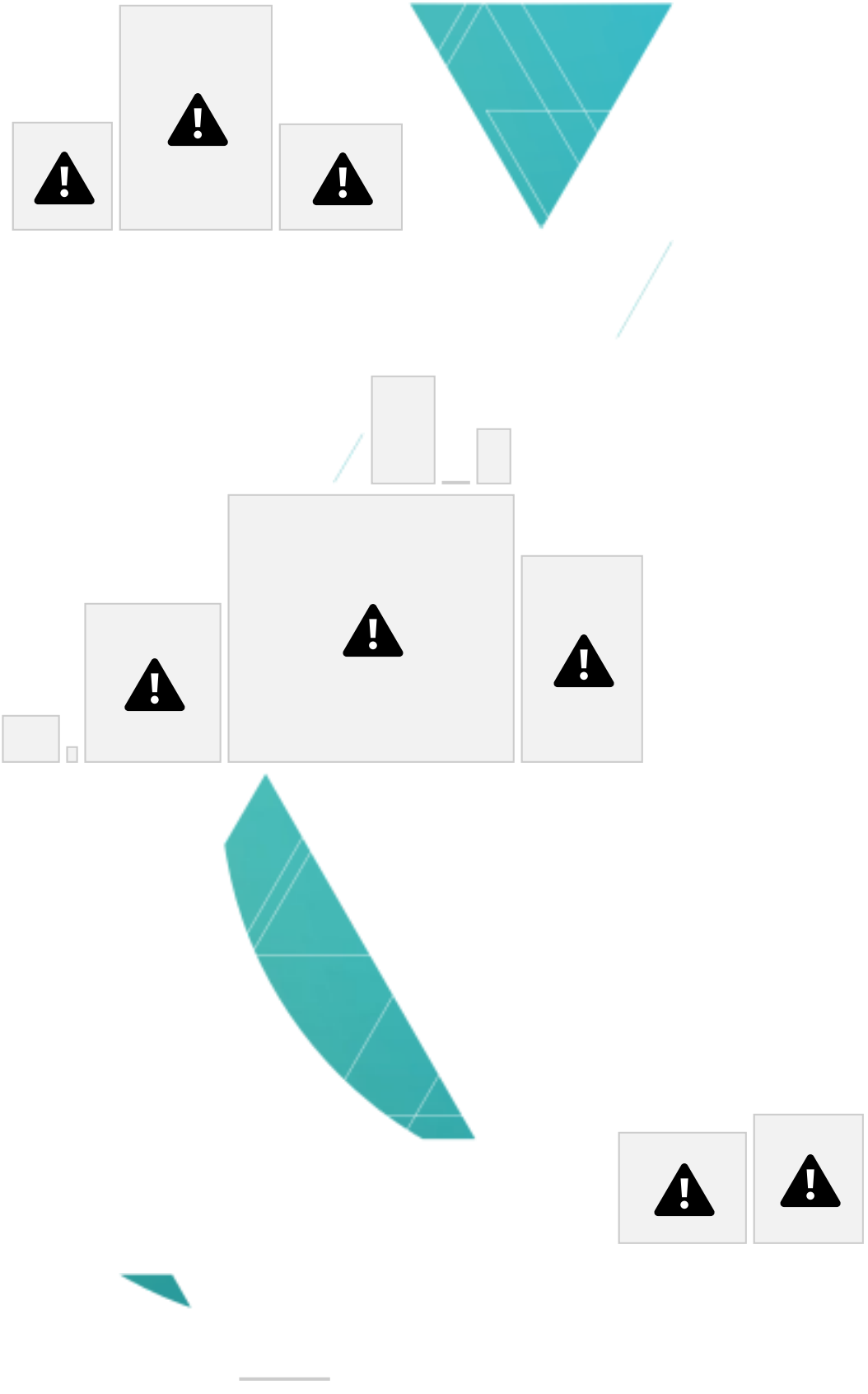


Mastering Machine Learning with Python in Six Steps

Manohar Swamynathan

Mastering Machine Learning with Python in Six Steps

A Practical Implementation Guide to Predictive Data Analytics Using Python

Manohar Swamynathan

Mastering Machine Learning with Python in Six Steps

Manohar Swamynathan
Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-2865-4 ISBN-13 (electronic): 978-1-4842-2866-1 DOI 10.1007/978-1-4842-2866-1

Library of Congress Control Number: 2017943522

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Celestin Suresh John
Development Editor: Anila Vincent and James Markham
Technical Reviewer: Jojo Moolayil
Coordinating Editor: Sanchita Mandal
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

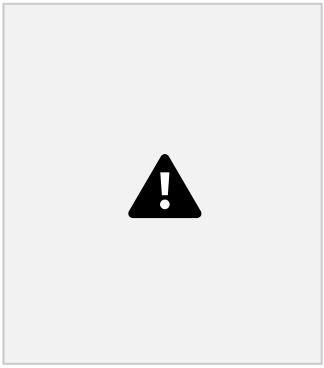
Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/ 978-1-4842-2865-4. For more detailed information, please visit http://www.apress.com/ source-code.

Printed on acid-free paper

Contents at a Glance

Contents

About the Author

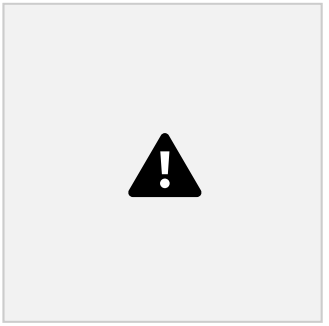


Manohar Swamynathan is a data science practitioner and an avid programmer, with over 13 years of experience in various data science-related areas that include data warehousing, Business Intelligence (BI), analytical tool development, ad hoc analysis, predictive modeling, data science product development, consulting, formulating strategy, and executing analytics program.

He's had a career covering life cycles of data across different domains such as U.S. mortgage banking, retail, insurance, and industrial IoT. He has a bachelor's degree with specialization in physics, mathematics, and computers; and a master's degree in project management. He's currently living in Bengaluru, the Silicon Valley of India, working as Staff Data Scientist with General Electric Digital, contributing to the next big digital industrial revolution.

You can visit him at http://www.mswamynathan.com to learn more about his various other activities.

About the Technical Reviewer



Jojo Moolayil is a Data Scientist and the author of the book: ***Smarter Decisions – The Intersection of Internet of Things and Decision Science***. With over 4 years of industrial experience in Data Science, Decision Science and IoT, he has worked with industry leaders on high impact and critical projects across multiple verticals. He is currently associated with **General Electric**, the pioneer and leader in data science for Industrial IoT and lives in Bengaluru—the silicon valley of India.

He was born and raised in Pune, India and graduated from University of Pune with a major in Information Technology Engineering. He started his career with Mu Sigma Inc., the world's largest pure play analytics provider and worked with the leaders of many Fortune 50 clients. One of the early enthusiasts to venture into IoT analytics, he converged his learnings from decision science to bring the problem solving frameworks and his learnings from data and decision science to IoT Analtyics.

To cement his foundations in data science for industrial IoT and scale the impact of the problem solving experiments, he joined a fast growing IoT Analytics startup called Flutura based in Bangalore and headquartered in the valley. After a

short stint with Flutura, Jojo moved on to work with the leaders of Industrial IoT - General Electric, in Bangalore, where he focused on solving decision science problems for Industrial IoT use cases. As a part of his role in GE, Jojo also focuses on developing data science and decision science products and platforms for Industrial IoT.

Apart from authoring books on Decision Science and IoT, Jojo has also been Technical Reviewer for various books on Machine Learning, Deep Learning and Business Analytics with Apress. He is an active Data Science tutor and maintains a blog at http://www.jojomoolayil.com/web/blog/.

Profile

http://www.jojomoolayil.com/

https://www.linkedin.com/in/jojo62000

I would like to thank my family, friends and mentors.

—Jojo Moolayil

Acknowledgments

I'm grateful to my mom, dad, and loving brother; I thank my wife Usha and son Jivin for providing me the space for writing this book.

I would like to express my gratitude to my mentors, colleagues, and friends from current/previous organizations for their inputs, inspiration, and support. Thanks to Jojo for the encouragement to write this book and his technical review inputs. Big thanks to the Apress team for their constant support and help.

Finally, I would like to thank you the reader for showing an interest in this book and sincerely hope to help your pursuit to machine learning quest.

Note that the views expressed in this book are author's personal.

Introduction

This book is your practical guide towards novice to master in machine learning with Python in six steps. The six steps path has been designed based on the “Six degrees of separation” theory that states that everyone and everything is a maximum of six steps away. Note that the theory deals with the quality of connections, rather than their existence. So a great effort has been taken to design eminent, yet simple six steps covering fundamentals to advanced topics gradually that will help a beginner walk his way from no or least knowledge of machine learning in Python to all the way to becoming a master practitioner. This book is also helpful for current Machine Learning practitioners to learn the advanced topics such as Hyperparameter tuning, various ensemble techniques, Natural Language Processing (NLP), deep learning, and the basics of reinforcement learning. See Figure 1.

Figure 1. *Learning Journey - Mastering Python Machine Learning: In Six Steps*

Each topic has two parts: the first part will cover the theoretical concepts and the second part will cover practical implementation with different Python packages. The traditional approach of math to machine learning, that is, learning all the mathematics then understanding how to implement it to solve problems needs a great deal of time/effort, which has proven to be not efficient for working professionals looking to switch careers. Hence the focus in this book has been more on simplification, such that the theory/math behind algorithms have been covered only to the extent required to get you started.

I recommend you work with the book instead of reading it. Real learning goes on only through active participation. Hence, all the code presented in the book is available in the form of iPython notebooks to enable you to try out these examples yourselves and extend them to your advantage or interest as required later.

Who This Book Is for

This book will serve as a great resource for learning machine learning concepts and implementation techniques for the following:

- Python developers or data engineers looking to expand their knowledge or career into the machine learning area.
- A current non-Python (R, SAS, SPSS, Matlab, or any other language) machine learning practitioners looking to expand their implementation skills in Python.
- Novice machine learning practitioners looking to learn advanced topics such as hyperparameter tuning, various ensemble techniques, Natural Language Processing (NLP), deep learning, and basics of reinforcement learning.

What You Will Learn

Chapter 1, Step 1 - Getting started in Python. This chapter will help you to set up the environment, and introduce you to the key concepts of Python programming language in relevance to machine learning. If you are already well versed with Python basics, I recommend you glance through the chapter quickly and move onto the next chapter.

Chapter 2, Step 2 - Introduction to Machine Learning. Here you will learn about the history, evolution, and different frameworks in practice for building machine learning systems. I think this understanding is very important as it will give you a broader perspective and set the stage for your further expedition. You'll understand the different types of machine learning (supervised / unsupervised / reinforcement learning). You will also learn the various concepts are involved in core data analysis packages (NumPy, Pandas, Matplotlib) with example codes.

Chapter 3, Step 3 - Fundamentals of Machine Learning This chapter will expose you to various fundamental concepts involved in feature engineering, supervised learning (linear regression, nonlinear regression, logistic regression, time series forecasting and classification algorithms), unsupervised learning (clustering techniques, dimension reduction technique) with the help of scikit-learn and statsmodel packages.

Chapter 4, Step 4 - Model Diagnosis and Tuning. in this chapter you'll learn advanced topics around different model diagnosis, which covers the common problems that arise, and various tuning techniques to overcome these issues to build efficient models. The topics include choosing the correct probability cutoff, handling an imbalanced dataset, the variance, and the bias issues. You'll also learn various tuning techniques such as ensemble models and hyperparameter tuning using grid / random search.

Chapter 5, Step 5 - Text Mining and Recommender System. Statistics says 70% of the data available in the business world is in the form of text, so text mining has vast scope across various domains. You will learn the building blocks and basic concepts to advanced NLP techniques. You'll also learn the recommender systems that are most commonly used to create personalization for customers.

Chapter 6, Step 6 - Deep and Reinforcement Learning. There has been a great advancement in the area of Artificial Neural Network (ANN) through deep learning techniques and it has been the buzzword in recent times. You'll learn various aspects of deep learning such as multilayer perceptrons, Convolution Neural Network (CNN) for image classification, RNN (Recurrent Neural Network) for text classification, and transfer learning. And you'll also learn the q-learning example to understand the concept of reinforcement learning.

Chapter 7, Conclusion. This chapter summarizes your six step learning and you'll learn quick tips that you should remember while starting with real-world machine learning problems.

CHAPTER 1

Step 1 – Getting Started in Python

In this chapter you will get a high-level overview of the Python language and its core philosophy, how to set up the Python development environment, and the key concepts around Python programming to get you started with basics. This chapter is

an additional step or the prerequisite step for non-Python users. If you are already comfortable with Python, I would recommend that you quickly run through the contents to ensure you are aware of all of the key concepts.

The Best Things in Life Are Free

As the saying goes, "*The best things in life are free!*" Python is an open source, high-level, object-oriented, interpreted, and general-purpose dynamic programming language. It has a community-based development model. Its core design theory accentuates code readability, and its coding structure enables programmers to articulate computing concepts in fewer lines of code as compared to other high-level programming languages such as Java, C or C++.

The design philosophy of Python is well summarized by the document "The Zen of Python" (Python Enhancement Proposal, information entry number 20), which includes mottos such as the following:

- Beautiful is better than ugly – be consistent.
- Complex is better than complicated – use existing libraries.
- Simple is better than complex – keep it simple and stupid

(KISS). • Flat is better than nested – avoid nested ifs.

- Explicit is better than implicit – be clear.
- Sparse is better than dense – separate code into modules.
- Readability counts – indenting for easy readability.
- Special cases aren't special enough to break the rules – everything is an object.
- Errors should never pass silently – good exception handler.

  M. Swamynathan, *Mastering Machine Learning with Python in Six Steps*,
DOI 10.1007/978-1-4842-2866-1_1

- Although practicality beats purity - if required, break the rules.
- Unless explicitly silenced – error logging and traceability.
- In ambiguity, refuse the temptation to guess – Python syntax is simpler; however, many times we might take a longer time to decipher it.
- Although that way may not be obvious at first unless you're Dutch – there is not only one of way of achieving something.
- There should be preferably only one obvious way to do it – use existing libraries.
- If the implementation is hard to explain, it's a bad idea – if you can't explain in simple terms then you don't understand it well enough.
- Now is better than never – there are quick/dirty ways to get the job done rather than trying too much to optimize.
- Although never is often better than *right* now – although there is a quick/dirty way, don't head in the path that will not allow a

graceful way back.

- Namespaces are one honking great idea, so let's do more of those! – be specific.
- If the implementation is easy to explain, it may be a good idea – simplicity.

The Rising Star

Python was officially born on February 20, 1991, with version number 0.9.0 and has taken a tremendous growth path to become the most popular language for the last 5 years in a row (2012 to 2016). Its application cuts across various areas such as website development, mobile apps development, scientific and numeric computing, desktop GUI, and complex software development. Even though Python is a more general-purpose programming and scripting language, it has been gaining popularity over the past 5 years among data scientists and Machine Learning engineers. See Figure 1-1.

Figure 1-1. *Popular Coding Language(Source: codeeval.com) and Popular Machine Learning Programming Language (Source:KDD poll)*

There are well-designed development environments such as IPython Notebook and Spyder that allow for a quick introspection of the data and enable developing of machine learning models interactively.

Powerful modules such as NumPy and Pandas exist for the efficient use of numeric data. Scientific computing is made easy with SciPy package. A number of primary machine learning algorithms have been efficiently implemented in scikit-learn (also known as sklearn). HadooPy, PySpark provides seamless work experience with big data technology stacks. Cython and Numba modules allow executing Python code in par with the speed of C code. Modules such as nosetest emphasize high-quality, continuous integration tests, and automatic deployment.

Combining all of the above has made many machine learning engineers embrace Python as the choice of language to explore data, identify patterns, and build and deploy models to the production environment. Most importantly the business-friendly licenses for various key Python packages are encouraging the collaboration of businesses and the open source community for the benefit of both worlds. Overall the Python programming ecosystem allows for quick results and happy programmers. We have been seeing the trend of developers being part of the open source community to contribute to the bug fixes and new algorithms for the use by the global community, at the same time protecting the core IP of the respective company they work for.

Python 2.7.x or Python 3.4.x?

Python 3.4.x is the latest version and comes with nicer, consistent functionalities! However, there is very limited third-party module support for it, and this will be the trend for at least a couple of more years. However, all major frameworks still run on version 2.7.x and are likely to continue to do so for a significant amount of time. Therefore, it is advised to start with Python 2, for the fact that it is the most widely used version for building machine learning systems as of today.

For an in-depth analysis of the differences between Python 2 vs. 3, you can refer to Wiki. python.org (https://wiki.python.org/moin/Python2orPython3v), which says that there are benefits to each.

I recommend Anaconda (Python distribution), which is BSD licensed and gives you permission to use it commercially and for redistribution. It has around 270 packages including the most important ones for most scientific applications, data analysis, and machine learning such as NumPy, SciPy, Pandas, IPython, matplotlib, and scikit-learn. It also provides a superior environment tool conda that allows you to easily switch between environments, even between Python 2 and 3 (if required). It is also updated very quickly as soon as a new version of a package is released and you can just use *conda update <packagename>* to update it.

You can download the latest version of Anaconda from their official website at https://www.continuum.io/downloads and follow the installation instructions.

Windows Installation

- Download the installer depending on your system configuration (32 or 64 bit).
- Double-click the .exe file to install Anaconda and follow the installation wizard on your screen.

OSX Installation

For Mac OS, you can install either through a graphical installer or from a command line.

Graphical Installer

- Download the graphical installer.
- Double-click the downloaded .pkg file and follow the installation wizard instructions on your screen.

Or

Command-Line Installer

- Download the command-line installer.
- In your terminal window type one of the below and follow the instructions: bash <Anaconda2-x.x.x-MacOSX-x86_64.sh>.

Linux Installation

- Download the installer depending on your system configuration (32 or 64 bit).
- In your terminal window type one of the below and follow the instructions: bash Anaconda2-x.x.x-Linux-x86_xx.sh.

Python from Official Website

For some reason if you don't want to go with the Anaconda build pack, alternatively you can go to Python's official website https://www.python.org/downloads/ and browse to the appropriate OS section and download the installer. Note that OSX and most of the Linux come with preinstalled Python so there is no need of additional configuring.

Setting up PATH for Windows: When you run the installer make sure to check the "Add Python to PATH option." This will allow us to invoke the Python interpreter from any directory. If you miss ticking "Add Python to PATH option," follow these instructions:

- Right-click on "My computer."
- Click "Properties."
- Click "Advanced system settings" in the side panel.

- Click "Environment Variables."
- Click the "New" below system variables.
- For the name, enter pythonexe (or anything you want).
- For the value, enter the path to your Python (example: C:\Python32\).
- Now edit the Path variable (in the system part) and add %pythonexe%; to the end of what's already there.

Running Python

From the command line, type "Python" to open the interactive interpreter. A Python script can be executed at the command line using the syntax here:

python <scriptname.py>

All the code used in this book are available as IPython Notebook (now known as the Jupyter Notebook), it is an interactive computational environment, in which you can combine code execution, rich text, mathematics, plots and rich media. You can launch the Jupyter Notebook by clicking on the icon installed by Anaconda in the start menu (Windows) or by typing 'jupyter notebook' in a terminal (cmd on Windows). Then browse for the relevant IPython Notebook file that you would like to paly with.

Note that the codes can break with change is package version, hence for reproducibility, I have shared my package version numbers, please refer Module_Versions IPython Notebook.

Key Concepts

There are a couple of fundamental concepts in Python, and understanding these are essential for a beginner to get started. A brief look at these concepts is to follow.

Python Identifiers

As the name suggests, identifiers help us to differentiate one entity from another. Python entities such as class, functions, and variables are called identifiers.

- It can be a combination of upper- or lowercase letters (a to z or A to Z).
- It can be any digits (0 to 9) or an underscore (_).
- The general rules to be followed for writing identifiers in Python.
- It cannot start with a digit. For example, 1variable is not valid, whereas variable1 is valid.
- Python reserved keywords (refer to Table 1-1) cannot be used as identifiers.
- Except for underscore (_), special symbols like !, @, #, $, % etc cannot be part of the identifiers.

Keywords

Table 1-1 lists the set of reserved words used in Python to define the syntax and structure of the language. Keywords are case sensitive, and all the keywords are in lowercase except *True, False,* and *None*.

Table 1-1. *Python keywords*

FALSE Class Finally Is return None Continue For Lambda try TRUE Def

From nonlocal while And Del Global Not with As Elif If Or yield Assert Else

Import Pass

Break Except In Raise

My First Python Program

Launch the Python interactive on the command prompt, and then type the following text and press Enter.

```
>>> print "Hello, Python World!"
```

If you are running Python 2.7.x from the Anaconda build pack, then you can also use the print statement with parentheses as in print (“Hello, Python World!”), which would produce the following result: Hello, Python World! See Figure 1-2.

Figure 1-2. *Python vs. Others*

Code Blocks (Indentation & Suites)

It is very important to understand how to write code blocks in Python. Let's look at two key concepts around code blocks.

Indentation

One of the most unique features of Python is its use of indentation to mark blocks of code. Each line of code must be indented by the same amount to denote a block of code in Python. Unlike most other programming languages, indentation is not used to help make the code look pretty. Indentation is required to indicate which block of code a code or statement belongs to.

Suites

A collection of individual statements that makes a single code block are called suites in Python. A header line followed by a suite are required for compound or complex statements such as *if, while, def,* and *class* (we will understand each of these in details in the later sections). Header lines begin with a keyword, and terminate with a colon (:) and are followed by one or more lines that make up the suite. See Listings 1-1 and 1-2.

Listing 1-1. Example of correct indentation

```
# Correct indentation
print ("Programming is an important skill for Data Science") print ("Statistics is a
important skill for Data Science") print ("Business domain knowledge is a
important skill for Data Science")

# Correct indentation, note that if statement here is an example of suites x = 1
if x == 1:
    print ('x has a value of 1')
else:
    print ('x does NOT have a value of 1')
```

Listing 1-2. Example of incorrect indentation

```
# incorrect indentation, program will generate a syntax error # due to the
space character inserted at the beginning of second line print ("Programming
```

```
is an important skill for Data Science")
  print ("Statistics is a important skill for Data Science") print ("Business domain
knowledge is a important skill for Data Science") 3
# incorrect indentation, program will generate a syntax error # due to
the wrong indentation in the else statement
x = 1
if x == 1:
    print ('x has a value of 1')
else:
  print ('x does NOT have a value of 1')
```


Basic Object Types

According to the Python data model reference, objects are Python's notion for data. All data in a Python program is represented by objects or by relations between objects. In a sense, and in conformance to Von Neumann's model of a "stored program computer," code is also represented by objects.Every object has an identity, a type, and a value. See Table 1-2 and Listing 1-3.

Table 1-2. *Python object types*

Type Examples Comments None None # singleton null object Boolean True, False

Integer -1, 0, 1, sys.maxint

Long 1L, 9787L

Float 3.141592654

inf, float('inf') # infinity

-inf # neg infinity

nan, float('nan') # not a number

Complex 2+8j # note use of j

String 'this is a string', "also me" # use single or double quote

r'raw string', b'ASCII string'

u'unicode string'

Tuple empty = () # empty tuple

(1, True, 'ML') # immutable list or unalterable list

List empty = [] empty list

[1, True, 'ML'] # mutable list or alterable list

Set empty = set() # empty set set(1, True, 'ML') # mutable or alterable

dictionary empty = {} # mutable object or alterable object
{'1':'A', '2':'AA', True = 1, False = 0}

File f = open('filename', 'rb')

Listing 1-3. Code For Basic Object Types

```
none = None # singleton null object
boolean = bool(True)
integer = 1
Long = 3.14

# float
Float = 3.14
Float_inf = float('inf')
Float_nan = float('nan')

# complex object type, note the usage of letter j
Complex = 2+8j

# string can be enclosed in single or double quote
string = 'this is a string'
me_also_string = "also me"

List = [1, True, 'ML'] # Values can be changed

Tuple = (1, True, 'ML') # Values can not be changed

Set = set([1,2,2,2,3,4,5,5]) # Duplicates will not be stored

# Use a dictionary when you have a set of unique keys that map to values
Dictionary = {'a':'A', 2:'AA', True:1, False:0}

# lets print the object type and the value
print type(none), none
print type(boolean), boolean
print type(integer), integer
print type(Long), Long
print type(Float), Float
print type(Float_inf), Float_inf
print type(Float_nan), Float_nan
print type(Complex), Complex
print type(string), string
print type(me_also_string), me_also_string
print type(Tuple), Tuple
print type(List), List
print type(Set), Set
print type(Dictionary), Dictionary

----- output ------
```

```
<type 'NoneType'> None
<type 'bool'> True
```



```
<type 'int'> 1
<type 'float'> 3.14
<type 'float'> 3.14
<type 'float'> inf
<type 'float'> nan
<type 'complex'> (2+8j)
<type 'str'> this is a string
<type 'str'> also me
<type 'tuple'> (1, True, 'ML')
<type 'list'> [1, True, 'ML']
<type 'set'> set([1, 2, 3, 4, 5])
<type 'dict'> {'a': 'A', True: 1, 2: 'AA', False: 0}
```

When to Use List vs. Tuples vs. Set vs. Dictionary

- *List*: Use when you need an ordered sequence of homogenous collections, whose values can be changed later in the program.
- *Tuple*: Use when you need an ordered sequence of heterogeneous collections whose values need not be changed later in the program.
- *Set*: It is ideal for use when you don't have to store duplicates and you are not concerned about the order or the items. You just want to know whether a particular value already exists or not.
- *Dictionary*: It is ideal for use when you need to relate values with keys, in order to look them up efficiently using a key.

Comments in Python

Single line comment: Any characters followed by the # (hash) and up to the end of the line are considered a part of the comment and the Python interpreter ignores them. Multiline comments: Any characters between the strings """ (referred as multiline string), that is, one at the beginning and end of your comments will be ignored by the Python interpreter. See Listing 1-4.

Listing 1-4. Example code for comments

```
# This is a single line comment in Python
print "Hello Python World" # This is also a single line comment in Python

""" This is an example of a multi line
comment that runs into multiple lines.
Everything that is in between is considered as comments
"""
```

Multiline Statement

Python's oblique line continuation inside parentheses, brackets, and braces is the favorite way of casing longer lines. Using backslash to indicate line continuation makes readability better; however if needed you can add an extra pair of parentheses around the expression. It is important to correctly indent the continued line of your code. Note that the preferred place to break around the binary operator is after the operator, and not before it. See Listing 1-5.

Listing 1-5. Example code for multiline statements

```
# Example of implicit line continuation
x = ('1' + '2' +
    '3' + '4')

# Example of explicit line continuation
y = '1' + '2' + \
    '11' + '12'

weekdays = ['Monday', 'Tuesday', 'Wednesday',
 'Thursday', 'Friday']

weekend = {'Saturday',
          'Sunday'}

print ('x has a value of', x)
print ('y has a value of', y)
print days
print weekend

------ output -------
('x has a value of', '1234')
('y has a value of', '1234')
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
set(['Sunday', 'Saturday'])
```

Multiple Statements on a Single Line

Python also allows multiple statements on a single line through usage of the semicolon (;), given that the statement does not start a new code block. See Listing 1-6.

Listing 1-6. Code example for multistatements on a single line

```
import os; x = 'Hello'; print x
```

Basic Operators

In Python, operators are the special symbols that can manipulate the value of operands. For example, let's consider the expression 1 + 2 = 3. Here, 1 and 2 are called operands, which are the value on which operators operate and the symbol + is called an operator. Python language supports the following types of operators.

- Arithmetic Operators
- Comparison or Relational Operators
- Assignment Operators
- Bitwise Operators
- Logical Operators
- Membership Operators
- Identity Operators

Let's learn all operators through examples one by one.

Arithmetic Operators

Arithmetic operators are useful for performing mathematical operations on numbers such as addition, subtraction, multiplication, division, etc. See Table 1-3 and then Listing 1-7.

Table 1-3. *Arithmetic operators*

Operator Description Example + Addition x + y = 30 - Subtraction x –

y = -10 * Multiplication x * y = 200

/ Division y / x = 2 % Modulus y % x = 0 ** Exponent Exponentiation x**b =10 to the power 20

// Floor Division – Integer division rounded toward minus infinity 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0/

Listing 1-7. Example code for arithmetic operators

```
# Variable x holds 10 and variable y holds 5
x = 10
y = 5

# Addition
print "Addition, x(10) + y(5) = ", x + y

# Subtraction
print "Subtraction, x(10) - y(5) = ", x - y

# Multiplication
print "Multiplication, x(10) * y(5) = ", x * y

# Division
print "Division, x(10) / y(5) = ",x / y

# Modulus
print "Modulus, x(10) % y(5) = ", x % y

# Exponent
print "Exponent, x(10)**y(5) = ", x**y

# Integer division rounded towards minus infinity
print "Floor Division, x(10)//y(5) = ", x//y

-------- output --------

Addition, x(10) + y(5) =  15
Subtraction, x(10) - y(5) =  5
Multiplication, x(10) * y(5) =  50
Divions, x(10) / y(5) =  2
Modulus, x(10) % y(5) =  0
Exponent, x(10)**y(5) =  100000
Floor Division, x(10)//y(5) =  2
```

Comparison or Relational Operators

As the name suggests the comparison or relational operators are useful to compare values. It would return True or False as a result for a given condition. See Table 1-4 and Listing 1-8.

Table 1-4. *Comparison or Relational operators*

Operator Description Example

== The condition becomes True, if the values of two operands are equal.

!= The condition becomes True, if the values of two operands are not equal.

<> The condition becomes True, if values of two operands are not equal.

> The condition becomes True, if the value of left operand is greater than the value of right operand.

< The condition becomes True, if the value of left operand is less than the value of right operand.

>= The condition becomes True, if the value of left operand is greater than or equal to the value of right operand.

<= The condition becomes True, if the value of left operand is less than or equal to the value of right operand.

Listing 1-8. Example code for comparision/relational operators

```
# Variable x holds 10 and variable y holds 5 x = 10
y = 5

# Equal check operation
print "Equal check, x(10) == y(5) ", x == y

# Not Equal check operation
print "Not Equal check, x(10) != y(5) ", x != y

# Not Equal check operation
print "Not Equal check, x(10) <>y(5) ", x<>y

# Less than check operation
print "Less than check, x(10) <y(5) ", x<y

# Greater check operation
print "Greater than check, x(10) >y(5) ", x>y

# Less than or equal check operation
print "Less than or equal to check, x(10) <= y(5) ", x<= y
```

(x == y) is not true.

(x<>y) is true. This is similar to != operator.

(x>y) is not true. (x<y) is true.

(x>= y) is not true. (x<= y) is true.

```
# Greater than or equal to check operation
print "Greater than or equal to check, x(10) >= y(5) ", x>= y

-------- output --------
Equal check, x(10) == y(5)  False
Not Equal check, x(10) != y(5)  True
Not Equal check, x(10) <>y(5)  True
Less than check, x(10) <y(5)  False
Greater than check, x(10) >y(5)  True
Less than or equal to check, x(10) <= y(5)  False
Greater than or equal to check, x(10) >= y(5)  True
```

Assignment Operators

In Python, assignment operators are used for assigning values to variables. For example, consider x = 5; it is a simple assignment operator that assigns the numeric value 5, which is on the right side of the operator onto the variable x on the left side of operator. There is a range of compound operators in Python like x += 5 that add to the variable and later assign the same. It is as good as x = x + 5. See Table 1-5 and Listing 1-9.

Table 1-5. *Assignment operators*

Operator	Description	Example
=	Assigns values from right side operands to left side operand.	
+=	Add AND	It adds right operand to the left operand and assigns the result to left

operand.

-= Subtract AND It subtracts right operand from the left operand and assigns the result to left operand.

*= Multiply AND It multiplies right operand with the left operand and assigns the result to left operand.

/= Divide AND It divides left operand with the right operand and assigns the result to left operand.

%= Modulus AND It takes modulus using two operands and assigns the result to left operand.

**= Exponent AND Performs exponential (power) calculation on operators and assigns value to the left operand.

//= Floor Division It performs floor division on operators and assigns value to the left operand.

z = x + y assigns value of x + y into z

z += x is equivalent to z = z + x

z -= x is equivalent to z = z - x

z *= x is equivalent to z = z * x

z /= x is equivalent to z = z/ xz /= x is equivalent to z = z / x

z %= x is equivalent to z = z % x

z **= x is equivalent to z = z ** x

z //= x is equivalent to z = z// x

Listing 1-9. Example code for assignment operators

```
# Variable x holds 10 and variable y holds 5
x = 5
y = 10

x += y
print "Value of a post x+=y is ", x

x *= y
print "Value of a post x*=y is ", x

x /= y
print "Value of a post x/=y is ", x

x %= y
print "Value of a post x%=y is ", x

x **= y
print "Value of x post x**=y is ", x

x //= y
print "Value of a post x//=y is ", x

-------- output --------

Value of a post x+=y is  15
Value of a post x*=y is  150
Value of a post x/=y is  15
Value of a post x%=y is  5
Value of a post x**=y is  9765625
```

Value of a post x//=y is 976562

Bitwise Operators

As you might be aware, everything in a computer is represented by bits, that is, a series of 0's (zero) and 1's (one). Bitwise operators enable us to directly operate or manipulate bits. Let's understand the basic bitwise operations. One of the key usages of bitwise operators is for parsing hexadecimal colors.

Bitwise operators are known to be confusing for newbies to Python programming, so don't be anxious if you don't understand usability at first. The fact is that you aren't really going to see bitwise operators in your everyday machine learning programming. However, it is good to be aware about these operators.

For example let's assume that x = 10 (in binary 0000 1010) and y = 4 (in binary 0000 0100). See Table 1-6 and Listing 1-10.

Table 1-6. *Bitwise operators*

Operator	Description	Example
&	Binary AND This operator copies a bit to the result if it exists in both operands.	bits specified by the right operand.
|	Binary OR This operator copies a bit if it exists in either operand.	
^	Binary XOR This operator copies the bit if it is set in one operand but not both.	
~	Binary Ones Complement This operator is unary and has the effect of 'flipping' bits.	
<<	Binary Left Shift The left operands value is moved left by the number of bits specified by the right operand.	
>>	Binary Right Shift The left operands value is moved right by the number of	

Listing 1-10. Example code for bitwise operators

```
# Basic six bitwise operations
(x&y) (means  0000 0000)
(x | y) = 14
(means 0000  1110)
(x ^ y) = 14
(means 0000  1110)
(~x ) = -11
(means 1111  0101)
x<< 2= 42
(means 0010  1000)
x>> 2 = 2 (means  0000 0010)
```

```
# Let x = 10 (in binary0000 1010) and y = 4 (in binary0000 0100) x = 10
y = 4

print x >> y  # Right Shift
print x << y  # Left Shift
print x & y   # Bitwise AND
```

```
print x | y   # Bitwise OR
print x ^ y # Bitwise XOR
print ~x    # Bitwise NOT

-------- output --------

0
160
0
14
14
-11
```


Logical Operators

The AND, OR, NOT operators are called logical operators. These are useful to check two variables against given condition and the result will be True or False appropriately. See Table 1-7 and Listing 1-11.

Table 1-7. *Logical operators*

Operator	Description	Example
and	Logical AND If both the operands are true then condition becomes true.	(var1 and var2) is true.
or	Logical OR If any of the two operands are non-zero then condition becomes true.	(var1 or var2) is true.
not	Logical NOT Used to reverse the logical state of its operand.	Not (var1 and var2) is false.

Listing 1-11. Example code for logical operators

```
var1 = True
var2 = False
print('var1 and var2 is',var1and var2)
print('var1 or var2 is',var1 or var2)
print('not var1 is',not var1)

-------- output --------

('var1 and var2 is', False)
('var1 or var2 is', True)
('not var1 is', False)
```

Membership Operators

Membership operators are useful to test if a value is found in a sequence, that is, string, list, tuple, set, or dictionary. There are two membership operators in Python, '*in*' and *'not in'*. Note that we can only test for presence of key (and not the value) in case of a dictionary. See Table 1-8 and Listing 1-12.

Table 1-8. *Membership operators*

Operator	Description	Example
in	Results to True if a value is in the specified sequence and False otherwise.	var1 in var2
not in	Results to True, if a value is not in the specified sequence and False otherwise.	var1 not in var2

Listing 1-12. Example code for membership operators

```
var1 = 'Hello world'          # string
var1 = {1:'a',2:'b'}# dictionary
print('H' in var1)
print('hello' not in var2)
print(1 in var2)
print('a' in var2)

-------- output --------
True
True
True
False
```

Identity Operators

Identity operators are useful to test if two variables are present on the same part of

the memory. There are two identity operators in Python, '*is*' and '*is not*'. Note that two variables having equal values do not imply they are identical. See Table 1-9 and Listing 1-13.

Table 1-9. *Identity operators*

Operator	Description	Example
is	Results to True, if the variables on either side of the operator point to the same object and False otherwise.	var3 = [1,2,3] var3 = [1,2,3] print(var1 is not var1) print(var2 is var2) print(var3 is var3) -------- output -------- False True False
is not	Results to False, if the variables on either side of the operator point to the same object and True otherwise.	var1 is var2 Var1 is not var2

Listing 1-13. Example code for identity operators

```
var1 = 5
var1 = 5
var2 = 'Hello'
var2 = 'Hello'
```


Control Structure

A control structure is the fundamental choice or decision-making process in programming. It is a chunk of code that analyzes values of variables and decides a direction to go based on a given condition. In Python there are mainly two types of control structures: (1) selection and (2) iteration.

Selection

Selection statements allow programmers to check a condition and based on the

result will perform different actions. There are two versions of this useful construct: (1) if and (2) if…else. See Listings 1-14, 1-15, and 1-16.

Listing 1-14. Example code for a simple 'if' statement

```
var = -1
if var < 0:
   print var
   print("the value of var is negative")

# If there is only a signle cluse then it may go on the same line as the  header
statement
if ( var  == -1 ) : print "the value of var is negative"
```

Listing 1-15. Example code for 'if else' statement

```
var = 1

if var < 0:
   print "the value of var is negative"
   print var
else:
   print "the value of var is positive"
   print var
```

Listing 1-16. Example code for nested if else statements

```
Score = 95

if score >= 99:
   print('A')
elif score >=75:
   print('B')
elif score >= 60:
   print('C')
elif score >= 35:
   print('D')
else:
   print('F')
```


Iteration

A loop control statement enables us to execute a single or a set of programming statements multiple times until a given condition is satisfied. Python provides two essential looping statements: (1) for (2) while statement.

For loop: It allows us to execute code block for a specific number of times or against a specific condition until it is satisfied. See Listing 1-17.

Listing 1-17. Example codes for a 'for loop' statement

```
# First Example
print "First Example"
for item in [1,2,3,4,5]:
   print 'item :', item
```

```
# Second Example
print "Second Example"
letters = ['A', 'B', 'C']
for letter in letters:
    print ' First loop letter :', letter

# Third Example - Iterating by sequency index
print "Third Example"
for index in range(len(letters)):
    print 'First loop letter :', letters[index]

# Fourth Example - Using else statement
print "Fourth Example"
for item in [1,2,3,4,5]:
    print 'item :', item
else:
    print 'looping over item complete!'
----- output ------
First Example
item : 1
item : 2
item : 3
item : 4
item : 5
Second Example
First loop letter : A
First loop letter : B
First loop letter : C
Third Example
First loop letter : A
First loop letter : B
First loop letter : C
Fourth Example
```



```
item : 1
item : 2
item : 3
item : 4
item : 5
looping over item complete!
```

While loop: The while statement repeats a set of code until the condition is true. See Listing 1-18.

Listing 1-18. Example code for while loop statement

```
count = 0
while (count <3):
   print 'The count is:', count
   count = count + 1
```

■ **Caution** If a condition never becomes FALSE, a loop becomes an infinite loop.

An else statement can be used with a while loop and the else will be executed when the condition becomes false. See Listing 1-19.

Listing 1-19. example code for a 'while with a else' statement

```
count = 0
while count <3:
   print count, " is  less than 5"
   count = count + 1
else:
   print count, " is not less than 5"
```

Lists

Python's lists are the most flexible data type. It can be created by writing a list of comma separated values between square brackets. Note that that the items in the list need not be of the same data type. See Table 1-10; and Listings 1-20, 1-21, 1-22, 1-23, and 1-24.

Table 1-10. *Python list operations*

Description Python Expression Example Results Creating a list of items [item1, item2, …] list = ['a','b','c','d'] ['a','b','c','d']

Accessing items in list list[index] list = ['a','b','c','d'] c list[2]

Length len(list) len([1, 2, 3]) 3 Concatenation list_1 + list_2 [1, 2, 3] + [4, 5, 6] [1, 2, 3, 4, 5, 6]

Repetition list * int ['Hello'] * 3 ['Hello', 'Hello', 'Hello']

Membership item in list 3 in [1, 2, 3] TRUE

Iteration for x in list: print x for x in [1, 2, 3]: print x, 1 2 3

Count from the right list[-index] list = [1,2,3]; list[-2] 2 Slicing fetches

sections list[index:] list = [1,2,3]; list[1:] [2,3]

Comparing lists cmp(list_1, list_2) print cmp([1,2,3,4], 1 -1 [5,6,7]); print cmp([1,2,3], [5,6,7,8])

Return max item max(list) max([1,2,3,4,5]) 5 Return min item min(list) max([1,2,3,4,5]) 1 Append object to list list.append(obj) [1,2,3,4].append(5) [1,2,3,4,5] Count item occurrence list.count(obj) [1,1,2,3,4].count(1) 2

Append content of sequence to list list.extend(seq) [‘a’, 1].extend([‘b’, 2]) [‘a’, 1, ‘b’, 2]

Return the first index position of item — list.index(obj) [‘a’, ‘b’,‘c’,1,2,3]. index(‘c’) — object from list — ‘b’,‘c’,1,2,3].pop() [‘a’, ‘b’,‘c’,1,2,3].pop(2) 2

Insert object to list at a desired index — list.insert(index, obj) [‘a’, ‘b’,‘c’,1,2,3]. insert(4, ‘d’) — [‘a’, ‘b’,‘c’,‘d’, 1,2,3]

Remove and return last — list.pop(obj=list[-1]) [‘a’, ‘b’,‘c’,1,2,3]. remove(‘c’) — 3 c [‘a’, ‘b’, 1,2,3]

Remove object from list list.remove(obj)

Reverse objects of list in place — list.reverse() [‘a’, ‘b’,‘c’,1,2,3]. reverse() — [3,2,1,‘c’,‘b’,‘a’]

Sort objects of list list.sort() [‘a’, ‘b’,‘c’,1,2,3].sort() [‘a’, ‘b’,‘c’,1,2,3]. sort(reverse = True) — [1,2,3,‘a’, ‘b’,‘c’] [‘c’,‘b’,‘a’,3,2,1]

Listing 1-20. Example code for accessing lists

```
# Create lists
list_1 = ['Statistics', 'Programming', 2016, 2017, 2018]; list_2 =
['a', 'b', 1, 2, 3, 4, 5, 6, 7 ];

# Accessing values in lists
print "list_1[0]: ", list_1[0]
print "list2_[1:5]: ", list_2[1:5]

---- output ----

list_1[0]:  Statistics
list2_[1:5]:  ['b', 1, 2, 3]
```

Listing 1-21. Example code for adding new values to

lists print "list_1 values: ", list_1

```
# Adding new value to list
list_1.append(2019)
```

```
print "list_1 values post append: ", list_1
---- output ----
list_1 values:  ['c', 'b', 'a', 3, 2, 1]
list_1 values post append:  ['c', 'b', 'a', 3, 2, 1, 2019]
```

Listing 1-22. Example code for updating existing values of

```
lists print "Values of list_1: ", list_1

# Updating existing value of list
print "Index 2 value : ", list_1[2]
list_1[2] = 2015;
print "Index 2's new value : ", list_1[2]

---- output ----

Values of list_1:  ['c', 'b', 'a', 3, 2, 1, 2019] Index 2 value
:  a
Index 2's new value :  2015
```

Listing 1-23. Example code for deleting a list

```
element Print "list_1 values: ", list_1

# Deleting list element
del list_1[5];
print "After deleting value at index 2 : ", list_1 ---- output

----
```



```
list_1 values:  ['c', 'b', 2015, 3, 2, 1, 2019]
After deleting value at index 2 :  ['c', 'b', 2015, 3, 2, 2019]
```

Listing 1-24. Example code for basic operations on lists

```
print "Length: ", len(list_1)
print "Concatenation: ", [1,2,3] + [4, 5, 6]
print "Repetition :", ['Hello'] * 4
print "Membership :", 3 in [1,2,3]
print "Iteration :"
for x in [1,2,3]: print x

# Negative sign will count from the right
print "slicing :", list_1[-2]
# If you dont specify the end explicitly, all elements from the specified  start index
will be printed
print "slicing range: ", list_1[1:]

# Comparing elements of lists
print "Compare two lists: ", cmp([1,2,3, 4], [1,2,3])
print "Max of list: ", max([1,2,3,4,5])
print "Min of list: ", min([1,2,3,4,5])
print "Count number of 1 in list: ", [1,1,2,3,4,5,].count(1)
list_1.extend(list_2)
```

```
print "Extended :", list_1
print "Index for Programming : ", list_1.index( 'Programming') print
list_1
print "pop last item in list: ", list_1.pop()
print "pop the item with index 2: ", list_1.pop(2)
list_1.remove('b')
print "removed b from list: ", list_1
list_1.reverse()
print "Reverse: ", list_1
list_1 = ['a', 'b','c',1,2,3]
list_1.sort()
print "Sort ascending: ", list_1
list_1.sort(reverse = True)
print "Sort descending: ", list_1

---- output ----

Length:  5
Concatenation:  [1, 2, 3, 4, 5, 6]
Repetition : ['Hello', 'Hello', 'Hello', 'Hello']
Membership : True
Iteration :
1
2
```



```
3
slicing : 2017
slicing range:  ['Programming', 2015, 2017, 2018]
Compare two lists:  1
Max of list:  5
Min of list:  1
Count number of 1 in list:  2
Extended : ['Statistics', 'Programming', 2015, 2017, 2018, 'a', 'b', 1, 2,  3, 4, 5, 6, 7]
Index for Programming :  1
['Statistics', 'Programming', 2015, 2017, 2018, 'a', 'b', 1, 2, 3, 4, 5, 6, 7] pop last item
in list:  7
pop the item with index 2:  2015
removed b from list:  ['Statistics', 'Programming', 2017, 2018, 'a', 1, 2,  3, 4, 5, 6]
Reverse:  [6, 5, 4, 3, 2, 1, 'a', 2018, 2017, 'Programming', 'Statistics'] Sort
ascending:  [1, 2, 3, 'a', 'b', 'c']
Sort descending:  ['c', 'b', 'a', 3, 2, 1]
```

Tuple

A Python tuple is a sequences or series of immutable Python objects very much similar to the lists. However there exist some essential differences between lists and tuples, which are the following. See also Table 1-11; and Listings 1-25, 1-26, 1-27, and 1-28.

1. Unlike list, the objects of tuples cannot be changed.
2. Tuples are defined by using parentheses, but lists are defined

by square brackets.

Table 1-11. *Python Tuple operations*

Description	Python Expression	Example	Results
Creating a tuple with one item, note comma is required	(item1, item2, …) # empty tuple (item1,) # Tuple	tuple = (1,) (‘a’,‘b’,‘c’,‘d’,1,2,3) tuple = (‘a’,‘b’,‘c’, ‘d’,1,2,3) tuple = ()	() 1
Accessing items	in tuple tuple[index] tuple[start_index: end_index]	tuple = (‘a’,‘b’,‘c’, ‘d’,1,2,3) tuple[2]	tuple[0:2] c a, b, c

Deleting a tuple del tuple_name del tuple

Length len(tuple) len((1, 2, 3)) 3 Concatenation tuple_1 + tuple_2 (1, 2, 3) +

(4, 5, 6) (1, 2, 3, 4, 5, 6)

Repetition tuple * int (‘Hello’,) * 4 (‘Hello’, ‘Hello’, ‘Hello’, ‘Hello’)

Membership item in tuple 3 in (1, 2, 3) TRUE
Iteration for x in tuple: print x for x in (1, 2, 3): print x tuple = (1,2,3); list[-2] 1 2 3 2

Count from the right tuple[-index]

Slicing fetches sections tuple[index:] tuple = (1,2,3); list[1:] (2,3)

Comparing lists cmp(tuple_1, tuple_2) print cmp((1,2,3,4), (5,6,7)); print cmp((1,2,3), (5,6,7,8)) 1 -1

Return max item max(tuple) max((1,2,3,4,5)) 5 Return min item min(tuple)

max((1,2,3,4,5)) 1 Convert a list to tuple tuple(seq) tuple([1,2,3,4])
(1,2,3,4,5)

Listing 1-25. Example code for creating tuple

```
# Creating a tuple

Tuple = ()
print "Empty Tuple: ", Tuple

Tuple = (1,)
print "Tuple with single item: ", Tuple
```



```
Tuple = ('a','b','c','d',1,2,3)
print "Sample Tuple :", Tuple

---- output ----
Empty Tuple:  ()
Tuple with single item:  (1,)
Sample Tuple : ('a', 'b', 'c', 'd', 1, 2, 3)
```

Listing 1-26. Example code for accessing tuple

```
# Accessing items in tuple
Tuple = ('a', 'b', 'c', 'd', 1, 2, 3)

print "3rd item of Tuple:", Tuple[2]
print "First 3 items of Tuple", Tuple[0:2]

---- output ----
3rd item of Tuple: c
First 3 items of Tuple ('a', 'b')
```

Listing 1-27. Example code for deleting tuple

```
# Deleting tuple

print "Sample Tuple: ", Tuple
del Tuple
print Tuple # Will throw an error message as the tuple does not exist ----

output ----

Sample Tuple:  ('a', 'b', 'c', 'd', 1, 2, 3)
---------------------------------------------------------------------------
NameError                 Traceback (most recent call last)
<ipython-input-35-6a0deb3cfbcf> in <module>()
     3 print "Sample Tuple: ", Tuple
     4 del Tuple
----> 5 print Tuple # Will throw an error message as the tuple does not exist
```

NameError: name 'Tuple' is not defined

Listing 1-28. Example code for basic operations on tupe (not exhaustive)

```
# Basic Tuple operations
Tuple = ('a','b','c','d',1,2,3)

print "Length of Tuple: ", len(Tuple)

Tuple_Concat = Tuple + (7,8,9)
print "Concatinated Tuple: ", Tuple_Concat
```



```
print "Repetition: ", (1, 'a',2, 'b') * 3
print "Membership check: ", 3 in (1,2,3)

# Iteration
for x in (1, 2, 3): print x

print "Negative sign will retrieve item from right: ", Tuple_Concat[-2] print "Sliced
Tuple [2:] ", Tuple_Concat[2:]

# Comparing two tuples
print "Comparing tuples (1,2,3) and (1,2,3,4): ", cmp((1,2,3), (1,2,3,4)) print
"Comparing tuples (1,2,3,4) and (1,2,3): ", cmp((1,2,3,4), (1,2,3))

# Find max
print "Max of the Tuple (1,2,3,4,5,6,7,8,9,10): ",
max((1,2,3,4,5,6,7,8,9,10))
print "Min of the Tuple (1,2,3,4,5,6,7,8,9,10): ",
min((1,2,3,4,5,6,7,8,9,10))
print "List [1,2,3,4] converted to tuple: ", type(tuple([1,2,3,4])) ---- output ----

  Length of Tuple:  7
Concatinated Tuple:  ('a', 'b', 'c', 'd', 1, 2, 3, 7, 8, 9) Repetition:  (1,
'a', 2, 'b', 1, 'a', 2, 'b', 1, 'a', 2, 'b') Membership check:  True
1
2
3
Negative sign will retrieve item from right:  8
Sliced Tuple [2:]  ('c', 'd', 1, 2, 3, 7, 8, 9)
Comparing tuples (1,2,3) and (1,2,3,4):  -1
Comparing tuples (1,2,3,4) and (1,2,3):  1
Max of the Tuple (1,2,3,4,5,6,7,8,9,10):  10
Min of the Tuple (1,2,3,4,5,6,7,8,9,10):  1
 List [1,2,3,4] converted to tuple:  <type 'tuple'>
```

Sets

As the name implies, sets are the implementations of mathematical sets. Three key characteristic of set are the following.

1. The collection of items is unordered.

2. No duplicate items will be stored, which means that each item is unique.

3. Sets are mutable, which means the items of it can be changed.

An item can be added or removed from sets. Mathematical set operations such as union, intersection, etc., can be performed on Python sets. See Table 1-12 and Listing 1-29.

Table 1-12. *Python set operations*

Description	Python Expression	Example Results
Creating a set. set{item1, item2, …} set() # empty set languages = set(['Python', 'R', 'SAS', 'Julia'])	set(['SAS', 'Python', 'R', 'Julia'])	add() languages. add('SPSS') set(['SAS', 'SPSS', 'Python', 'R', 'Julia'])
Add an item/ element to a set.		
Remove all items/ elements from a set.	clear() languages.clear() set([])	
Return a copy of a set.	set(['C', 'Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'AI', 'R', 'SAS', 'Machine Learning'])	'Python', 'Data Science', 'Julia', 'SPSS', 'R', 'SAS', 'Machine Learning'])
Remove an item/ element from set if it is a member. (Do nothing if the element is not in set).	languages. discard('AI')	
Remove an item/ element from a set. If the element is not a member, raise a KeyError.	remove() languages = set(['C', 'Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'AI', 'R', 'SAS', 'Machine Learning']) languages. remove('AI')	set(['C', 'Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'R', 'SAS', 'Machine Learning'])
Remove and return an arbitrary set element. RaiseKeyError if the set is empty. copy() lang = languages. copy() print lang	pop() languages = set(['C', 'Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'AI', 'R', 'SAS', 'Machine Learning']) print "Removed:", (languages.pop()) print(languages) set(['SAS', 'SPSS', 'Python', 'R', 'Julia'])	Removed: C set(['Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'R', 'SAS', 'Machine Learning'])
discard() languages =	set(['C', 'Java',	

(*continued*)

Table 1-12. (*continued*)

Description	Python Expression	Example	Results
Return the difference of two or more sets as a new set.	difference() # initialize A and B A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8} A.difference(B)		A.intersection(B) {1, 2, 3}
Remove all item/ elements of another set from this set.	difference_update() # initialize A and B A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8} A.difference_ update(B) print A		set([1, 2, 3]) {4, 5}
Return the intersection of two sets as a new set.	intersection() # initialize A and B A = {1, 2, 3, 4,		
Update the set with the intersection of	itself and another. intersection_ update()	# initialize A and B A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8}	A.intersection_ update B print A set([4, 5])
			A.issubset(B)
	Return True if this set contains another set.		issuperset() # initialize A and B A = {1, 2, 3,
Return True if two sets have a null intersection.	isdisjoint() # initialize A and B A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8} A.isdisjoint(B)		4, 5} B = {4, 5, 6, 7, 8} print A.issuperset(B) FALSE FALSE FALSE
Return True if another set contains this set.	issubset() # initialize A and B A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8} print		
Return the symmetric difference of two sets as a new set.	difference() # initialize A and B A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8} A. symmetric_ difference(B)		{1, 2, 3, 6, 7, 8}

(*continued*)

Table 1-12. (*continued*)

Description	Python Expression	Example	Results

Update a set with the symmetric difference of itself and another.	symmetric_ difference_update() # initialize A and B	5} B = {4, 5, 6, 7, 8} A.symmetric_ difference(B) print	A A.symmetric_ difference_update (B) print A set([1, 2, 3, 6, 7, 8])
	A = {1, 2, 3, 4, B = {4, 5, 6, 7, 8}	B A = {1, 2, 3, 4, 5}	set([1, 2, 3, 4, 5])
	A.update(B) print A		
	len() A = {1, 2, 3, 4, 5} len(A)		set([1, 2, 3, 4, 5, 6, 7, 8])
Return the union of sets in a new set.	max() A = {1, 2, 3, 4, 5} max(A)		5
	min() A = {1, 2, 3, 4, 5} min(A)		1
Update a set with the union of itself and others.			5
Return the length (the number of items) in the set.			[4, 5, 6, 7, 8]
Return the largest item in the set.			15
Return the smallest item in the set.			
Return a new sorted list from elements in the set. Does not sort the set.			
Return the sum of all items/elements in the set.	sorted() A = {1, 2, 3, 4, 5} sorted(A)		
union() # initialize A and B A = {1, 2, 3, 4, 5} B = {4, 5, 6, 7, 8}			
A.union(B) print A	sum() A = {1, 2, 3, 4, 5} sum(A)		
update() # initialize A and			

Listing 1-29. Example code for creating sets

```
# Creating an empty set
languages = set()
print type(languages), languages

languages = {'Python', 'R', 'SAS', 'Julia'} print
type(languages), languages
```

```
# set of mixed datatypes
mixed_set = {"Python", (2.7, 3.4)}
print type(mixed_set), languages
---- output ----
<type 'set'> set([])
<type 'set'> set(['SAS', 'Python', 'R', 'Julia'])
<type 'set'> set(['SAS', 'Python', 'R', 'Julia'])
```

Accessing Set Elements

See Listing 1-30.

Listing 1-30. Example code for accessing set elements

```
print list(languages)[0]
print list(languages)[0:3]
---- output ----
C
['C', 'Java', 'Python']
```

Changing a Set in Python

Although sets are mutable, indexing on them will have no meaning due to the fact that they are unordered. So sets do not support accessing or changing an item/element using indexing or slicing. The add() method can be used to add a single element and the update() method for adding multiple elements. Note that the update() method can take the argument in the format of tuples, lists, strings, or other sets. However, in all cases the duplicates are ignored. See Listing 1-31.

Listing 1-31. Example code for changing set elements

```
# initialize a set
languages = {'Python', 'R'}
print(languages)

# add an element
languages.add('SAS')
print(languages)
# add multiple elements
languages.update(['Julia','SPSS'])
print(languages)

# add list and set
languages.update(['Java','C'], {'Machine Learning','Data Science','AI'})
print(languages)
```



```
---- output ----
set(['Python', 'R'])
```

```
set(['Python', 'SAS', 'R'])
set(['Python', 'SAS', 'R', 'Julia', 'SPSS'])
set(['C', 'Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'AI', 'R',  'SAS', 'Machine
Learning'])
```

Removing Items from Set

The discard() or remove() method can be used to remove a particular item from a set. The fundamental difference between discard() and remove() is that the first do not take any action if the item does not exist in the set, whereas remove() will raise an error in such a scenario. See Listing 1-32.

Listing 1-32. Example code for removing items from set

```
# remove an element
languages.remove('AI')
print(languages)

# discard an element, although AI has already been removed discard will not throw
an error
languages.discard('AI')
print(languages)

# Pop will remove a random item from set
print "Removed:", (languages.pop()), "from", languages

---- output ----
set(['C', 'Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'R', 'SAS',  'Machine
Learning'])
set(['C', 'Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'R', 'SAS',  'Machine
Learning'])

Removed: C from set(['Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'R',  'SAS',
'Machine Learning'])
```

Set Operations

As discussed earlier, sets allow us to use mathematical set operations such as union, intersection, difference, and symmetric difference. We can achieve this with the help of operators or methods.

Set Union

A union of two sets A and B will result in a set of all items combined from both sets. There are two ways to perform union operation: 1) Using | operator 2) using union() method. See Listing 1-33.

Listing 1-33. Example code for set union operation

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
```

```
# use | operator
print "Union of A | B", A|B

# alternative we can use union()
A.union(B)
---- output ----
Union of A | B set([1, 2, 3, 4, 5, 6, 7, 8])
```

Set Intersection

An intersection of two sets A and B will result in a set of items that exists or is common in both sets. There are two ways to achieve intersection operation: 1) using & operator 2) using intersection() method. See Listing 1-34.

Listing 1-34. Example code for set intersesction operation

```
# use & operator
print "Intersection of A & B", A & B

# alternative we can use intersection()
print A.intersection(B)
---- output ----
Intersection of A & B set([4, 5])
```

Set Difference

The difference of two sets A and B (i.e., A - B) will result in a set of items that exists only in A and not in B. There are two ways to perform a difference operation: 1) using '–' operator, and 2) using difference() method. See Listing 1-35.

Listing 1-35. Example code for set difference operation

```
# use - operator on A
print "Difference of A - B", A - B

# alternative we can use difference()
print A.difference(B)
---- output ----
Difference of A - B set([1, 2, 3])
```

Set Symmetric Difference

A symmetric difference of two sets A and B is a set of items from both sets that are not common. There are two ways to perform a symmetric difference: 1) using ^ operator, and 2) using *symmetric_difference()method.* See Listing 1-36.

Listing 1-36. Example code for set symmetric difference operation

```
# use ^ operator
print "Symmetric difference of A ^ B", A ^ B

# alternative we can use symmetric_difference()
A.symmetric_difference(B)
---- output ----
Symmetric difference of A ^ B set([1, 2, 3, 6, 7, 8])
```

Basic Operations

Let's look at fundamental operations that can be performed on Python sets. See Listing 1-37.

Listing 1-37. Example code for basic operations on sets

```
# Return a shallow copy of a set
lang = languages.copy()
print languages
print lang

# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

print A.isdisjoint(B) # True, when two sets have a null intersection print
A.issubset(B) # True, when another set contains this set print
A.issuperset(B) # True, when this set contains another set sorted(B) #
Return a new sorted list
print sum(A) # Retrun the sum of all items
print len(A) # Return the length
print min(A) # Return the largestitem
print max(A) # Return the smallest item
---- output ----
set(['C', 'Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'AI', 'R',  'SAS', 'Machine
Learning'])
set(['C', 'Java', 'Python', 'Data Science', 'Julia', 'SPSS', 'AI', 'R',  'SAS', 'Machine
Learning'])
False
False
False
15
5
1
5
```


Dictionary

The Python dictionary will have a key and value pair for each item that is part of it. The key and value should be enclosed in curly braces. Each key and value is separated using a colon (:), and further each item is separated by commas (,). Note that the keys are unique within a specific dictionary and must be immutable data types such as strings, numbers, or tuples, whereas values can take duplicate data of any type. See Table 1-13; and Listings 1-38, 1-39, 1-40, 1-41, and 1-42.

Table 1-13. *Python Dictionary operations*

Description Python Expression Example Results

Creating a dictionary	dict = {‘key1’:‘value1’, ‘key2’:‘value2’…..}	dict = {‘Name’: ‘Jivin’, ‘Age’: 6, ‘Class’: ‘First’}	{‘Name’: ‘Jivin’, ‘Age’: 6, ‘Class’: ‘First’}
Accessing items in dictionary	dict [‘key’]	dict[‘Name’]	dict[‘Name’]: Jivin
Deleting a dictionary	del dict[‘key’]; dict.clear(); del dict;	dict.clear(); del dict;	‘Class’:‘First’}; {};
Updating a dictionary	del dict[‘Name’]; dict[‘key’] = new_value dict[‘Age’] = 6.5	{‘Age’:6, dict[‘Age’]: 6.5	
Length	len(dict)	len({‘Name’: ‘Jivin’, ‘Age’: 6, ‘Class’: ‘First’})	3
Comparing elements of dicts	cmp(dict_1, dict_2)	cmp (dict1, dict4)	Return Value : -1 Return Value : 1 Return Value : 0

dict1 = {‘Name’: ‘Jivin’, ‘Age’: 6};
dict2 = {‘Name’: ‘Pratham’, ‘Age’: 7};
dict3 = {‘Name’: ‘Pranuth’, ‘Age’: 7};
dict4 = {‘Name’: ‘Jivin’, ‘Age’: 6};
print “Return Value: ”, cmp (dict1, dict2)
print “Return Value: ” , cmp (dict2, dict3)
print “Return Value: ”,

(*continued*)

Table 1-13. (*continued*)

Description	Python Expression	Example	Results
String representation of dict		‘Jivin’, ‘Age’: 6}; print “Equivalent String: ”, str (dict)	“New Dictionary: ”, str(dict)
Return the shallow copy of dict	dict.copy()	dict = {‘Name’: ‘Jivin’, ‘Age’: 6}; dict1 = dict.copy() print dict1	dict = dict. fromkeys(seq, 10) print “New Dictionary: ”, str(dict) Equivalent String : {‘Age’: 6, ‘Name’: ‘Jivin’}
Create a new dictionary with keys from seq and values set to value	dict.fromkeys()	seq = (‘name’, ‘age’, ‘sex’)	
str(dict)	dict = dict. fromkeys(seq)	print	{‘Age’: 6, ‘Name’: ‘Jivin’}

None, ‘name’: None, ‘sex’: None} New Dictionary : {‘age’: 10, ‘name’: 10, ‘sex’: 10}

New Dictionary : {‘age’:

returns value or default if key not in dictionary dict.get(key, default=None) dict = {‘Name’: ‘Jivin’, ‘Age’: 6} print “Value for Age: ”, dict.get(‘Age’) print “Value for Education: ”, dict.get(‘Education’, “First Grade”) Value : 6 Value : First Grade

For key key,

‘Jivin’, ‘Age’: 6}

print “dict items: ”, dict.items()

dict.keys() dict = {‘Name’: ‘Jivin‘, ‘Age’: 6} Value : [(‘Age’, 6), (‘Name’, ‘Jivin’)]

Returns true if key in dictionary dict, false otherwise

Value : [‘Age’, ‘Name’]

Returns a list of dict’s (key, value) tuple pairs

(*continued*)

Returns list of dictionary dict’s keys dict.has_key(key) dict = {‘Name’: ‘Jivin’, ‘Age’: 6} print “dict keys: ”, dict.keys() Value : True Value : False print “Age exists? ”, dict.has_key(‘Age’) print “Sex exists? ”, dict.has_key(‘Sex’)

dict.items() dict = {‘Name’:

Table 1-13. (*continued*)

Description	Python Expression	Example	Results
Similar to get(), but will set dict[key]=default if key is not already in dict	dict.setdefault(key, default=None) dict = {‘Name’:	‘Jivin’, ‘Age’: 6} print “Value for Age: ”, dict.setdefault (‘Age’, None) print “Value for Sex: ”,	dict. setdefault(‘Sex’, None) Value : 6 Value : None
Adds dictionary dict2’s key-values pairs to dict	Returns list of dictionary dict’s values dict.update(dict2) dict =	{‘Name’: ‘Jivin’, ‘Age’: 6} dict2 = {‘Sex’: ‘male’ } dict.update(dict2) print “dict. update(dict2) = ”, dict	

dict.values() dict = {‘Name’: ‘Jivin’, ‘Age’: 6}
print “Value: ”, dict.values()

Value : {‘Age’: 6, ‘Name’: ‘Jivin’, ‘Sex’: ‘male’}

Value : [6, ‘Jivin’]

Listing 1-38. Example code for creating dictionary

```
# Creating dictionary

dict = {'Name': 'Jivin', 'Age': 6, 'Class': 'First'} print "Sample
dictionary: ", dict

---- output ----

Sample dictionary:  {'Age': 6, 'Name': 'Jivin', 'Class': 'First'}
```

Listing 1-39. Example code for accessing dictionary

```
# Accessing items in dictionary
print "Value of key Name, from sample dictionary:", dict['Name']

---- output ----
Value of key Name, from sample dictionary: Jivin
```


Listing 1-40. Example for deleting dictionary

```
# Deleting a dictionary
dict = {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
print "Sample dictionary: ", dict
del dict['Name'] # Delete specific item
print "Sample dictionary post deletion of item Name:", dict

dict = {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
dict.clear() # Clear all the contents of dictionary
print "dict post dict.clear():", dict

dict = {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
del dict # Delete the dictionary

---- output ----

Sample dictionary:  {'Age': 6, 'Name': 'Jivin', 'Class': 'First'} Sample dictionary post
```

deletion of item Name: {'Age': 6, 'Class': 'First'} dict post dict.clear(): {}

Listing 1-41. Example code for updating dictionary

```
# Updating dictionary

dict = {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
print "Sample dictionary: ", dict
dict['Age'] = 6.5

print "Dictionary post age value update: ", dict
---- output ----
Sample dictionary:  {'Age': 6, 'Name': 'Jivin', 'Class': 'First'} Dictionary post age
value update:  {'Age': 6.5, 'Name': 'Jivin', 'Class':  'First'}
```

Listing 1-42. Example code for basic operations on dictionary

```
# Basic operations

dict = {'Name': 'Jivin', 'Age': 6, 'Class': 'First'}
print "Length of dict: ", len(dict)

dict1 = {'Name': 'Jivin', 'Age': 6};
dict2 = {'Name': 'Pratham', 'Age': 7};
dict3 = {'Name': 'Pranuth', 'Age': 7};
dict4 = {'Name': 'Jivin', 'Age': 6};
print "Return Value: dict1 vs dict2", cmp (dict1, dict2) print
"Return Value: dict2 vs dict3", cmp (dict2, dict3) print "Return
Value: dict1 vs dict4", cmp (dict1, dict4)
```



```
# String representation of dictionary
dict = {'Name': 'Jivin', 'Age': 6}
print "Equivalent String: ", str (dict)

# Copy the dict
dict1 = dict.copy()
print dict1

# Create new dictionary with keys from tuple and values to set value seq =
('name', 'age', 'sex')

dict = dict.fromkeys(seq)
print "New Dictionary: ", str(dict)

dict = dict.fromkeys(seq, 10)
print "New Dictionary: ", str(dict)

# Retrieve value for a given key
dict = {'Name': 'Jivin', 'Age': 6};
print "Value for Age: ", dict.get('Age')
# Since the key Education does not exist, the second argument will be
returned
print "Value for Education: ", dict.get('Education', "First Grade")
```

```
# Check if key in dictionary
print "Age exists? ", dict.has_key('Age')
print "Sex exists? ", dict.has_key('Sex')

# Return items of dictionary
print "dict items: ", dict.items()

# Return items of keys
print "dict keys: ", dict.keys()

# return values of dict
print "Value of dict: ",  dict.values()

# if key does not exists, then the arguments will be added to dict and  returned
print "Value for Age : ", dict.setdefault('Age', None) print
"Value for Sex: ", dict.setdefault('Sex', None)

# Concatenate dicts
dict = {'Name': 'Jivin', 'Age': 6}
dict2 = {'Sex': 'male' }

dict.update(dict2)
print "dict.update(dict2) = ",  dict
```



```
---- output ----
Length of dict:  3
Return Value: dict1 vs dict2 -1
Return Value: dict2 vs dict3 1
Return Value: dict1 vs dict4 0
Equivalent String:  {'Age': 6, 'Name': 'Jivin'}
{'Age': 6, 'Name': 'Jivin'}
New Dictionary:  {'age': None, 'name': None, 'sex': None}
New Dictionary:  {'age': 10, 'name': 10, 'sex': 10}
Value for Age:  6
Value for Education:  First Grade
Age exists?  True
Sex exists?  False
dict items:  [('Age', 6), ('Name', 'Jivin')]
dict keys:  ['Age', 'Name']
Value for Age :  6
Value for Sex:  None
dict.update(dict2) =  {'Age': 6, 'Name': 'Jivin', 'Sex': 'male'} Value of
dict:  [6, 'Jivin', 'male']
```

User-Defined Functions

A user-defined function is a block of related code statements that are organized to achieve a single related action. The key objective of the user-defined functions concept is to encourage modularity and enable reusability of code.

Defining a Function

Functions need to be defined, and below is the set of rules to be followed to define a function in Python.

- The keyword def denotes the beginning of a function block, which will be followed by the name of the function and open, close parentheses. After this a colon (:) to be put to indicate the end of the function header.
- Functions can accept arguments or parameters. Any such input arguments or parameters should be placed within the parentheses in the header of the parameter.
- The main code statements are to be put below the function header and should be indented, which indicates that the code is part of the same function.
- Functions can return an expression to the caller. If return method is not used at the end of the function, it will act as a subprocedure.
  The key difference between the function and the subprocedure is that a function will always return expression whereas a subprocedure will not. See Listings 1-43 and I-44.

Syntax for creating functions without argument:

```
def functoin_name():
   1st block line
   2nd block line
   ...
```

Listing 1-43. Example code for creating functions without argument

```
# Simple function
def someFunction():
   print "Hello World"

# Call the function
someFunction()

----- output -----
Hello world
```

Syntax for Creating Functions with Argument

```
def functoin_name(parameters):
   1st block line
   2nd block line
   ...
   return [expression]
```

Listing 1-44. Example code for creating functions with arguments

```
# Simple function to add two numbers
def sum_two_numbers(x, y):
    return x + y

# after this line x will hold the value 3
print sum_two_numbers(1,2)
----- output -----
3
```

Scope of Variables

The availability of a variable or identifier within the program during and after the execution is determined by the scope of a variable. There are two fundamental variable scopes in Python.

1. Global variables
2. Local variables

Note that Python does support global variables without you having to explicitly express that they are global variables. See Listing 1-45.

Listing 1-45. Example code for defining variable scopes

```
# Global variable
x = 10

# Simple function to add two numbers
def sum_two_numbers(y):
    return x + y

# Call the function and print result
print sum_two_numbers(10)

----- output -----
20
```

Default Argument

You can define a default value for an argument of function, which means the function will assume or use the default value in case any value is not provided in the function call for that argument. See Listing 1-46.

Listing 1-46. Example code for function with default argument

```
# Simple function to add two number with b having default value of 10 def
sum_two_numbers(x, y = 10):
    return x + y
```

```
# Call the function and print result
print sum_two_numbers(10)
20

print sum_two_numbers(10, 5)
15
```

Variable Length Arguments

There are situations when you do not know the exact number of arguments while defining the function and would want the ability to process all the arguments dynamically. Python's answer for this situation is the variable length argument that enables us to process more arguments than you specified while defining the function. The *args and **kwargs is a common idiom to allow a dynamic number of arguments.

The *args Will Provide All Function Parameters in the Form of a tuple

See Listing 1-47.

Listing 1-47. Example code for passing argumens as *args

```
# Simple function to loop through arguments and print them def
sample_function(*args):
    for a in args:
        print a

# Call the function
Sample_function(1,2,3)
1
2
3
```

The **kwargs will give you the ability to handle named or keyword arguments keyword that you have not defined in advance. See Listing 1-48.

Listing 1-48. Example code for passing argumens as **kwargs

```
# Simple function to loop through arguments and print them def
sample_function(**kwargs):
    for a in kwargs:
        print a, kwargs[a]

# Call the function
sample_function(name='John', age=27)
age 27
name 'John'
```

Module

A module is a logically organized multiple independent but related set of codes or functions or classes. The key principle behind module creating is it's easier to understand, use, and has efficient maintainability. You can import a module and the Python interpreter will search for the module in interest in the following sequences.

1. Currently active directly, that is, the directory from which the Python your program is being called.
2. If the module isn't found in currently active directory, Python then searches each directory in the path variable PYTHONPATH. If this fails then it searches in the default package installation path.

Note that the module search path is stored in the system module called sys as the sys. path variable, and this contains the current directory, PYTHONPATH, and the installation dependent default.

When you import a module, it's loaded only once, regardless of the number of times it is imported. You can also import specific elements (functions, classes, etc.) from your module into the current namespace. See Listing 1-49.

Listing 1-49. Example code for importing modules

```
# Import all functions from a module
import module_name
from modname import*

# Import specific function from module
from module_name import function_name
```

Python has an internal dictionary known as namespace that stores each variable or identifier name as the key and their corresponding value is the respective Python object. There are two types of namespace, local and global. The local namespace gets created during execution of a Python program to hold all the objects that are being created by the program. The local and global variable have the same name and the local variable shadows the global variable. Each class and function has its own local namespace. Python assumes that any variable assigned a value in a function is local. For global variables you need to explicitly specify them.

Another key built-in function is the dir(), and running this will return a sorted list of strings containing the names of all the modules, variables, and functions that are defined in a module. See Listing 1-50.

Listing 1-50. Example code dir() operation

```
Import os

content = dir(os)
print content

---- output ----

['F_OK', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_NOINHERIT',
```

'O_RANDOM', 'O_RDONLY', 'O_RDWR', 'O_SEQUENTIAL', 'O_SHORT_LIVED', 'O_TEMPORARY', 'O_TEXT', 'O_TRUNC', 'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO', 'P_OVERLAY', 'P_WAIT', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_ SET', 'TMP_MAX', 'UserDict', 'W_OK', 'X_OK', '_Environ', '__all__', '__ builtins__', '__doc__', '__file__', '__name__', '__package__', '_copy_reg', '_execvpe', '_exists', '_exit', '_get_exports_list', '_make_stat_result', '_make_statvfs_result', '_pickle_stat_result', '_pickle_statvfs_result', 'abort', 'access', 'altsep', 'chdir', 'chmod', 'close', 'closerange', 'curdir', 'defpath', 'devnull', 'dup', 'dup2', 'environ', 'errno', 'error', 'execl', 'execle', 'execlp', 'execlpe', 'execv', 'execve', 'execvp',

'execvpe', 'extsep', 'fdopen', 'fstat', 'fsync', 'getcwd', 'getcwdu', 'getenv', 'getpid', 'isatty', 'kill', 'linesep', 'listdir', 'lseek', 'lstat', 'makedirs', 'mkdir', 'name', 'open', 'pardir', 'path', 'pathsep', 'pipe', 'popen', 'popen2', 'popen3', 'popen4', 'putenv', 'read', 'remove', 'removedirs', 'rename', 'renames', 'rmdir', 'sep', 'spawnl', 'spawnle', 'spawnv', 'spawnve', 'startfile', 'stat', 'stat_float_times', 'stat_ result', 'statvfs_result', 'strerror', 'sys', 'system', 'tempnam', 'times', 'tmpfile', 'tmpnam', 'umask', 'unlink', 'unsetenv', 'urandom', 'utime', 'waitpid', 'walk', 'write']

Looking at the above output, __name__ is a special string variable name that denotes the module's name and __file__is the filename from which the module was loaded.

File Input/Output

Python provides easy functions to read and write information to a file. To perform read or write operation on files we need to open them first. Once the required operation is complete, it needs to be closed so that all the resources tied to that file are freed. See Table 1-14. Below is the sequence of a file operation.

- Open a file
- Perform operations that are read or write
- Close the file

Table 1-14. *File input / output operations*

Description	Syntax	Example
Opening a file	obj=open(filename , access_ mode , buffer)	f = open('vehicles.txt', 'w')
Reading from a file	fileobject.read(value)	f = open('vehicles.txt') f.readlines()
Closing a file	fileobject.close()	f.close()
Writing to a file	fileobject.write(string str)	vehicles = ['scooter\n', 'bike\n', 'car\n'] f = open('vehicles.txt', 'w') f.writelines(vehicles) f.close()

Opening a File

While opening a file the access_mode will determine the file open mode that is read, write, append etc. Read (r) mode is the default file access mode and this is an optional parameter,

Please refer to Table 1-15 for a complete list of file opening modes. Also see Listing 1-51.

Table 1-15. *File opening modes*

Modes Description

R reading only

Rb reading only in binary format

r+ file will be available for both read and write

rb+ file will be available for both read and write in binary format W writing only

Wb writing only in binary format

w+ open for both writing and reading, if file existing overwrite else create

wb+ open for both writing and reading in binary format; if file existing, overwrite, else create

A Opens file in append mode. Creates a file if does not exist Ab opens file in append mode. Creates a file if it does not exist

a+ opens file for both append and reading. Creates a file if does not exist

ab+ Opens file for both append and reading in binary format. Creates a file if it does not exist

Listing 1-51. Example code for file operations

```
# Below code will create a file named vehicles and add the items. \n is a newline character
vehicles = ['scooter\n', 'bike\n', 'car\n']
f = open('vehicles.txt', 'w')
f.writelines(vehicles)

# Reading from file
f = open('vechicles.txt')
print f.readlines()
f.close()

---- output ----

['scooter\n', 'bike\n', 'car\n']
```

Exception Handling

Any error that happens while a Python program is being executed that will interrupt the expected flow of the program is called as exception. Your program should be designed to handle both expected and unexpected errors.

Python has rich set of built-in exceptions that forces your program to output an error when something in it goes wrong.

Below in Table 1-16 is the list of Python Standard Exceptions as described in Python's official documentation (https://docs.python.org/2/library/exceptions.html).

Table 1-16. *Python built-in exception handling*

Exception name	Description
Exception	Base class for all exceptions.
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement. AttributeError
	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence. KeyError Raised
	when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
IOError	Raised for operating system-related errors. (*continued*)

Table 1-16. (*continued*)

Exception name	Description
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

You can handle exceptions in your Python program using try, raise, except, and finally statements.

try and except: try clause can be used to place any critical operation that can raise an exception in your program and an exception clause should have the code that will handle a raised exception. See Listing 1-52.

Listing 1-52. Example code for exception handling

```
try:
x = 1
y = 1
   print "Result of x/y: ", x / y
except (ZeroDivisionError):
   print("Can not divide by zero")
except (TypeError):
   print("Wrong data type, division is allowed on numeric data type only") except:
   print "Unexpected error occurred", '\n', "Error Type: ", sys.exc_info() [0], '\n', "Error
Msg: ", sys.exc_info()[1]
```

```
---- output ----
Result of x/y:  1
```

■ **Note** 1) changing value of b to zero in the above code will print the statement “Can’t divide by zero.”

2) Replacing ‘a’ with ‘A’ in divide statement will print below output.

Unexpected error occurred

Error Type: <type ‘exceptions.NameError’>

Error Msg: name ‘A’ is not defined

Finally: This is an optional clause that is intended to define clean-up actions that must be executed under all circumstances. See Listing 1-53.

Listing 1-53. Example code for exception handling with file operations

```
# Below code will open a file and try to convert the content to integer try:
   f = open('vechicles.txt')
   print f.readline()
   i = int(s.strip())
except IOError as e:
   print "I/O error({0}): {1}".format(e.errno, e.strerror) except
ValueError:
   print "Could not convert data to an integer."
except:
    print "Unexpected error occurred", '\n', "Error Type: ", sys.exc_info() [0], '\n',
      "Error Msg: ", sys.exc_info()[1]
finally:
   f.close()
   print "file has been closed"

---- output ----
scooter
Could not convert data to an integer.
file has been closed
```

Python executes a finally clause always before leaving the try statement irrespective of an exception occurrence. If an exception clause not designed to handle the exception is raised in the try clause, the same is re-raised after the finally clause has been executed. If usage of statements such as break, continue, or return forces the program to exit the try clause, still the finally is executed on the way out. See Figure 1-3.

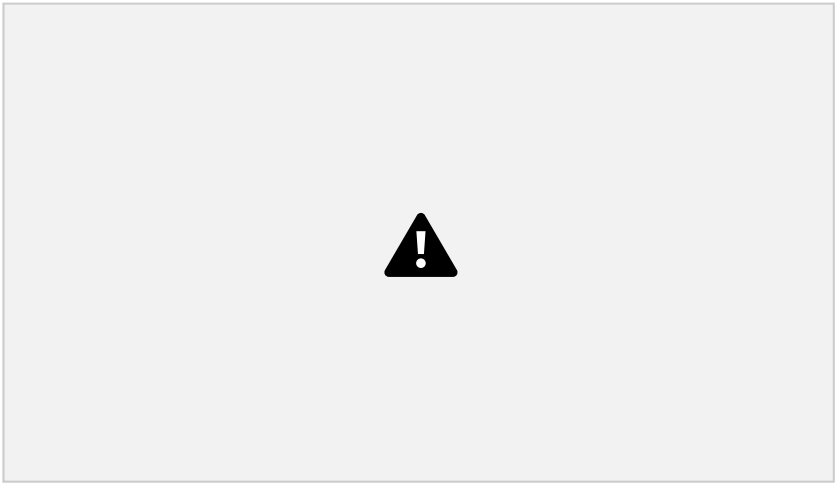


Figure 1-3. *Code Flow for Error Handler*

Note that generally it's a best practice to follow a single exit point principle by using finally. This means that either after successful execution of your main code or your error handler has finished handling an error, it should pass through the finally so that the code will be exited at the same point under all circumstances.

Endnotes

With this we have reached the end of this chapter. So far I have tried to cover the basics and the essential topics to get you started in Python, and there is an abundance of online / offline resources available to increase your knowledge depth about Python as a programming language. On the same note, I would like to leave you with some useful resources for your future reference. See Table 1-17.

Table 1-17. *Additional resources*

Resource	Description	Mode
http://docs.python-guide.org/en/latest/intro/learning/	This is the Python's official tutorial, and it covers all the basics and offers a detailed tour of the language and standard libraries.	Online
http://awesome-python.com/	A curated list of awesome Python frameworks, libraries, software, and resources.	
The Hacker's Guide to Python	This book is aimed at developers who already know Python but want to learn from more experienced Python developers.	Online Book

Step 2 – Introduction to Machine Learning

Machine learning is a subfield of computer science that evolved from the study of *pattern recognition* and computational learning theory in Artificial Intelligence (AI). Let's look at a few other versions of definitions that exist for machine learning:

- In 1959, Arthur Samuel, an American pioneer in the field of computer gaming, machine learning, and artificial intelligence has defined machine learning as a "Field of study that gives computers the ability to learn without being explicitly programmed."
- Machine learning is a field of computer science that involves using statistical methods to create programs that either improve performance over time, or detect patterns in massive amounts of data that humans would be unlikely to find.
- Machine Learning explores the study and construction of algorithms that can learn from and make predictions on data. Such algorithms operate by building a model from example inputs in order to make data driven predictions or decisions, rather than following strictly static program instructions.

All the above definitions are correct; in short, "Machine Learning is a collection of algorithms and techniques used to create computational systems that learn from data in order to make predictions and inferences."

Machine learning application area is abounding. Let's look at some of the most common day-to-day applications of Machine Learning that happens around us. *Recommendation System*: YouTube brings videos for each of its users based on a recommendation system that believes that the individual user will be interested in. Similarly Amazon and other such e-retailers suggest products that the customer will be interested in and likely to purchase by looking at the purchase history for a customer and a large inventory of products.

Spam detection: Email service providers use a machine learning model that can automatically detect and move the unsolicited messages to the spam folder.

M. Swamynathan, *Mastering Machine Learning with Python in Six Steps*,
DOI 10.1007/978-1-4842-2866-1_2

Prospect customer identification: Banks, insurance companies, and financial

organizations have machine learning models that trigger alerts so that organizations intervene at the right time to start engaging with the right offers for the customer and persuade them to convert early. These models observe the pattern of behavior by a user during the initial period and map it to the past behaviors of all users to identify those that will buy the product and those that will not.

History and Evolution

Machine learning is a subset of Artificial Intelligence (AI), so let's first understand what AI is and where machine learning fits within its wider umbrella. AI is a broad term that aims at using data to offer solutions to existing problems. It is the science and engineering of replicating, even surpassing human level intelligence in machines. That means observe or read, learn, sense, and experience.

The AI process loop is as follows in Figure 2-1:

Figure 2-1. *AI process loop*

- Observe – identify patterns using the data
- Plan – find all possible solutions
- Optimize – find optimal solution from the list of possible solutions
- Action – execute the optimal solution
- Learn and Adapt – is the result giving expected result, if no adapt

The AI process loop discussed above can be achieved using intelligent agents. A robotic intelligent agent can be defined as a component that can perceive its environment through different kinds of sensors (camera, infrared, etc.), and will take actions within the environment through efforts. Here robotic agents are designed to reflect humans. We have different sensory organs such as eyes, ears, noses, tongues, and skin to perceive our environment and organs such as hands, legs, and mouths are the effectors that enable us to take action within our environment based on the perception.

A detailed discussion on designing the agent has been discussed in the book *Artificial Intelligence, A Modern Approach* by Stuart J. Russell and Peter Norvig in 1995. Figure 2-2 shows a sample pictorial representation.

Figure 2-2. *Depict of robotic intelligent agent concept that interacts with its environment through sensors and effectors*

To get a better understanding of the concept, let's look at an intelligent agent designed for a particular environment or use case. Consider designing an automated taxi driver. See Table 2-1.

Table 2-1. *Example intelligent agent components*

Intelligent Agent's Component Name	Description
Agent Type	Taxi driver
Goals	Safe trip, legal, comfortable trip, maximize profits, convenient, fast
Environment	Roads, traffic, signals, signage, pedestrians, customers
Percepts	Speedometer, microphone, GPS, cameras, sonar, sensors
Actions	Steer, accelerate, break, talk to passenger

The taxi driver robotic intelligent agent will need to know its location, the direction in which its traveling, the speed at which its traveling, what else is on the road! This information can be obtained from the percepts such as controllable cameras in appropriate places, the speedometer, odometer, and accelerometer. For understanding the mechanical state of the vehicle and engine, it needs electrical system sensors. In addition a satellite global positioning system (GPS) can help to provide its accurate position information with respect to an electronic map and infrared/sonar sensors to detect distances to other cars or obstacles around it. The actions available to the intelligent taxi driver agent are the control over the engine through the pedals for accelerating, braking, and steering for controlling the direction. There should also be a way to interact with or talk to the passengers to understand the destination or goal.

In 1950, Alan Turing a well-known computer scientist proposed a test known as Turing test in his famous paper "Computing Machinery and Intelligence." The test was designed to provide a satisfactory operational definition of intelligence, which required that a human being should not be able to distinguish the machine from another human being by using the replies to questions put to both.

To be able to pass the Turing test, the computer should possess the following capabilities:

- Natural language processing, to be able to communicate successfully in a chosen language
- Knowledge representation, to store information provided before or during the interrogation that can help in finding information, making decisions, and planning. This is also known as 'Expert System'
- Automated reasoning (speech), to use the stored knowledge map information to answer questions and to draw new conclusions where required
- Machine learning, to analyzing data to detect and extrapolate patters that will help adapt to new circumstances
- Computer vision to perceive objects or the analyzing of images to find features of the images
- Robotics devices that can manipulate and interact with its environment. That means to move the objects around based on the circumstance
- Planning, scheduling, and optimization, which means figuring ways to make decision plans or achieve specified goals, as well as analyzing the performance of the plans and designs

The above-mentioned seven capability areas of AI have seen a great deal of research and growth over the years. Although many of the terms in these areas are used interchangeably, we can see from the description that their objectives are different. Particularly machine learning has seen a scope to cut across all the seven areas of AI. See Figure 2-3.

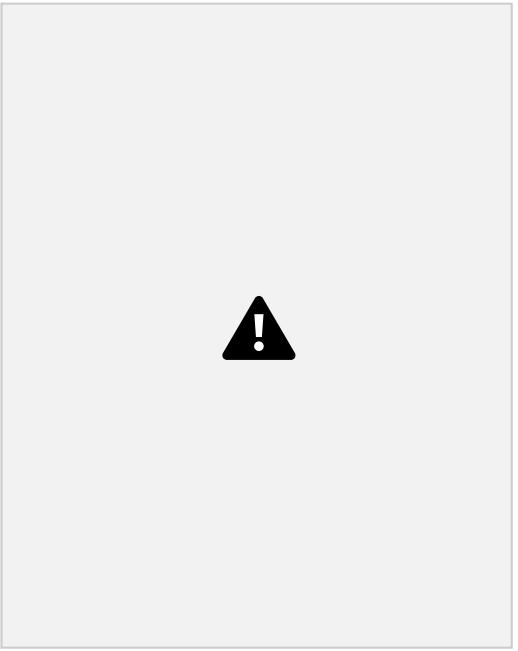


Figure 2-3. *Artificial Intelligence Areas*

Artificial Intelligence Evolution

Let's understand briefly the artificial intelligence past, present, and future.

Artificial Narrow Intelligence (ANI): Machine intelligence that equals or exceeds human intelligence or efficiency at a specific task. An example is IBM's Watson, which requires close participation of subject matter or domain experts to supply data/information and evaluate its performance.

Artificial General Intelligence (AGI): A machine with the ability to apply intelligence to any problem for an area, rather than just one specific problem. Self-driving cars are a good example of this.

Artificial Super Intelligence (ASI): An intellect that is much smarter than the best human brains in practically every field, general wisdom, social skills, and including scientific creativity. The key theme over here is "don't model the world, model the mind".

Different Forms

Is Machine Learning the only subject in which we use data to learn and use for prediction/inference?

To answer the above questions, let's have a look at the definition (Wikipedia) of the few other key terms (not an exhaustive list) that are often heard relatively:

- *Statistics*: It is the study of the collection, analysis, interpretation, presentation, and organization of data.
- *Data Mining*: It is an interdisciplinary subfield of computer science. It is the computational process of discovering patterns in large data sets (from data warehouse) involving methods at the intersection of artificial intelligence, machine learning, statistics, and database systems.
- *Data Analytics*: It is a process of inspecting, cleaning, transforming, and modeling data with the goal of discovering useful information, suggesting conclusions, and supporting decision making. This is also known as Business Analytics and is widely used in many industries to allow companies/organization to use the science of examining raw data with the purpose of drawing conclusions about that information and make better business decisions.
- *Data Science*: Data science is an interdisciplinary field about processes and systems to extract knowledge or insights from data in various forms, either structured or unstructured, which is a continuation of some of the data analysis fields such as statistics, machine learning, data mining, and predictive analytics, similar to Knowledge Discovery in Databases (KDD).

Yes, from the above definitions it is clear and surprising to find out that Machine Learning isn't the only subject in which we use data to learn from it and use further for prediction/inference. Almost identical themes, tools, and techniques are being talked about in each of these areas. This raises a genuine question about why there are so many different names with lots of overlap around learning from data? What is the difference between these?

The short answer is that all of these are practically the same. However, there exists a subtle difference or shade of meaning, expression, or sound between each of these. To get a better understanding we'll have to go back to the history of each of these areas and closely examine the origin, core area of application, and evolution of these terms.

Statistics

German scholar Gottfried Achenwall introduced the word "Statistics" in the middle of the 18th century (1749). Usage of this word during this period meant that it was related to the administrative functioning of a state, supplying the numbers that reflect the periodic actuality regarding its various area of administration. The origin of the word statistics may be traced to the Latin word "Status" ("council of state") or the Italian word "Statista" ("statesman" or "politician"); that is, the meaning of these words is "Political State" or a

Government. Shakespeare used a word Statist in his drama *Hamlet* (1602). In the

past, the statistics were used by rulers that designated the analysis of data about the state, signifying the “science of state.”

In the beginning of the 19th century, statistics attained the meaning of the collection and classification of data. The Scottish politician, Sir John Sinclair, introduced it to the English in 1791 in his book *Statistical Account of Scotland.* Therefore, the fundamental purpose of the birth of statistics was around data to be used by government and centralized administrative organizations to collect census data about the population for states and localities.

Frequentist

John Graunt was one of the first demographers and is our first vital statistician. He published his observation on the Bills of Mortality (in 1662), and this work is often quoted as the first instance of descriptive statistics. He presented vast amounts of data in a few tables that can be easily comprehended, and this technique is now widely known as descriptive statistics. In it we note that weekly mortality statistics first appeared in England in 1603 at the Parish-Clerks Hall. We can learn from it that in 1623, of some 50,000 burials in London, only 28 died of the plague. By 1632, this disease had practically disappeared for the time being, to reappear in 1636 and again in the terrible epidemic of 1665. This exemplifies that the fundamental nature of descriptive statistics is counting. From all the Parish registers, he counted the number of persons who died, and who died of the plague. The counted numbers every so often were relatively too large to follow, so he also simplified them by using proportion rather than the actual number. For example, the year 1625 had 51,758 deaths and of which 35,417 were of the plague. To simplify this he wrote, “We find the plague to bear unto the whole in proportion as 35 to 51. Or 7 to 10.” With these he is introducing the concept that the relative proportions are often of more interest than the raw numbers. We would generally express the above proportion as 70%. This type of conjecture that is based on a sample data’s proportion spread or frequency is known as “frequentist statistics.” . Statistical hypothesis testing is based on inference framework, where you assume that observed phenomena are caused by unknown but fixed processes.

Bayesian

In contract Bayesian statistics (named after Thomas Bayes), it describes that the probability of an event, based on conditions that might be related to the event. At the core of Bayesian statistics is Bayes’s theorem, which describes the outcome probabilities of related (dependent) events using the concept of conditional probability. For example, if a particular illness is related to age and life style, then applying a Bay’s theorem by considering a person’s age and life style more accurately increases the probability of that individual having the illness can be assessed.

Bayes theorem is stated mathematically as the following equation:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

Where A and B are events and P (B) ≠ 0

- P (A) and P (B) are the probabilities of observing A and B without regard to each other.
- P (A | B), a conditional probability, is the probability of observing event A given that B is true.
- P (B | A) is the probability of observing event B given that A is true.

For example, a doctor knows that lack of sleep causes migraine 50% of the time. Prior probability of any patient having lack of sleep is 10000/50000 and prior probability of any patient having migraine is 300/1000. If a patient has a sleep disorder, let's apply Bayes's theorem to calculate the probability he/she is having a migraine.

P (Sleep disorder | Migraine) = P(Migraine | Sleep disorder) * P(Migraine) / P(Sleep disorder)

P (Sleep disorder | Migraine) = .5 * 10000/50000 / (300/1000) = 33%

In the above scenario, there is a 33% chance that a patient with a sleep disorder will also have a migraine problem.

Regression

Another major milestone for statisticians was the regression, which was published by Legendre in 1805 and by Guass in 1809. Legendre and Gauss both applied the method to the problem of determining, from astronomical observations, the orbits of bodies about the Sun, mostly comets, but also later the then newly discovered minor planets. Gauss published a further development of the theory of least squares in 1821. Regression analysis is an essential statistical process for estimating the relationships between factors. It includes many techniques for analyzing and modeling various factors, and the main focus here is about the relationship between a dependent factor and one or many independent factors also called predictors or variables or features. We'll learn about this more in the fundamentals of machine learning with scikit-learn.

Over time the idea behind the word statistics has undergone an extraordinary transformation. The character of data or information provided has been extended to all spheres of human activity. Let's understand the difference between two terms quite often used along with statistics, that is, 1) data and 2) method. Statistical data is the numerical statement of facts, whereas statistical methods deal with information of the principles and techniques used in collecting and analyzing such data. Today, statistics as a separate discipline from mathematics is closely associated with almost all branches of education and human endeavor that are mostly numerically representable. In modem times, it has innumerable and varied applications both qualitatively and quantitatively. Individuals and organizations use statistics to understand data and make informed decisions throughout the natural and social sciences, medicine, business, and other areas. Statistics has served as the backbone and given rise to many other disciplines, which you'll understand as you read further.

Data Mining

The term “Knowledge Discovery in Databases” (KDD) is coined by Gregory Piatetsky Shapiro in 1989 and also at the same time he cofounded the first workshop named KDD. The term “Data mining” was introduced in the 1990s in the database community, but data mining is the evolution of a field with a slightly long history.

Data mining techniques are the result of research on the business process and product development. This evolution began when business data was first stored on computers in the relational databases and continued with improvements in data access, and further produced new technologies that allow users to navigate through their data in real time. In the business community, data mining focuses on providing “Right Data” at the “Right Time” for the “Right Decisions.” This is achieved by enabling a tremendous amount of data collection and applying algorithms to them with the help of distributed multiprocessor computers to provide real-time insights from data.

We’ll learn more about the five stages proposed by KDD for data mining in the section on the framework for building machine learning systems.

Data Analytics

Analytics have been known to be used in business, since the time of management movements toward industrial efficiency that were initiated in late 19th century by Frederick Winslow Taylor, an American mechanical engineer. The manufacturing industry adopted measuring the pacing of the manufacturing and assembly line, as
a result revolutionizing industrial efficiency. But analytics began to command more awareness in the late 1960s when computers had started playing a dominating role as organizations’ decision support systems. Traditionally business managers were making decisions based on past experiences or rules of thumb, or there were other qualitative aspects to decision making; however, this changed with development of data warehouses and enterprise resource planning (ERP) systems. The business managers and leaders considered data and relied on ad hoc analysis to affirm their experience/knowledge based assumptions for daily and critical business decisions. This evolved as data-driven business intelligence or business analytics for the decision-making process was fast adopted by organizations and companies across the globe. Today, businesses of all sizes use analytics. Often the word “Business Analytics” is used interchangeably for “Data Analytics” in the corporate world.

In order for businesses to have a holistic view of the market and how a company competes efficiently within that market to increase the RoI (Return on Investment), requires a robust analytic environment around the kind of analytics that is possible. This can be broadly categorized into four types. See Figure 2-4.

1. Descriptive Analytics
2. Diagnostic Analytics

3. Predictive Analytics
4. Prescriptive Analytics

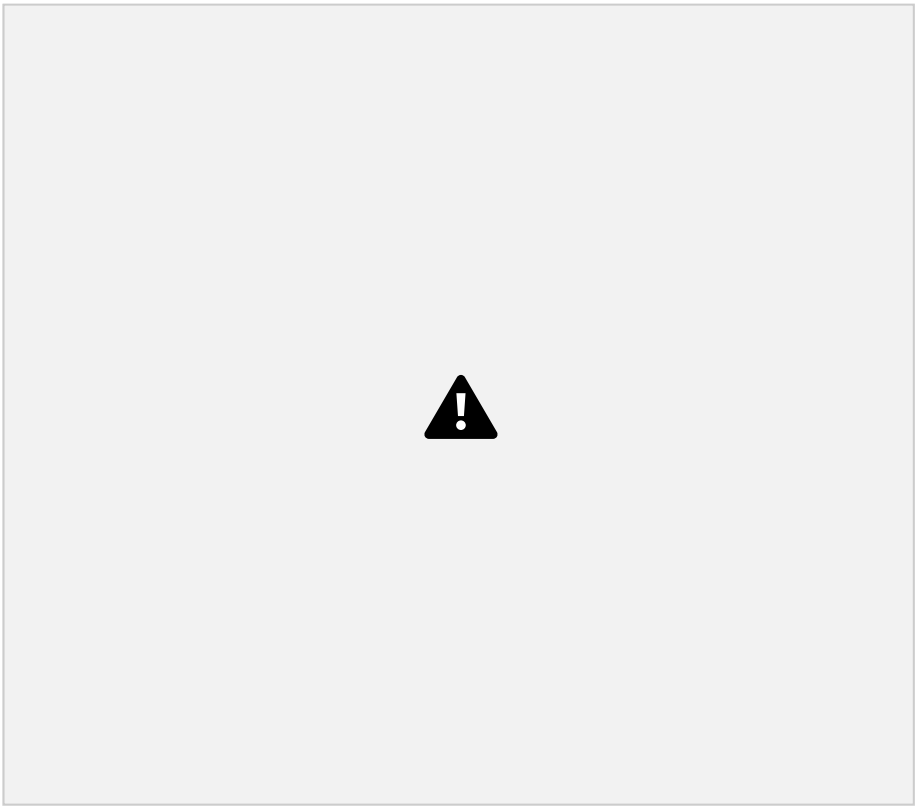


Figure 2-4. *Data Analytics Types*

Descriptive Analytics

They are the analytics that describe the past to tell us “What has happened?” To elaborate, as the name suggests, any activity or method that helps us to describe or summarize raw data into something interpretable by humans can be termed ‘Descriptive Analytics’. These are useful because they allow us to learn from past behaviors, and understand how they might influence future outcomes.

The statistics such as arithmetic operation of count, min, max, sum, average, percentage, and percent change, etc., fall into this category. Common examples of descriptive analytics are a company’s business intelligence reports that cover different aspects of the organization to provide historical hindsights regarding the company’s production, operations, sales, revenue, financials, inventory, customers, and market share.