in the following example:

```
# -Example 20-
# Creating a scatter plot without color and the same size
plt.scatter(dictionary['a'], dictionary['b'])

# Labeling the plot and showing it
plt.xlabel('A Data Points')
plt.ylabel('B Data Points')
plt.show()
```

The output of the above code will be a scatter plot as shown in *figure 20*. To better understand the working of scatter plots, let us resort to our old friends: lists `x` and `y`. We defined them earlier whem we learned line plots and scatter plots. To refresh our memory, we re-define the same lists below:

```
# Data points for scatter plot
```

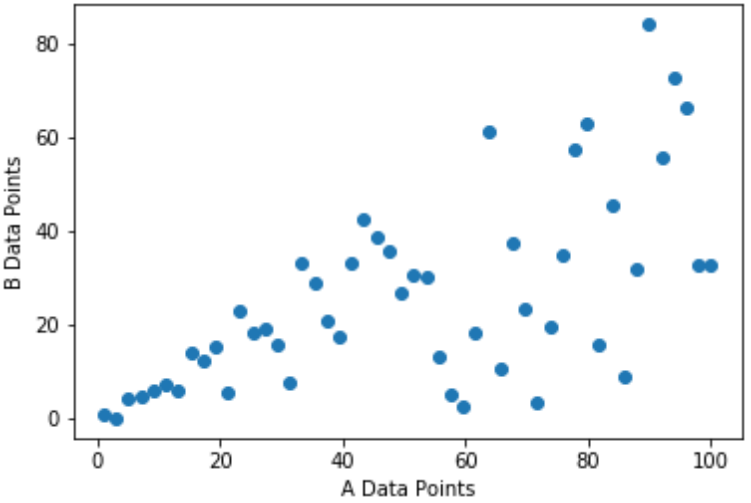

Figure 20: Scatter plot with the same size and color

```
x = [1.3, 2.9, 3.1, 4.7, 5.6, 6.5, 7.4, 8.8, 9.2, 10]
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]
```

In addition to these lists, we would be defining two more NumPy arrays `color` and `size` which determines the color and size respectively of each data point while plotting the scatter plot.

```
# Arrays which defines color and size of each data point
color = np.random.rand(10)
size = np.random.randint(50, 100, 10)
```

Now that we have data points ready, we can plot a scatter plot out of them as below:

```
# -Example 21-
# Creating a scatter plot
plt.scatter(x, y, c=color, s=size)

# Labeling the plot and showing it
plt.xlabel('Values from list x')
plt.ylabel('Values from list y')
plt.show()
```

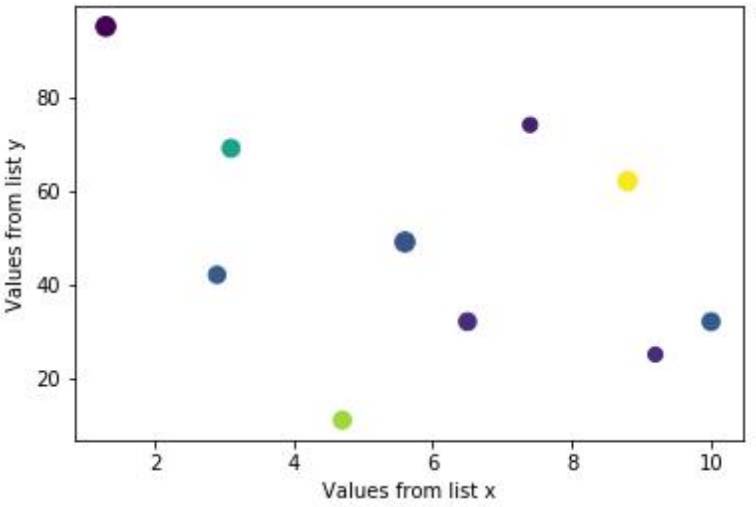

Figure 21: Scatter plot of lists x and y

The scatter plot would contain data points each with different color and size (as they are randomly generated). The output is shown in figure *figure 21*.

In finance, scatter plots are widely used to determine the relations between two data sets visually. With our working knowledge of scatter plots, let's plot `AdjOpen` and `AdjClose` prices of AAPL stock that we have in pandas dataframe `data`. When it comes to plotting data directly from a pandas dataframe, we can almost always resort to `plot` method on pandas to plot all sorts of plots. That is, we can directly use the `plot` method on the dataframe to plot scatter plots akin to line plots. However, we need to specify that we are interested in plotting a scatter plot using the argument `kind='scatter'` as shown below:

```
# -Example 22-
# Plotting a scatter plot of 'AdjOpen' and 'AdjClose' of
# AAPL stock
data.plot(x='AdjOpen', y='AdjClose', kind='scatter')
plt.show()
```

Interestingly, we only need to specify column names of a dataframe `data`

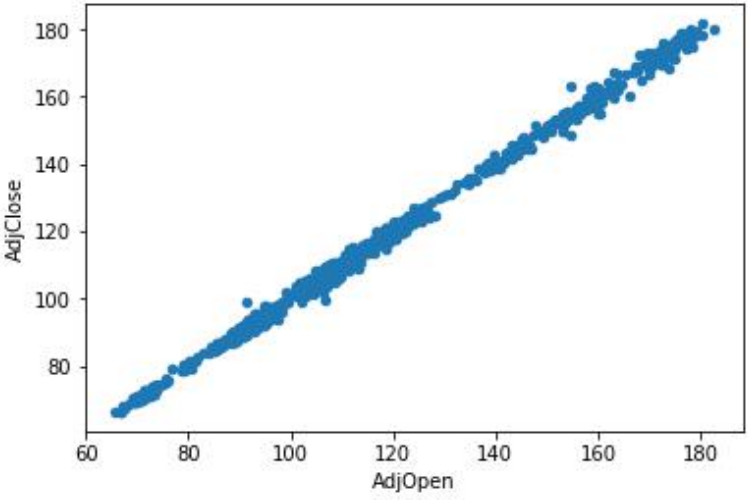

Figure 22: Scatter plot of columns `AdjOpen` and `AdjClose`

for `x` and `y` coordinates along with the argument `kind` which gets resulted in the output as shown in *figure 22*.

By visualizing price patterns using a scatter plot, it can be inferred that open and close prices are positively correlated. Furthermore, we can generate the same plot using the `plt.scatter` method.

```
# Method 1
plt.scatter(x='AdjOpen', y='AdjClose', data=data)
plt.show()

# Method 2
plt.scatter(x=data['AdjOpen'], y=data['AdjClose'])
plt.show()
```

The first method uses the argument `data` which specifies the data source, whereas the second method directly uses dataframe slicing and hence, there is no need to specify the `data` argument.

12.2.3 Histogram Plots

A histogram is a graphical representation of the distribution of data. It is a kind of bar graph and a great tool to visualize the frequency distribution of data that is easily understood by almost any audience. To construct a histogram, the first step is to *bin* the range of data values, divide the entire range into a series of intervals and finally count how many values fall into each interval. Here, the bins are consecutive and non-overlapping. In other words, histograms shows the data in the form of some groups. All the bins/groups go on X-axis, and Y-axis shows the frequency of each bin/group.

The matplotlib library offers a very convenient way to plot histograms. To create a histogram, we use the `hist` method of `pyplot` sub-module of the `matplotlib` library as shown in the below example:

```
# -Example 23-
# Data values for creating a histogram
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]

# Creating a histogram
plt.hist(y)
plt.xlabel('Bins')
plt.ylabel('Frequency')
plt.show()
```

This is the simplest code possible to plot a histogram with minimal arguments. We create a range of values and simply provide it to the `hist` method and let it perform the rest of the things (creating bins, segregating each value to corresponding bin, plotting, etc.). It produces the plot as shown in *figure 23*. The `hist` method also take `bins` as an optional argument. If this argument is specified, bins will be created as per the specified value, otherwise, it will create bins on its own. To illustrate this, we explicitly specify the number of bins in the above code and generate the plot. The modified code and output is shown below:

```
# -Example 24-
# Data values for creating a histogram
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]
```

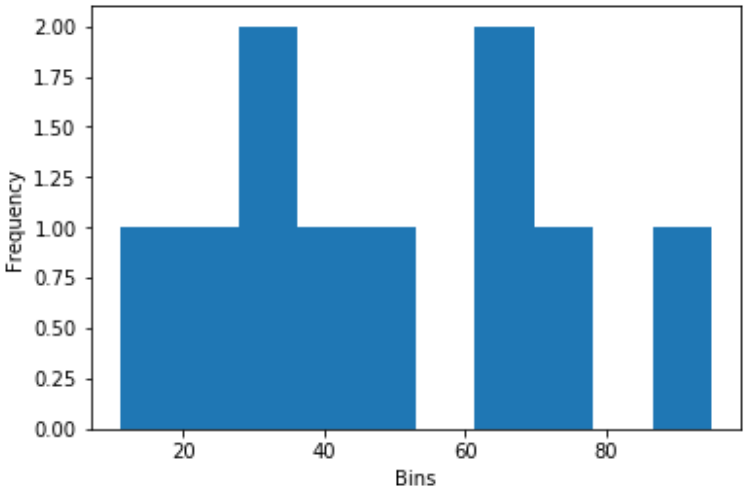

Figure 23: A histogram

```
# Creating a histogram
plt.hist(y, bins= 20)
plt.xlabel('Bins')
plt.ylabel('Frequency')
plt.show()
```

The output we got in *figure 24* is very straight forward. Number `32` appears twice in the list `y` and so it's twice as tall as the other bars on the plot. We specify the number of bins to be `20` and hence, the `hist` method tries to divide the whole range of values into 20 bins and plots them on the X-axis. Similar to the `plot` method, the `hist` method also takes any sequential data structure as its input and plots histogram of it. Let us try to generate a histogram of an array which draws samples from the standard normal distribution.

```
# -Example 25-
# Creating an array
array = np.random.normal(0, 1, 10000)

# Creating a histogram
```

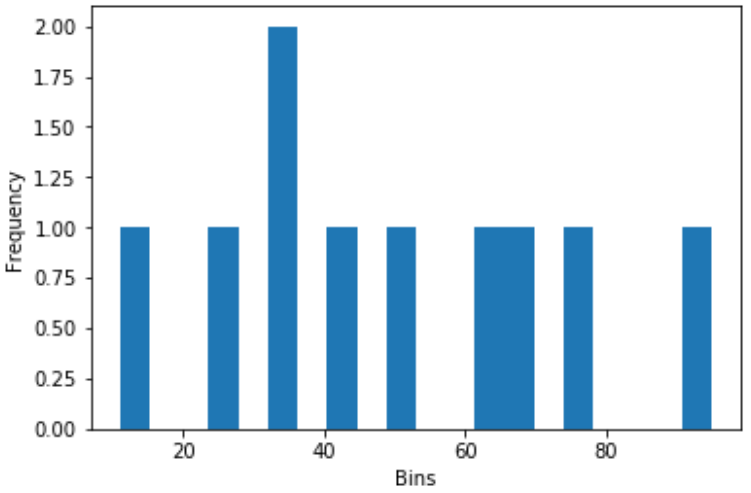

Figure 24: Histogram with 20 bins

```
plt.hist(array)
plt.xlabel('Bins')
plt.ylabel('Frequency')
plt.show()
```

The output we got in *figure 25* shows that the data distribution indeed resembles a normal distribution. Apart from `bins` argument, other arguments that can be provided to `hist` are `color` and `histtype`. There are a number of arguments that can be provided, but we will keep our discussion limited to these few arguments only. The color of a histogram can be changed using the `color` argument. The `histtype` argument takes some of the pre-defined values such as `bar`, `barstacked`, `step` and `stepfilled`. The below example illustrates the usage of these arguments and the output is shown in *figure 26*.

```
# -Example 26-
# Creating an array
array = np.random.normal(0, 1, 10000)

# Creating a histogram and plotting it
plt.hist(array, color='purple', histtype='step')
```

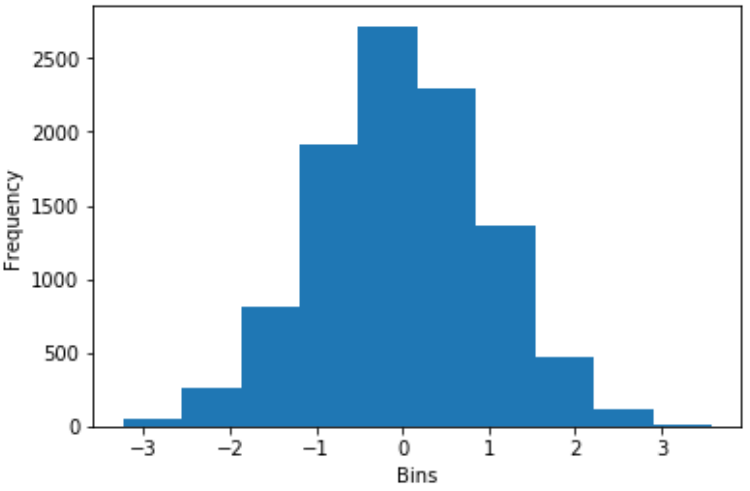

Figure 25: Histogram of an array

```
plt.xlabel('Bins')
plt.ylabel('Frequency')
plt.show()
```

In addition to optional arguments discussed so far, one argument that needs attention is `orientation`. This argument takes either of two values: `horizontal` or `vertical`. The default is `vertical`. The below given example demonstrate the usage of orientation and the output is shown in *figure 27*.

```
# -Example 27-
# Creating an array
array = np.random.normal(0, 1, 10000)

# Creating a histogram and plotting it
plt.hist(array, color='teal', orientation='horizontal')
plt.xlabel('Frequency')
plt.ylabel('Bins')
plt.show()
```

We now shift our focus on plotting a histogram directly from a pandas dataframe. Again, the `plot` method within pandas provides a wrapper

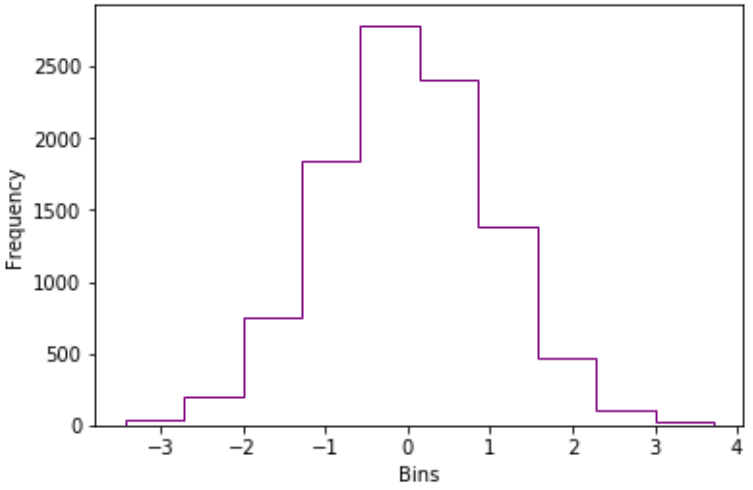

Figure 26: Histogram with `histtype='step'`

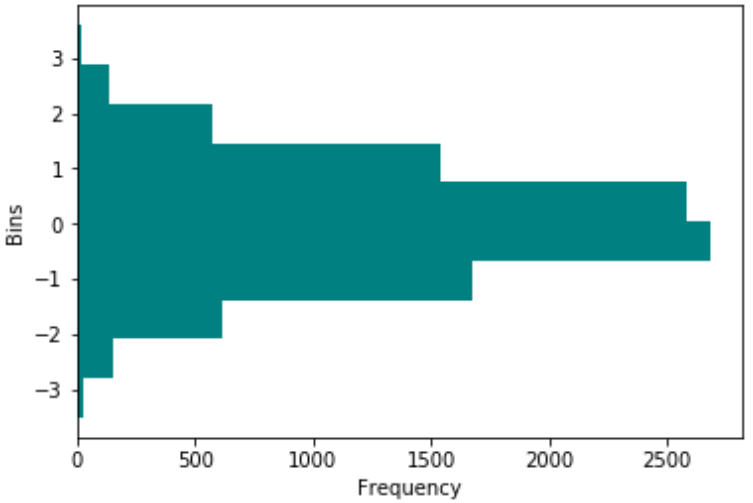

Figure 27: Histogram with horizontal orientation

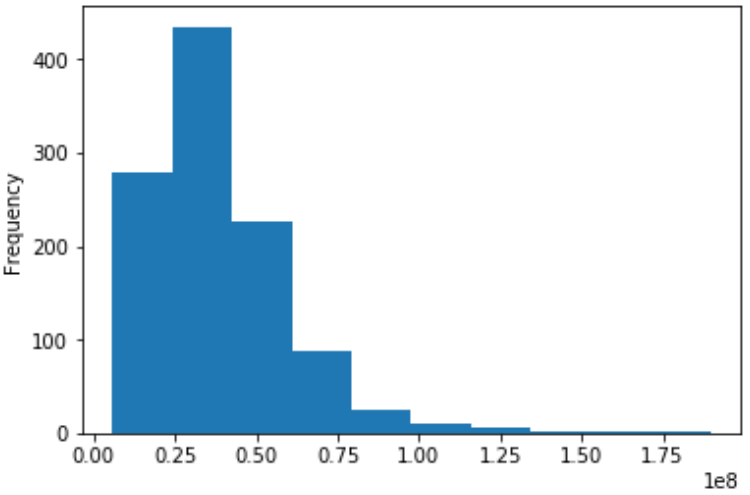

Figure 28: Histogram of a volume column

around the `hist` function in matplotlib as was the case with scatter plots. To plot a histogram, we need to specify the argument `kind` with the value `hist` when a call to `plot` is made directly from the dataframe. We will be working with the same dataframe `data` that contains historical data for AAPL stock.

```
# -Example 28: Technique 1-
# Creating a histogram using a dataframe method
data['Volume'].plot(kind='hist')
plt.show()

# -Example 28: Technique 2-
plt.hist(data['Volume'])
plt.ylabel('Frequency')
plt.show()
```

In the first method, we directly make a call to `plot` method on the dataframe `data` sliced with `Volume` column. Whereas in the second method, we use the `hist` method provided by `matplotlib.pyplot` module to plot the histogram. Both methods plot the same result as shown in *figure 28*.

Figure 29: Line plot of close prices

12.3 Customization

Now that we have got a good understanding of plotting various types of charts and their basic formatting techniques, we can delve deeper and look at some more formatting techniques. We already learned that matplotlib does not add any styling components on its own. It will plot a simple plain chart by default. We, as users, need to specify whatever customization we need. We start with a simple line plot and will keep on making it better. The following example shows plotting of close prices of the AAPL ticker that is available with us in the dataframe `data`.

```
# -Example 29-
# Extracting close prices from the dataframe
close_prices = data['AdjClose']

# Plotting the close prices
plt.plot(close_prices)
plt.show()
```

Here, as shown in *figure 29* the `close_prices` is the pandas Series object which gets plotted using the `plot` method. However, values on the X-axis

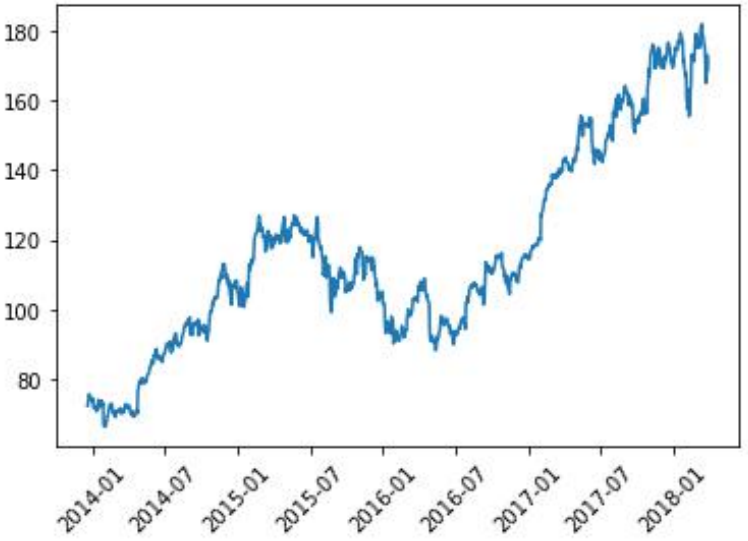

Figure 30: Line plot with rotated `xticks`

are something that we don't want. They are all overlapped with each other. This happens as the `plot` method did not find sufficient space for each date. One way to overcome this issue is to rotate the values on the X-axis to make it look better.

```
# -Example 30-
plt.plot(close_prices)

# Rotating the values along x-axis to 45 degrees
plt.xticks(rotation=45)
plt.show()
```

The `xticks` method along with the `rotation` argument is used to rotate the values/tick names along the x-axis. The output of this approach is shown in *figure 30*. Another approach that can be used to resolve the overlapping issue is to increase the figure size of the plot such that the matplotlib can easily show values without overlapping. This is shown in the below example and the output is shown in *figure 31*:

```
# -Example 31-
# Creating a figure with the size 10 inches by 5 inches
```

Figure 31: Line plot with custom figure size

```
fig = plt.figure(figsize=(10, 5))
plt.plot(close_prices)
plt.show()
```

Similarly, the matplotlib provides `yticks` method that can be used to customize the values on the Y-axis. Apart from the `rotation` argument, there are a bunch of other parameters that can be provided `xticks` and `yticks` to customize them further. We change the font size, color and orientation of ticks along the axes using the appropriate arguments within these methods in the following example:

```
# -Example 32-
# Creating a figure, setting its size and plotting close
# prices on it
fig = plt.figure(figsize=(10, 5))
plt.plot(close_prices, color='purple')

# Customizing the axes
plt.xticks(rotation=45, color='teal', size=12)
plt.yticks(rotation=45, color='teal', size=12)

# Setting axes labels
plt.xlabel('Dates', {'color': 'orange', 'fontsize':15})
plt.ylabel('Prices', {'color': 'orange', 'fontsize':15})
plt.show()
```

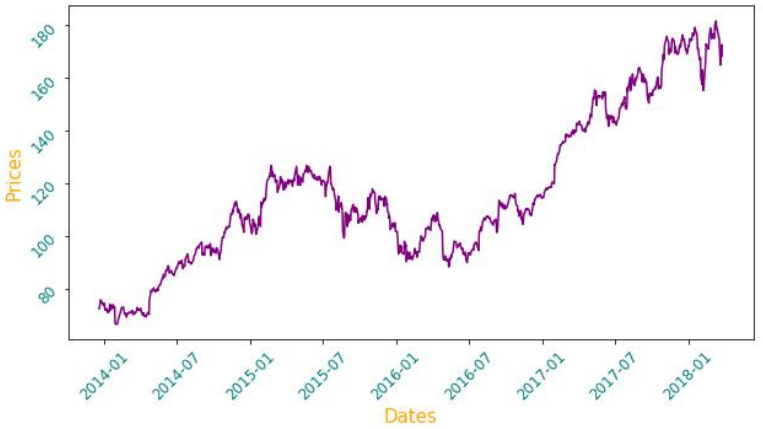

Figure 32: Line plot with rotated ticks on axes and colored values

Along with the axes values, we change the color and font size of axes labels as shown in *figure 32*. There are numbers of other customizations possible using various arguments and matplotlib provides total flexibility to create the charts as per one's desire. Two main components that are missing in the above plot are title and legend, which can be provided using the methods `title` and `legends` respectively. Again, as with the other methods, it is possible to customize them in a variety of way, but we will be restricting our discussion to a few key arguments only. Adding these two methods as shown below in the above code would produce the plot as shown in *figure 33*:

```
# -Example 33-
# Showing legends and setting the title of plot
plt.legend()
plt.title('AAPL Close Prices', color='purple', size=20)
```

Another important feature that can be added to a figure is to draw a grid within a plot using the `grid` method which takes either `True` or `False`. If true, a grid is plotted, otherwise not. An example of a plot with grid is shown below and its output is shown in *figure 34*.

```
# -Example 34-
# Adding the grid to the plot
plt.grid(True)
```

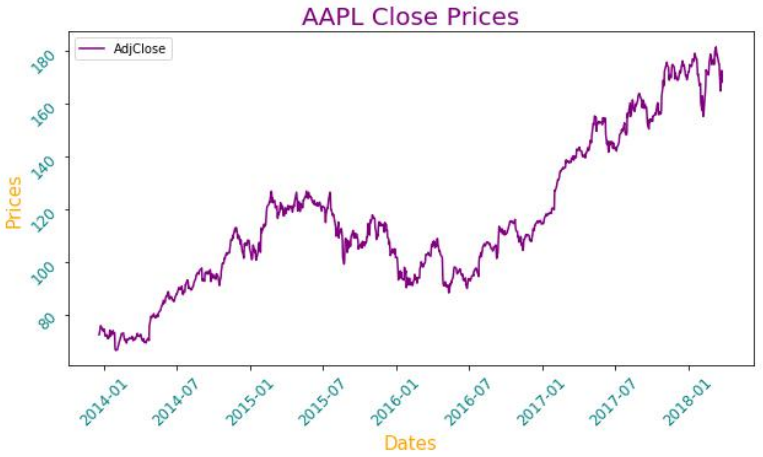

Figure 33: Line plot with legends and the title

The `axhline` method allows us to add a horizontal line across the axis to the plot. For example, we might consider adding the mean value of close prices to show the average price of a stock for the whole duration. It can be added using `axhline` method. Computation of mean value and its addition to the original plot is shown below:

```
# -Example 35-
# Importing NumPy library
import numpy as np

# Calculating the mean value of close prices
mean_price = np.mean(close_prices)

# Plotting the horizontal line along with the close prices
plt.axhline(mean_price, color='r', linestyle='dashed')
```

Now that we have the mean value of close prices plotted in the *figure 35*, one who looks at the chart for the first time might think what this red line conveys? Hence, there is a need to explicitly mention it. To do so, we can use the `text` method provided by `matplotlib.pyplot` module to plot text anywhere on the figure.

```
# -Example 36-
```

Figure 34: Line plot with a grid

Figure 35: Line plot with horizontal line

Figure 36: Line plot with text on it

```
# Importing DateTime from DateTime library
from datetime import datetime

# Plotting text on date 2014-1-7 and price 120
plt.text(datetime(2014,1,7), 120, 'Mean Price',
size=15, color='r')
```

The `text` method takes three compulsory arguments: `x`, `y` and `t` which specifies the coordinates on X and Y-axis and text respectively. Also, we use a `datetime` sub-module from a `datetime` library to specify a date on the X-axis as the plot we are generating has dates on the X-axis. The chart with text indicating the mean price is shown in *figure 36*.

Using all these customization techniques, we have been able to evolve the dull looking price series chart to a nice and attractive graphic which is not only easy to understand but presentable too. However, we have restricted ourselves to plotting only a single chart. Let us brace ourselves and learn to apply these newly acquired customization techniques to multiple plots.

We already learned at the beginning of this chapter that a figure can have multiple plots, and that can be achieved using the `subplots` method. The following examples show stock prices of AAPL stock along with its traded

volume on each day. We start with a simple plot that plots stock prices and volumes in the below example:

```
# -Example 37-
# Extracting volume from the dataframe 'data'
volume = data['AdjVolume']

# Creating figure with two rows and one column
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1,
                               sharex=True,
                               figsize=(10, 8))

# Plotting close prices on the first sub-plot
ax1.plot(close_prices, color='purple')
ax1.grid(True)

# Plotting trading volume on the second sub-plot
ax2.bar(volume.index, volume)
ax2.grid(True)

# Displaying the plot
plt.show()
```

First, we extract the `AdjVolume` column from the `data` dataframe into a `volume` which happens to be pandas series object. Then, we create a figure with sub-plots having two rows and a single column. This is achieved using `nrows` and `ncols` arguments respectively. The `sharex` argument specifies that both sub-plots will share the same x-axis. Likewise, we also specify the figure size using the `figsize` argument. These two subplots are unpacked into two axes: `ax1` and `ax2` respectively. Once, we have the axes, desired charts can be plotted on them.

Next, we plot the `close_prices` using the `plot` method and specify its color to be `purple` using the `color` argument. Similar to the `plot` method, matplotlib provides `bar` method to draw bar plots which takes two arguments: the first argument to be plotted on the X-axis and second argument to be plotted along the y-axis. For our example, values on X-axis happens to be a date (specified by `volume.index`), and value for each bar on the Y-axis is provided using the recently created `volume` series. After that, we plot grids

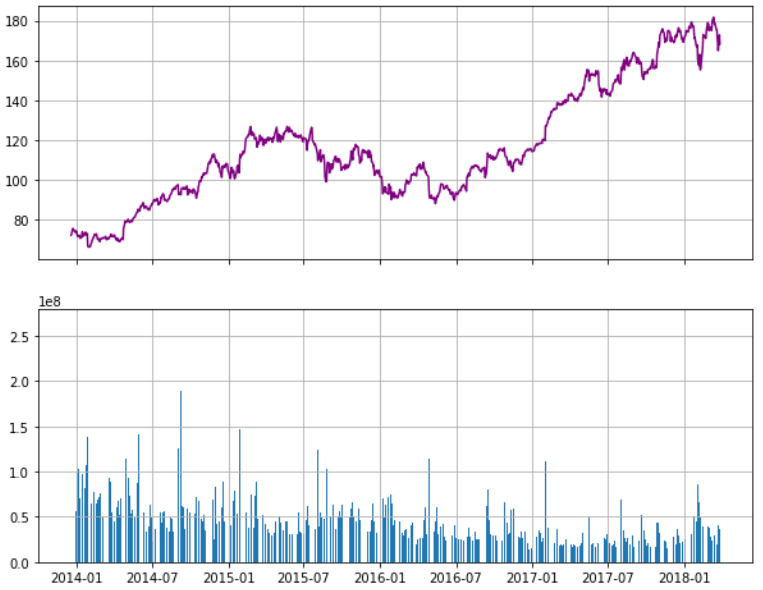

Figure 37: Sub-plots with stock price and volume

on both plots. Finally, we display both plots. As can be seen above in *figure 37*, matplotlib rendered a decent chart. However, it misses some key components such as title, legends, etc. These components are added in the following example:

```
# -Example 38-
# Creating figure with multiple plots
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1,
                               sharex=True,
                               figsize=(10, 8))
ax1.plot(close_prices, color='purple', label='Prices')
ax1.grid(True)

# Setting the title of a first plot
ax1.set_title('Daily Prices')

# Setting the legend for the first plot
ax1.legend()
```

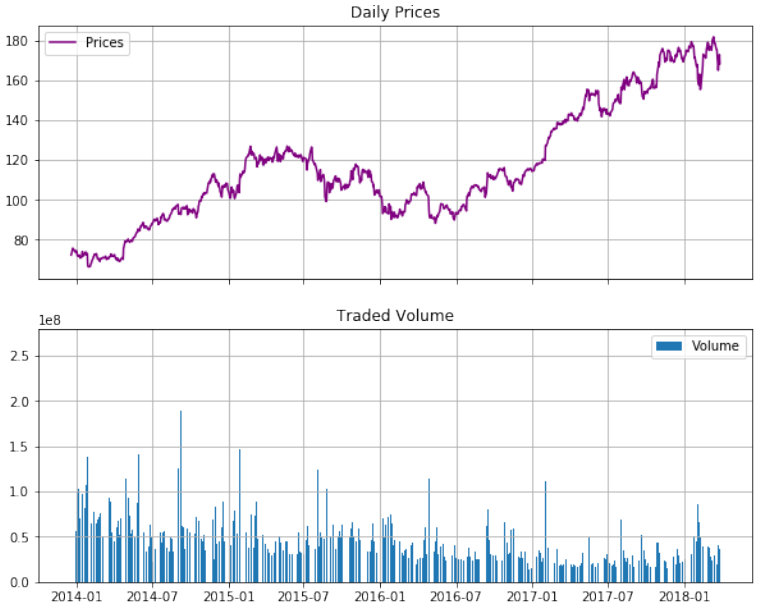

Figure 38: Sub-plots with legends and titles

```
ax2.bar(volume.index, volume, label='Volume')
ax2.grid(True)

# Setting the title of a second plot
ax2.set_title('Volume')

# Setting the legend for the second plot
ax2.legend()

plt.show()
```

Here, we use the `legend` method to set legends in both plots as shown in *figure 38*. Legends will print the values specified by the `label` argument while plotting each plot. The `set_title` is used to set the title for each plot. Earlier, while dealing with the single plot, we had used the `title` method to set the title. However, it doesn't work the same way with multiple plots.

Another handy method provided by the matplotlib is the `tight_layout`

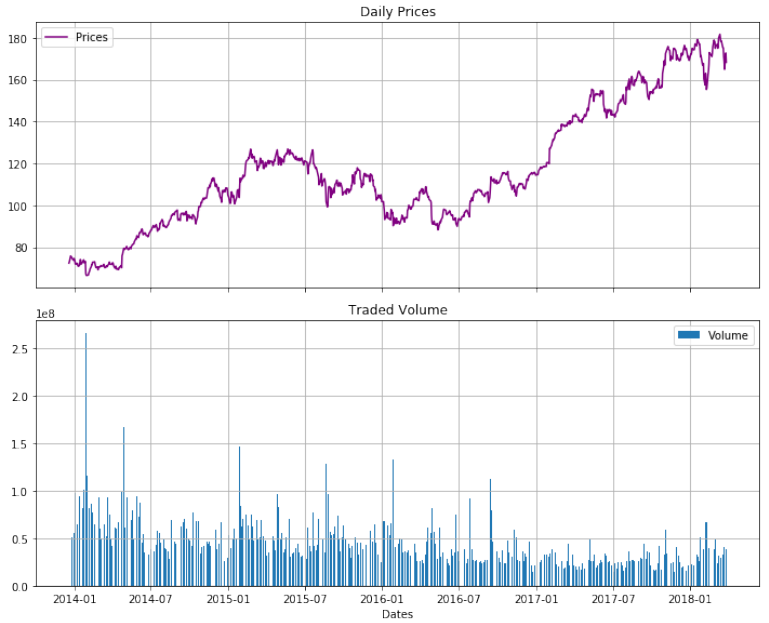

Figure 39: Sub-plots with tight layout

method which automatically adjusts the padding and other similar parameters between subplots so that they fits into the figure area.

```
# -Example 39-
# Setting layout
plt.tight_layout()

# Setting label on the x-axis
plt.xlabel('Dates')
plt.show()
```

The above code explicitly specifies the layout and the label on the x-axis which results into the chart as shown in *figure 39*.

In addition to all this customization, matplotlib also provides a number of predefined styles that can be readily used. For example, there is a predefined style called "ggplot", which emulates the aesthetics of *ggplot* (a popular plotting package for R language). To change the style of plots being

rendered, the new style needs to be explicitly specified using the following code:

```
plt.style.use('ggplot')
```

One the style is set to use, all plots rendered after that will use the same and newly set style. To list all available styles, execute the following code:

```
plt.style.available
```

Let us set the style to one of the pre-defined styles known as *'fivethirtyeight'* and plot the chart.

```
# -Example 40-
plt.style.use('fivethirtyeight')
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1,
                               sharex=True,
                               figsize=(10, 8))

ax1.plot(close_prices, color='purple', label='Prices')
ax1.grid(True)
ax1.set_title('Daily Prices')
ax1.legend()

ax2.bar(volume.index, volume, label='Volume')
ax2.grid(True)
ax2.set_title('Traded Volume')
ax2.legend()

plt.tight_layout()

plt.xlabel('Dates')
plt.show()
```

The output of the above code is shown in the *figure 40*. By changing the style, we get a fair idea about how styles play an important role to change the look of charts cosmetically while plotting them.

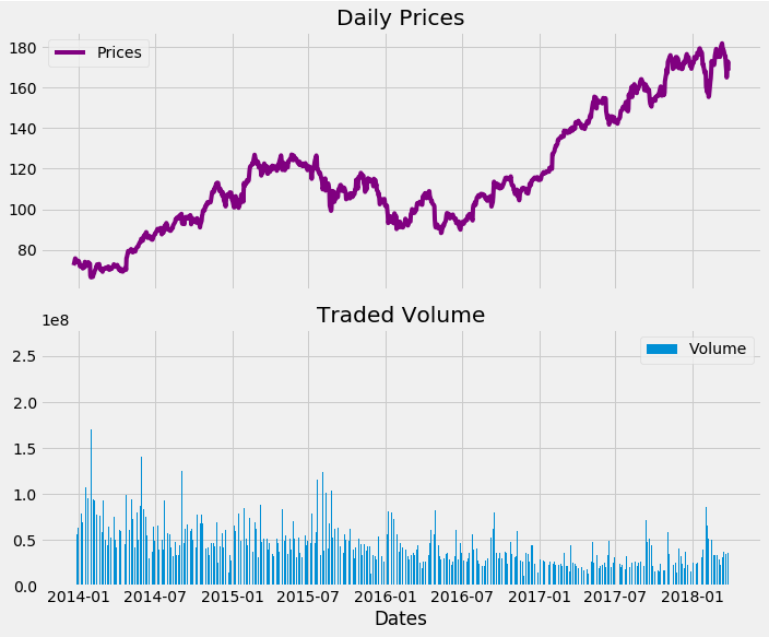

Figure 40: Plot with pre-defined style `'fivethirtyeight'`

The last method that we will study is the `savefig` method that is used to save the figure on a local machine. It takes the name of the figure by which it will be saved. This is illustrated below:

```
plt.savefig('AAPL_chart.png')
```

Executing the above code will save the chart we plotted above with the name `AAPL_chart.png`.
This brings us to the end of this chapter. We started with the basics of figure and plots, gradually learning various types of charts and along with the finer details.

We also learned customization and took a sneak peek into plotting multiple plots within the same chart.

12.4 Key Takeaways

1. Matplotlib does not fall under the Python Standard Library. Hence, it needs to be installed before it can be used.
2. The pyplot module within matplotlib provides the common charting functionality and is used to plot various kinds of charts. A common practice is to import it using the alias `plt`.
3. The code `%matplotlib inline` is used to enable plotting charts within Jupyter Notebook.
4. Figure is the top-level container and Axes is the area where plotting occurs. `plt.figure()` creates an empty figure with axes. The `figsize` argument is used to specify the figure size.
5. The `show()` method of pyplot sub-module is used to display the plot.
6. Methods available on the axes object can also be called directly using the `plt` notation.
7. The `plot()` method from pyplot module is used to plot line chart from a sequential data structures such as lists, tuples, NumPy arrays, pandas series, etc.
8. The `scatter()` method from pyplot module is used to generate scatter plots.
9. The `hist()` method from pyplot module is used to generate histogram plots.
10. The `bar()` method from pyplot module is used to create a bar chart.

11. `xlabel()` and `ylabel()` methods are used to label the charts. The `title()` method is used to title the chart.
12. The `legend()` method is used to annotate the charts.
13. The `axhline()` is used to add a horizontal line across the axis to a plot.
14. `xticks()` and `yticks()` methods are used to customize the ticks along the axes in a plot.
15. The `grid()` method is used to add a grid to a plot.
16. The `subplots()` method is used to create multiple plots within a figure.
17. The `style` attribute is used to configure the style of a plot.
18. Methods discussed in this chapter have their own attributes specific to each method.

References

Chapter 1 - Introduction

1. A Byte of Python, Swaroop CH:
   https://python.swaroopch.com/about_python.html
2. Hilpisch, Yves (2014): *"Python for Finance - ANALYZE BIG FINANCIAL DATA."* O'Reilly.
3. Tutorials Point:
   https://www.tutorialspoint.com/python/python_overview.htm
4. Python Official Documentation:
   https://www.python.org/about/apps/
5. Geeks for Geeks:
   https://www.geeksforgeeks.org/important-differences-between-python-2-x-and-python-3-x-with-examples/

Chapter 2 - Getting Started with Python

1. Zhang, Y (2015): *"An Introduction to Python and Computer Programming"*. Springer, Singapore.
2. Python Docs:
   https://docs.python.org/3/tutorial/introduction.html

Chapter 3 - Variables and Data Types in Python

1. A Byte of Python, Swaroop CH:
   https://python.swaroopch.com/about_python.html
2. W3 Schools:
   https://www.w3schools.com/python/python_ref_string.asp
3. Tutorials Point:
   https://www.tutorialspoint.com/python/python_strings.htm

Chapter 4 - Modules, Packages and Libraries

1. Python Official Documentation:
   https://docs.python.org/3/tutorial/modules.html and
   https://docs.python.org/3/library/index.html

Chapter 5 - Data Structures

1. Python Official Documentation:
   https://docs.python.org/3/tutorial/datastructures.html

Chapter 6 - Keywords & Operators

1. W3 Schools:
   https://www.w3schools.com/python/

Chapter 7 - Control Flow Statements

1. W3 Schools:
   https://www.w3schools.com/python/ref_func_range.asp
2. Programiz:
   https://www.programiz.com/python-programming/list-comprehension

Chapter 8 - Iterators & Generators

1. Iterators, generators and decorators:
   https://pymbook.readthedocs.io/en/latest/igd.html
2. W3 Schools:
   https://www.w3schools.com/python/python_iterators.asp

Chapter 9 - Functions in Python

1. Python Official Documentation:
   https://docs.python.org/3/tutorial/classes.html
2. A Byte of Python, Swaroop CH:
   https://python.swaroopch.com/
3. w3schools.com:
   https://www.w3schools.com/python/python_ref_functions.asp
4. Programiz:
   https://www.programiz.com/python-programming/

Chapter 10 - Numpy Module

1. Towards Data Science:
   https://towardsdatascience.com/a-hitchhiker-guide-to-python-numpy-arrays-9358de570121
2. NumPy documentation:
   https://docs.scipy.org/doc/numpy-1.13.0/

Chapter 11 - Pandas Module

1. 10 Minutes to pandas:
   https://pandas.pydata.org/pandas-docs/stable/10min.html
2. Pandas IO Tools:
   https://pandas.pydata.org/pandas-docs/stable/io.html

Chapter 12 - Data Visualization with Matplotlib

1. Matplotlib Plot:
   https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html
2. Pyplot Tutorial:
   https://matplotlib.org/tutorials/introductory/pyplot.html

QuantInsti® is one of the pioneer algorithmic trading research and training institutes across the globe. With its educational initiatives, QuantInsti is preparing financial market professionals for the contemporary field of algorithmic and quantitative trading. QuantInsti has also designed education modules and conducted knowledge sessions for/with various exchanges in South and South-East Asia and for leading educational and financial institutions.

QuantInsti's flagship programme 'Executive Programme in Algorithmic Trading' (EPAT®) is designed for professionals looking to grow in the field algorithmic and quantitative Trading. It inspires individuals towards a successful career by focusing on derivatives, quantitative trading, electronic market-making, financial computing and risk management. This comprehensive certificate offers unparalleled insights into the world of algorithms, financial technology and changing market microstructure with its exhaustive course curriculum designed by leading industry experts and market practitioners.

Quantra® is an e-learning portal by QuantInsti that specializes in short self-paced courses on algorithmic and quantitative trading. Quantra offers an interactive environment which supports 'learning by doing' through guided coding exercises, videos and presentations in a highly interactive fashion through machine enabled learning.

Quantra Blueshift® is a comprehensive trading and strategy development platform that lets you focus more on the strategy and less on coding and data. Our cloud-based backtesting engine helps you develop, test and analyse trading strategies and fine-tune them, for free. Apart from imparting knowledge on advanced concepts through its various courses, QuantInsti contributes to the industry through various initiatives including participating in & hosting webinars, conferences and workshops in different parts of the world.

QuantInsti Quantitative Learning Pvt. Ltd.

A-309, Boomerang, Chandivali Farm Road, Powai, Mumbai – 400 072
Email: contact@quantinsti.com