

# INTRODUCTION TO PYTHON

Kamakshaiah Musunuru



# Contents

<b>1</b>	<b>Introduction</b>	<b>15</b>
1.1	Language Comparison . . . . .	16
1.2	About Python . . . . .	21
1.3	Features . . . . .	23
1.4	Installing Python . . . . .	24
1.4.1	Installing in Linux . . . . .	25
1.4.2	Installing in Windows . . . . .	26
1.5	Editors . . . . .	27
1.5.1	IDLE . . . . .	31
1.5.2	IPython . . . . .	36
1.5.3	Spyder . . . . .	38
1.5.4	Sublime Text Editor . . . . .	41
1.5.5	Atom . . . . .	42
1.5.6	Jupyter . . . . .	43
1.6	PyPI . . . . .	44
1.6.1	Playing safe with packages . . . . .	47
1.7	A bit about <code>sys</code> module . . . . .	54
1.8	Understanding OS through Python . . . . .	55
<b>2</b>	<b>Programming in Python</b>	<b>61</b>
2.1	Understanding Data types . . . . .	64
2.1.1	Types of Operator . . . . .	66
2.1.2	Tuples, Lists, Sets and Dictionaries . . . . .	70
2.2	Control Flow in Python . . . . .	82
2.2.1	<code>if</code> statement . . . . .	82

2.2.2	Loops . . . . .	84
2.3	Functional Programming in Python . . . . .	93
2.3.1	Modules . . . . .	94
2.3.2	Functions . . . . .	95
2.3.3	<code>lambda, map, filter, and reduce</code> . . . . .	97
2.3.4	Arguments in Functions . . . . .	100
2.4	Using loops for matrix operations . . . . .	107
2.4.1	NumPy for matrices . . . . .	107
2.4.2	List Comprehensions . . . . .	109
2.4.3	Matrix Manipulation . . . . .	120
2.4.4	Python packages for Matrices . . . . .	128
2.5	Creating Packages . . . . .	134
2.5.1	What is <code>__pycache__</code> ? . . . . .	141
2.5.2	What is <code>__init__.py</code> ? . . . . .	142
2.6	Exception handling . . . . .	147
2.6.1	Catching Exceptions in Python . . . . .	147
2.6.2	Errors and Exceptions . . . . .	148
2.6.3	Handling Exceptions . . . . .	150
2.6.4	Raising Exceptions . . . . .	153
2.6.5	Exception Chaining . . . . .	154
2.6.6	User-defined Exceptions . . . . .	155
2.6.7	Defining Clean-up Actions . . . . .	156
2.6.8	Predefined Clean-up Actions . . . . .	158
2.7	Regexpressions . . . . .	159
2.7.1	Matching Characters . . . . .	160
2.7.2	Using Regular Expressions . . . . .	162
2.7.3	Module-Level Functions . . . . .	167
2.7.4	Compilation Flags . . . . .	167
2.7.5	Grouping . . . . .	170
2.7.6	Non-capturing and Named Groups . . . . .	172
2.7.7	Modifying Strings . . . . .	173
2.7.8	Splitting Strings . . . . .	174
2.7.9	Search and Replace . . . . .	175
3	<b>Objective Oriented Programming</b>	<b>181</b>
3.1	Objects and classes . . . . .	182
3.1.1	Creating Class and Object in Python . . . . .	185
3.2	Inheritance . . . . .	187

3.2.1	Use of Inheritance in Python . . . . .	187
3.3	Python Operator Overloading . . . . .	190
3.3.1	Python Special Functions . . . . .	190
3.3.2	Overloading Operators . . . . .	192
3.3.3	Overloading Comparison Operators . . . . .	193
3.4	Encapsulation . . . . .	195
3.4.1	Data Encapsulation in Python . . . . .	195
3.5	Polymorphism . . . . .	196
3.5.1	Polymorphism with class methods with interface	196
3.5.2	Polymorphism with Inheritance . . . . .	197
<b>4</b>	<b>Numpy</b>	<b>199</b>
4.1	Installing NumPy . . . . .	200
4.1.1	How to import NumPy . . . . .	200
4.1.2	Reading the example code . . . . .	201
4.2	NumPy Arrays . . . . .	201
4.2.1	Array creation functions . . . . .	202
4.2.2	What is an array? . . . . .	202
4.2.3	Array dimensionality . . . . .	203
4.2.4	How to create a basic array . . . . .	204
4.2.5	Specifying your data type . . . . .	205
4.2.6	Multidimensional arrays . . . . .	207
4.2.7	Adding, removing, and sorting elements . . . . .	208
4.2.8	shape and size of an array . . . . .	210
4.2.9	Reshaping arrays . . . . .	210
4.2.10	Indexing and slicing . . . . .	211
4.3	Vectorized operations . . . . .	214
4.3.1	Along axis . . . . .	214
4.3.2	Over axis . . . . .	215
4.3.3	Vectorization . . . . .	215
4.4	Statistical analysis . . . . .	216
4.4.1	Quantile . . . . .	216
4.4.2	Percentile . . . . .	217
4.4.3	Averages and variances . . . . .	217
4.4.4	Correlating . . . . .	219
4.4.5	Covariance . . . . .	221
4.4.6	Curve fitting . . . . .	221

<b>5 SciPy</b>	<b>227</b>
5.1 SciPy Organization . . . . .	228
5.2 Random variables . . . . .	228
5.2.1 Common methods . . . . .	229
5.2.2 Random number generation . . . . .	231
5.2.3 Shape parameters . . . . .	233
5.2.4 Freezing a distribution . . . . .	234
5.2.5 Broadcasting . . . . .	234
5.2.6 Specific points for discrete distributions . . . . .	235
5.3 Building specific distributions . . . . .	236
5.3.1 Making a continuous distribution, i.e., subclassing <i>rv_continuous</i> . . . . .	236
5.3.2 Subclassing <i>rv_discrete</i> . . . . .	237
5.4 Analysing one sample . . . . .	241
5.4.1 Descriptive statistics . . . . .	241
5.4.2 T-test and KS-test . . . . .	242
5.4.3 Tails of the distribution . . . . .	243
5.4.4 Special tests for normal distributions . . . . .	245
5.5 Comparing two samples . . . . .	246
5.5.1 Comparing means . . . . .	247
5.5.2 Kolmogorov-Smirnov test for two samples . . . . .	247
<b>6 Data analysis</b>	<b>249</b>
6.1 Univariate analysis . . . . .	250
6.1.1 Measures of central tendency . . . . .	250
6.1.2 Measures of variability . . . . .	252
6.1.3 Measures of shape . . . . .	254
6.1.4 Student's t-test . . . . .	258
6.2 Bivariate analysis . . . . .	260
6.2.1 Two-sample t-test . . . . .	260
6.2.2 Chi-squared test . . . . .	265
6.2.3 Cross tabulation . . . . .	267
6.2.4 Measures of association . . . . .	270
6.2.5 Exact tests . . . . .	274
6.2.6 Bivariate correlation . . . . .	276
6.2.7 Regression . . . . .	279
6.2.8 Curve fitting . . . . .	282
6.2.9 Simple linear regression . . . . .	284

6.3	Multivariate analysis . . . . .	285
6.3.1	Multivariate analysis . . . . .	285
6.3.2	Types of analysis . . . . .	286
6.3.3	Covariance . . . . .	288
6.3.4	Correlation . . . . .	290
6.3.5	Multiple regression . . . . .	292
<b>7</b>	<b>pandas</b>	<b>301</b>
7.1	Data import and export . . . . .	302
7.1.1	Setting a column as the index . . . . .	303
7.1.2	Selecting specific columns to read into memory .	303
7.1.3	Reading Data from a URL . . . . .	303
7.2	Summary statsitics . . . . .	304
7.2.1	Exporting the DataFrame to a CSV File . . . . .	305
7.3	Inferential statistics . . . . .	307
7.3.1	Models and assumptions . . . . .	308
7.3.2	T Test . . . . .	309
7.3.3	Paired Samples t-Test in Pandas . . . . .	311
7.3.4	One-Way ANOVA . . . . .	312
7.3.5	Chi-Square Test . . . . .	313
7.3.6	Correlations . . . . .	315
7.3.7	Regression . . . . .	318
7.3.8	Multiple linear regression . . . . .	321
7.4	Exploratory statistics . . . . .	327
7.4.1	Techniques and tools . . . . .	328
7.4.2	Principal component analysis (PCA) . . . . .	328
<b>8</b>	<b>Visualization</b>	<b>335</b>
8.1	A simple example . . . . .	336
8.2	Parts of a Figure . . . . .	337
8.2.1	Figure . . . . .	337
8.2.2	Axes . . . . .	338
8.2.3	Axis . . . . .	338
8.2.4	Artist . . . . .	338
8.3	Coding styles . . . . .	339
8.4	Colors . . . . .	340
8.5	Linewidths, linestyles, and markersizes . . . . .	340
8.6	Labelling plots . . . . .	341

8.6.1	Axes labels and text . . . . .	341
8.6.2	Annotations . . . . .	343
8.6.3	Legends . . . . .	343
8.7	Plotting data . . . . .	344
8.7.1	Nominal . . . . .	345
8.7.2	Ordinal scale . . . . .	347
8.7.3	Interval scale . . . . .	349
8.7.4	Ratio scale . . . . .	350

# About the Author



Dr. M. Kamakshaiah is a open source software evangelist, full stack software developer and academic of data science & analytics. His teaching interests are IT practices in business, Business analytics, AI & ML, Cloud computing & development, IoT (Arduino, Raspberry Pi, ESP8266), and Natural Language Processing (NLP). He also teaches theoretical courses related to multivariate data analysis, numerical simulations & optimization, total quality management & sixsigma. Visit <https://www.youtube.com/@kamakshaiah> for his video demonstrations.

He has developed few free and open source software solutions meant for corporate practitioners, academics, scholars and students engaging in data science and analytics. All his software applications are available from his Github portal <https://github.com/Kamakshaiah>.



# Foreword

I started writing this book for one simple reason that I got to teach Python for business analytics students. Actually I started preserving every little code, chunk by chunk, as and when I struck Python while I was on web. Though I thought it as notes but the stuff that I gathered became rather more significant to compile it as a book. I am not a computerist but a simple academic of business studies.

I turned into data analyst due to my being accidentally exposed to open source software when I was in Ethiopia. I got to use GNU/Linux due to a very simple reason that as I got to find out a way to keep my lappy away from virus infection in the university. I crashed Windows a couple of times when I was working in office at the University. I found GNU/Linux as a wonderful solution to address virus issues. My first experience of GNU/Linux, perhaps, is *Jaunty Jackalope*, as I suppose. This must be Ubuntu 9.04, an LTS version, if I am not wrong. However, my involvement in GNU/Linux became a serious affair though *Karmic Koala*.<sup>1</sup> I still remember few of my colleagues used to visit my home for OS installation in those days during my stint at Ethiopia.

My first encounter with analytics software is R. R is *lingua franca* of statistics. I taught “business statistics” several times as academic but without using any software tool at first. Just before R, I used Excel for teaching statistical analysis. In fact, I had used SPSS very scantily

---

<sup>1</sup>*Karmic Koala* is Ubuntu 9.10. Visit <https://wiki.ubuntu.com/Releases> for more information on Ubuntu releases.

and much later after R for regular classroom teaching. Once, I notice this little yet uppercase R over there while I was browsing for statistical software in Ubuntu *package manager*. R changed not only my understanding of Statistics but the very way of learning the same. I addicted to R so much so that for every pretty little calculations I used to refer R manuals. It is not extraneous even if I state that I must be the first academic to introduce R in south Indian business management curriculum, yet remained unknown to others. I used to denounce and put it in writing while replying for every pretty academic conference/-workshop/seminars, whenever I came across practice of commercial software for data analytics. At first I was never understood here in academic fraternity until certain time. I started seeing the response and also the change in neighboring academic institutes slowly. Today, R is not only a best tool for learning data analysis but also a standard for the same.

I came across Python just as in same way as I discovered R in Ubuntu. Python was one of the default programming languages for many things in Ubuntu. At first I did not know the power of Python for everything I learned it was only by self study. All my learning is from online resources. However, I could produce a couple of thousands of data analysts through my formal classes by now. All that I learned and taught is only by my self study, I mean, through very informal personal learning. I think Python is suitable for both beginners and versatile developers. Its a programmer friendly language. Python has close to 4 million packages through its <https://pypi.org/>. Python is best programming language for budding data analysts and software developers. Its syntax is highly intuitive, and massive add-on libraries makes it a best choice for beginners. Python is gaining momentum in full stack development. Python's Django framework is being used by few best global companies such as Instagram, National Geographic, Mozilla, Spotify, and many more.<sup>2</sup> Python's Flask micro-services framework is other popular software for web development. Python is also accepted as one of the best programming language in IoT communities. MicroPython is offspring of Python which is widely used for programming microcontrollers.

---

<sup>2</sup>Retrieved from <https://www.trio.dev/blog/django-applications>.

I am writing all this not to show myself as a valiant learner, but to demonstrate how can a novice and a naive enthusiast like me can learn programming tools like R and Python. Today, I am offering a couple of courses to teach Hadoop and IoT, that is all by passion, not by formal education in computer science. So, I would like to give confidence to the reader that you don't need any formal learning in computer science or IT to learn data science and adopt it as profession. However, you need tons and tons of patience and passion.

This book is meant for beginners in data science and analytics. It has 5 chapters each section represents a unique concept of data analytics. This book may be useful for both practitioners and academics to acquire knowledge of Python. The first chapter *introduction to Python*, deals with very short information. Chapter I has information related to installation of Python in Linux, Windows and few other OSes. Chapter 2 deals with *Programming in Python* and this chapter has information related to data types, data structures, control flow *etc*. Chapter 3, *Data Simulations* deals with *numerical simulations* which shows as how to manage different type of data such as numeric, non-numeric and *etc*. Chapter 4, *Statistical Analysis* deals with different types of techniques related to statistics. This chapter is organized in three different sections such as *univariate*, *bivariate* and *multivariate* analyses. Chapter 5, *Python for Visualization*, rather offers various ways to plot different types of data. Python has very nice methods to make visuals and plot data appropriately as by the types.

As far as coding is concerned; those code sections where there exist left-bar such as the below

```
statement 1
statement 2
statement 3
```

represents a *script*. A script is a plain code file in which there exists program statements. There are other code sections where each statement is preceded by python prompt (*>>>*). These code sections are meant for testing. These sections are useful either to evaluate a Python statement or provide evidence for logic.

All the code chunks used in this book are provided though my github

portal with a project name *PfDSaA*. Feel free to visit the site <https://github.com/Kamakshaiah/PfDSaA> and download required .py files for practice.

One last note is that this book is meant for both data scientists and analysts. The learner need not have any preliminary knowledge of Python. However, little logical thinking together with passion and patience are required. The book is written in such a way that even a person having no knowledge on Python can pick up and become maverick at the end of reading. This book covers from very basic details ranging from installation to creating packages. I am not covering very advanced concepts such as software and applications development due to one reason to keep this book reasonable for both beginners and advanced users.

This book is also useful for practitioners of statistics. There is some certain yet worthy material meant for statisticians. All that information provided in data analysis is more suitable to students, scholars, academics and corporate practitioners of data science and analytics.

Happy reading ...

*Author*  
*Dr. M. Kamakshaiah*

# Chapter 1

## Introduction

“First, solve the problem. Then, write the code.”

- John Johnson

I don't know if at all there exists any way or personal style for writing programs or programming. As far as my style is concerned I always keep *calc* opened in my computer for all pretty numerical calculations. At least if you are a data analyst, I advise better keep a copy of *calc* in your computer.<sup>1</sup> From few years, I have been using *Libre Office Calc* that makes things easy. I do lots of calculations to check models first in *calc*. That makes things easy while writing programs in R or Python later. At least it saves a cup of tea or coffee at the end of the day lest I may end up with a gout attack.

As someone mentioned “*the programming is applying mental tricks to solve the problems*”. What a beautiful statement? Doesn't it sound nice? It is all about putting logic into a computer language so that it returns the expected or intended outcomes. The approach rather seems to be very simple; programming is all about factoring a problem and trying to see as how each and every part of it behaves so that after understanding the very problem might become susceptible for solving. While doing so the programmer tries to break down the problem into certain small parts so that each part behaves as the represent

of the whole. These parts are brought under the microscope to verify or establish logical relationships with other such parts of the problem. There are certain things that any user need to know about a programming language while before learning in-depth issues of the same. They are *data types/structures, conditional statements and loops*. This apart they are language specific tricks such as the one called *lambda* in Python.

## 1.1 Language Comparison

There are quite a few programming languages in the world. Each one of which is specific in its own entirety. The first high-level programming language was *Plankalkül*, created by Konrad Zuse between 1942 and 1945. The first high-level language to have an associated compiler was created by Corrado Böhm in 1951, for his PhD thesis. The first commercially available language was FORTRAN (FORmula TRANslation); developed in 1956 (first manual appeared in 1956, but first developed in 1954) by a team led by John Backus at **International Business Machines (IBM)**.

Another early programming language was devised by Grace Hopper in the US, called FLOW-MATIC. It was developed for the **UNIVersal Automatic Computer (UNIVAC)** I at Remington Rand during the period from 1955 until 1959. Hopper found that business data processing customers were uncomfortable with mathematical notation, and in early 1955, she and her team wrote a specification for an English programming language and implemented a prototype. The FLOW-MATIC compiler became publicly available in early 1958 and was substantially complete in 1959. Flow-Matic was a major influence in the design of **Common Business-Oriented Language (COBOL)**, since only it and its direct descendant **AIR MAterial COmpiler (AIMACO)** were in actual use at the time. Other languages still in use today include **LISP** (1958), invented by John McCarthy and **COBOL** (1959), created by the Short Range Committee.

### List of Programming Languages by their Year

Some notable languages that were developed in this period include:

Year	Language
1951	Regional Assembly Language
1952	Autocode
1954	IPL (forerunner to LISP)
1955	FLOW-MATIC (led to COBOL)
1957	FORTRAN (first compiler)
1957	COMTRAN (precursor to COBOL)
1958	LISP
1958	ALGOL 58
1959	FACT (forerunner to COBOL)
1959	COBOL
1959	RPG
1962	APL
1962	Simula
1962	SNOBOL
1963	CPL (forerunner to C)
1964	Speakeasy
1964	BASIC
1964	PL/I
1966	JOSS
1966	MUMPS
1967	BCPL (forerunner to C)

Table 1.1: List of Programming Languages by their Year

he period from the late 1960s to the late 1970s brought a major flowering of programming languages. Most of the major language paradigms now in use were invented in this period. *Speakeasy*, developed in 1964 at Argonne National Laboratory (ANL) by Stanley Cohen, is an OOPS (object-oriented programming system, much like the later MATLAB, IDL and Mathematica) numerical package. Speakeeasy has a clear Fortran foundation syntax. C, an early systems programming language, was developed by Dennis Ritchie and Ken Thompson at Bell Labs between 1969 and 1973.

The 1960s and 1970s also saw considerable debate over the merits of “structured programming”, which essentially meant programming without the use of “goto”. A significant fraction of programmers be-

lieved that, even in languages that provide “goto”, it is bad programming style to use it except in rare circumstances. This debate was closely related to language design: some languages did not include a “goto” at all, which forced structured programming on the programmer.

Some notable languages that were developed in this period include:

Year	Language
1967	BCPL (forerunner to B)
1968	Logo
1969	B (forerunner to C)
1970	Pascal
1970	Forth
1972	C
1972	Smalltalk
1972	Prolog
1973	ML
1975	Scheme
1978	SQL

Table 1.2: List of latest programming languages from 1967 to 1978

The 1980s were years of relative consolidation in imperative languages. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade. *C++* combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called fifth-generation programming languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Research in Miranda, a functional language with lazy evaluation, began to take hold in this decade. One important new trend in language design was an increased focus on programming for large-scale systems through the use of modules, or large-scale organizational units of code. Modula, Ada, and ML all developed notable module systems in the 1980s. The 1980s also brought advances in programming language implementation. The **Reduced Instruction Set Computer (RISC)** movement in

computer architecture postulated that hardware should be designed for compilers rather than for human assembly programmers. Aided by processor speed improvements that enabled increasingly aggressive compilation techniques, the RISC movement sparked greater interest in compilation technology for high-level languages. Language technology continued along these lines well into the 1990s. Some notable languages that were developed in this period include:

Year	Language
1980	C++ (as C with classes, renamed in 1983)
1983	Ada
1984	Common Lisp
1984	MATLAB
1984	dBase III, dBase III Plus
1985	Eiffel
1986	Objective-C
1986	LabVIEW (Visual Programming Language)
1986	Erlang
1987	Perl
1988	Tcl
1988	Wolfram Language
1989	FL (Backus)

Table 1.3: List of programming languages from 1980 to 1990

The rapid growth of the Internet in the mid 1990s was the next major historic event in programming languages. By opening up a radically new platform for computer systems, the Internet created an opportunity for new languages to be adopted. In particular, the JavaScript programming language rose to popularity because of its early integration with the Netscape Navigator web browser. Various other scripting languages achieved widespread use in developing customized applications for web servers such as [PHP](#). The 1990s saw no fundamental novelty in imperative languages, but much recombination and maturation of old ideas. This era began the spread of functional languages. A big driving philosophy was *programmer productivity*. Many “rapid application development” (RAD) languages emerged, which usually came with an IDE, garbage collection, and were descendants of older

languages. All such languages were object-oriented. These included Object Pascal, Visual Basic, and Java. Java in particular received much attention.

More radical and innovative than the RAD languages were the new scripting languages. These did not directly descend from other languages and featured new syntaxes and more liberal incorporation of features. Many consider these scripting languages to be more productive than even the RAD languages, but often because of choices that make small programs simpler but large programs more difficult to write and maintain. Nevertheless, scripting languages came to be the most prominent ones used in connection with the Web. Some notable languages that were developed in this period are given in Table 1.1.

Year	Language
1990	Haskell
1991	Python
1991	Visual Basic
1993	Lua
1993	R
1994	CLOS (part of ANSI Common Lisp)
1995	Ruby
1995	Ada 95
1995	Java
1995	Delphi (Object Pascal)
1995	JavaScript
1995	PHP
1997	Rebol

Table 1.4: List of programming languages from 1990 to 1997

In this section I am going to explain you installation methods for both Windows and Linux. Thought it is not a big deal as such but helps in working with both OSes. Most of the world is still revolving around MS Windows. In desktop computing, Windows is the second largest OS that rules the world with a share of 36.07%. <sup>2</sup> The Linux seems to be, as always, less than 1% nothing has changed much from 2014, the time when I wrote a paper on [GNU/Linux](#) usage for computing. <sup>3</sup> So,

most of the folk who does computing belongs to the Windows platform. So, I thought of including a section for Windows folk. However, most of the tasks that I have shown here are implemented in Linux platform.

## 1.2 About Python

Python is an easy to learn, powerful programming language. It has high-level data structures and a simple but effective approach called object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, <https://www.python.org/>, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation. The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customized applications. Python has standard objects and modules with rich formal definitions. Python allows writing extensions in C or C++.



The programming language Python was conceived in the late 1980s, and its implementation was started in December 1989 by *Guido van*

*Rossum* at **CWI** in the Netherlands as a successor to the ABC (programming language) capable of exception handling and interfacing with the Amoeba operating system. Python was named for the BBC TV show Monty Python's Flying Circus.<sup>4</sup><sup>5</sup> Guido Van Rossum is Python's principal author. He was always regarded with higher respect in the community and referred to as BDFL, which means Benevolent Dictator for Life.<sup>6</sup> Unfortunately, for the dismay of the community, he had announced about his voluntary exit from the responsibility on July 12th, 2018.<sup>7</sup>

One of the noteworthy versions of Python is Python 2.0. Python 2.0 was released on October 16, 2000, with many major new features, including a cycle-detecting garbage collector (in addition to reference counting) for memory management and support for Unicode. However, the most important change was to the development process itself, with a shift to a more transparent and community-backed process.<sup>8</sup>

Python 3.0, a major, backwards-incompatible release, was released on December 3, 2008 after a long period of testing. Many of its major features have also been backported to the backwards-compatible Python 2.6 and 2.7.

Python has a design philosophy that emphasizes code readability, and a syntax that allows programmers to express concepts in fewer lines of code.<sup>9</sup> If you ever had visited Python website (<https://www.python.org/>), you will find few examples on the very first page (home) which demonstrates easiness of following syntax of Python. For instance one of the demo-code snippets such as the one below shows how easy it is to put logic in words using very simple syntax.

```
>>> def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

>>> fib(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Don't worry. We are yet to start writing code. However, this code

demonstrates logic for creating febunaci series using IDLE editor.<sup>10</sup> This gives an idea as how easy it is to write a function that performs certain actions such as the one above.

## 1.3 Features

Python is a dynamic, high level, free open source and interpreted programming language. It supports object-oriented programming as well as procedural oriented programming. In Python, we don't need to declare the type of variable because it is a dynamically typed language. There are many features in Python, some of which are discussed below

1. Easy to code: Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C, Javascript, Java, etc. It is very easy to code in python language and anybody can learn python basics in a few hours or days. It is also a developer-friendly language.
2. Free and Open Source: Python language is freely available at the official website and you can download it from the given download link below click on the Download Python keyword. Download Python from <https://www.python.org/downloads/>. Since it is open-source, this means that source code is also available to the public. So you can download it as, use it as well as share it.
3. Object-Oriented Language: One of the key features of python is Object-Oriented programming. Python supports OOP philosophy and concepts of classes, objects encapsulation, etc.
4. GUI Programming Support: Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in python. PyQt5 is the most popular option for creating graphical apps with Python.
5. High-Level Language: Python is a high-level language. When we write programs in python, we do not need to remember the system architecture, nor do we need to manage the memory.
6. Extensible feature: Python is a Extensible language. We can write code using C or C++ or Java. Read Guido's own speech

at <https://www.python.org/doc/essays/omg-darpa-mcc-position/> to know about Python's integration with C and Java.

7. Python is Portable language: Python language is also a portable language. For example, if we have python code for windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.
8. Python is Integrated language: Look “Extensible feature” above.
9. Interpreted Language: Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile python code this makes it easier to debug our code. The source code of python is converted into an immediate form called bytecode.
10. Large Standard Library: Python has a large standard library which provides a rich set of module and functions so you do not have to write your own code for every single thing. There are many libraries present in python for such as regular expressions, unit-testing, web browsers, etc.
11. Dynamically Typed Language: Python is a dynamically-typed language. That means the type (for example - int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.
12. Expressive: Python provides us with a myriad of constructs that help us focus on the solution rather than on the syntax. This is one of the outstanding python features that tell you why you should learn Python.

## 1.4 Installing Python

Installing Python is very easy doesn't matter if it is Windows, Linux or MacOS.

### 1.4.1 Installing in Linux

Working with Python in Linux is very easy for being one simple reason that you don't need to install at all, if you are using a *debian* system like Ubuntu. It is default programming language which comes automatically while installing Ubuntu. To know the current version you got to execute the following command in the Terminal. Open the terminal by executing keyboard shortcut key such as *Alt+Ctrl+T*.

```
python --version
```

There will be two Python distributions, in any Linux, at least in Debian based systems. Same statement if executed in Ubuntu (18.04) will return 2.7. Python by default is 2.7. There is other command for version 3 i.e. `python3 --version`. If you skip `--version`, the terminal gets you **REPL**. Oh! I didn't tell you what is REPL right! REPL is known as *Read-Eval-Print Loop* in short we refer to REPL. It is *an interactive toplevel or language shell, is a simple, interactive computer programming environment that takes single user inputs (i.e. single expressions), evaluates them, and returns the result to the user.*

11

**In case if you are not Ubuntu user or by chance your distribution doesn't supply Python bundled with your Linux distribution. You still need not worry, you may follow below instructions to install Python in your computer.**

If your system is Debian based system:

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

You can also use `python2.7` in stead of `python3`. There are addeed advantages of having Python 2.7 if not both. This may be owing to the fact that python grew exponentially during 2.7.

If your system is Red Hat based distribution use the following command:

```
sudo yum install python3
```

If you belong to SUSE community then you may use

```
sudo zypper install python3
```

I used SUSE during my earlier days of using Linux. It is for those who aspire for beauty and aesthetics at work.

### 1.4.2 Installing in Windows

Installation in Windows is not that difficult. Python is available as executable binaries. Visit <https://www.python.org/> which is an official portal for all Python development. One important section that needs your attention is *PyPI* besides *Doc*. I mean *menus* at <https://www.python.org/> website.



Fig. 1.4.2 Python website

Installation is very simple. You will be able to download executable file once you click on “Download Python 3.6.4” button, refer to Figure 1.4.2. Once after downloading, execute the file in the windows explorer, also known as just “Window”.<sup>12 13 14</sup>

## 1.5 Editors

“Programs must be written for people to read, and only incidentally for machines to execute.”  
- *Harold Abelson*

IDE stands for **Integrated Development Environment (IDE)**. It’s a coding tool which allows you to write, test and debug your code in an easier way, as they typically offer code completion or code insight by highlighting, resource management, debugging tools etc. Even though the IDE is a strictly defined concept, it’s starting to be redefined as other tools such as notebooks start gaining more and more features that traditionally belong to IDEs. For example, debugging your code is also possible in Jupyter Notebook.

Python has got lot many editors. An editor might convey many, but we want to understand editor that make sense in the domain of programming. In computer science, when we mention editor it convey two things; (1) text editor, and (2) source code editor. However, every text editor one way the other can be source code editor. For instance, *Notepad* in Windows is a default text editor and has quite a few ways to write programs and develop source code. I always prefer Notepad for writing Java code when I dwell inside Windows.

While comign to Linux there are abundant of options for the same. The one I always prefer is *gedit*. Albeit of many reasons, I prefer *gedit* for it is easy to copy and paste the code while writing the programs. There are two other editors which assumes foremost position in Linux community, they are *vim* and *nano*. There are very hard-core fans for these editors in the field.

Programmers needs editors in order to make their work rather fast. There are couple of things which are expected by the system while writing or developing the code. First, presumably, some sort of workspace where the testing and development takes place regardless of the logic of the code. Second, programming needs quick assistance of choosing and selecting right command or statement regardless of the expertise of the programmer. This way, editors come handy to automate few routine or not-so-important tasks of implementation.

Python has two ways to execute the code. As it was mentioned earlier Python can be used through terminal as through **REPL**. Python's default prompt `>>>` appear, the moment we execute the command `python` in the Terminal. Doesn't matter whether it is Windows or Linux.

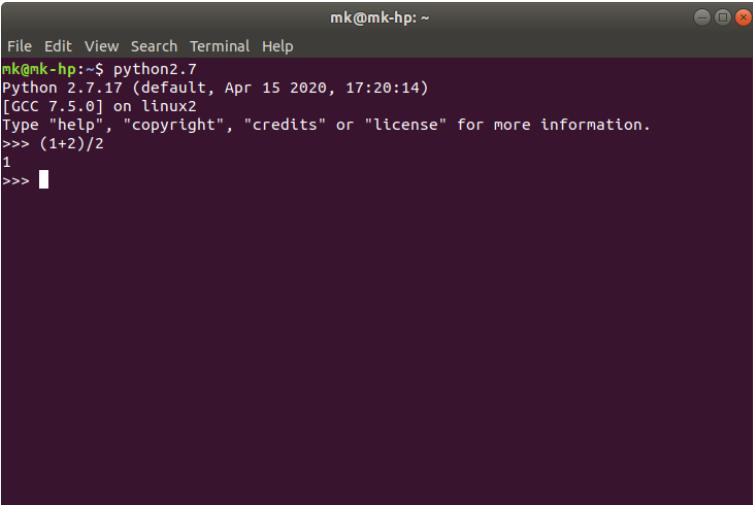
```
~\$ python
Python 2.7.12 (default, Dec 4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license"
    ↪ ...
>>>
```

The above code was executed on Dec, 04, 2017 at 14 : 50 : 18 through GCC and it was done in Linux system (UBUNTU 16.04). <sup>15</sup> This is how Python greets the user when invoked from the terminal. Of course, we can start writing anything after the prompt `>>>`. A *prompt* is text-based or command-line interface for shell. A *shell* is a software program that interprets commands from the user so that the operating system can understand them and perform the appropriate tasks. <sup>16</sup>

As it was mentioned earlier Python executes code in two modes; (1) interactive, (2) batch. Observe the following chunk of code.

```
>>> 1+2
3
>>> (1+2)/2
1.5
>>> print("Hello World!")
Hello World!
```

If you have Python 2.7, the result of `(1+2)/2 = 1` not 1.5. See Figure 1.5. Fortunately, from version 3, Python prints 1.5. For time being, let us explore the way Python executes the tasks. The very first expression is addition of 1, 2, of course the result is 3. The second expression is the addition of 1, 2 and division of the same through 2, of course the result should be 1.5. The third task prints “Hello World” in the Terminal. So simple, isn't it? When code is executed in this way it is called *interactive mode*. This type of execution is very much helpful to test the code, well before writing or developing modules or applications.

A screenshot of a terminal window titled "mk@mk-hp: ~". The window shows the following Python session:

```
File Edit View Search Terminal Help
mk@mk-hp:~$ python2.7
Python 2.7.17 (default, Apr 15 2020, 17:20:14)
[GCC 7.5.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> (1+2)/2
1
>>> 
```

The terminal has a dark background and light-colored text. The title bar at the top includes the user name "mk@mk-hp", the host name "mk-hp", and the current directory "~". The window has standard window control buttons (minimize, maximize, close) in the top right corner.

Fig. 1.5 Arithmetic Calculations in Python 2.7

Now that you know how to start Python, you should also know how to close the same. To quit Python you need to execute `exit()` not `exit`, please be cautious. The other type of execution is called *batch mode*. The primary purpose of batch mode, of course, is to develop modules and applications. Once the module is finished, it may be called as *source code*. However, developing source code and applications is beyond the scope of this book. However, it is better to know various modes of dealing with Python. We need to follow certain discipline to execute code in batch mode. The following are the few steps that might help understand the batch mode execution in Python.

1. Open text editor (such as *gedit*): It is always better to keep a separate directory for all Python related work. You may use `sudo mkdir your_directory_name`.<sup>17</sup>
2. Write Python statements: We will try a program that returns a sum of two values.
3. Close the file and go back to the Terminal.
4. Execute the file by using `python file`.

The following shows the execution of the code in Linux Terminal.

```
sudo mkdir python_work  
sudo gedit myFirstProg.py
```

1. The first statement creates a directory with a name `python_work`.
2. The second statement opens a text editor.<sup>18</sup>
3. You may write your program (such as the one in the below Figure 1.1)

```
a=2  
b=3  
c=a+b  
print("The Sum of %d and %d is %d" %(a, b, c))
```



Figure 1.1: Python code in *gedit* editor

4. Go back to Terminal and execute the following code.

```
~/python_work$ python myFirstProg.py  
The Sum of 2, 3 is 5
```

`7python_work$` is working directory that we created in step 1 above.

We need to have an idea about modes while trying to understand Editors. An Editor is not just a text editor but more than that. As it was mentioned earlier, most essentially an Editor provides an

environment for programming. That is why there is a habit of referring Editors as IDEs in programming.

Python has quite a few Editors to use. The following shows a better classification.<sup>19</sup>

1. Multiplatform Editors
2. Unix-Only Editors
3. Windows-Only Editors
4. Macintosh-Only Editors
5. Online Editors
6. Glorified Editors
7. Enhanced Python shells
8. Mobile Device Editors

Some of the Editors like *Emacs*, *Eclipse*, *Geany*, *PyCharm* are Multiplatform Editors. Few like *gedit*, *Vim*, *nano* are UNIX based Editors. *Notepad*, *Notepad++* are Windows-Only Editors. Some of the Editors like *Chocolate*, *Coda*, *TextMate* are MAC based Editors. Interestingly there are few online Editors like *Wakari*, *PLON*, *PythonAnyware* and etc.

### 1.5.1 IDLE

Very ironically, Python WIKI did not mention anything about one of the very important yet not-so-sophisticated programming environment called **Integrated Development and Learning Environment (IDLE)**. As name sounds, it is not only for development but also helps learners. We will be using this IDE for entire execution in this book.

Documentation for IDLE at official Python forums mentions following features of IDLE.<sup>20</sup>

- Coded in 100% pure Python, using the *tkinter* GUI toolkit
- Cross-platform: works mostly the same on Windows, Unix, and Mac OS X

- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- Multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- Search within any window, replace within editor windows, and search through multiple files (grep)
- Debugger with persistent breakpoints, stepping, and viewing of global and local namespaces configuration, browsers, and other dialogs

IDLE has two types of windows. One, Shell window, which helps testing code such as in interactive mode. Two, Editor window, which helps executing the code in batch mode. Moreover, IDLE has a very nice feature of opening multiple windows and working with each window independently at the same time.

### ***How to start IDLE***

In Linux; you can either start from a Terminal or through DASH (Super Button) in UBUNTU like OS. Starting IDLE from Terminal is not advisable unless if you are already working in your Terminal. It is always better to open IDLE from DASH like tool in *Unity* environment. Most of the Linux OSes will have *Applications* menu at top-left corner. You may find your favorite applications in there. If you are using GNOME environment probably *Alt + F2* might help you to open favorite application. So, in all possibilities given a *debian* system, you might be able to open idle simply executing command `idle` in the Terminal. You will then be opened with IDLE instance in interactive mode with Python prompt `>>>` in it. Just as the one in Figure 1.5.1.

The process is not that different in Windows. Perhaps starting IDLE in Windows is very easy. In Windows; just go to START (by pressing Windows SUPER button) and write “IDLE”. Click on IDLE if it is in the populated list.

We might need to know few essential short-cuts that helps us finish our job so easily in IDLE. These are related to Python Shell Window. To me the foremost command is browsing through *history*. IDLE doesn’t respond to upper or lower arrow key as we do in other editors. You

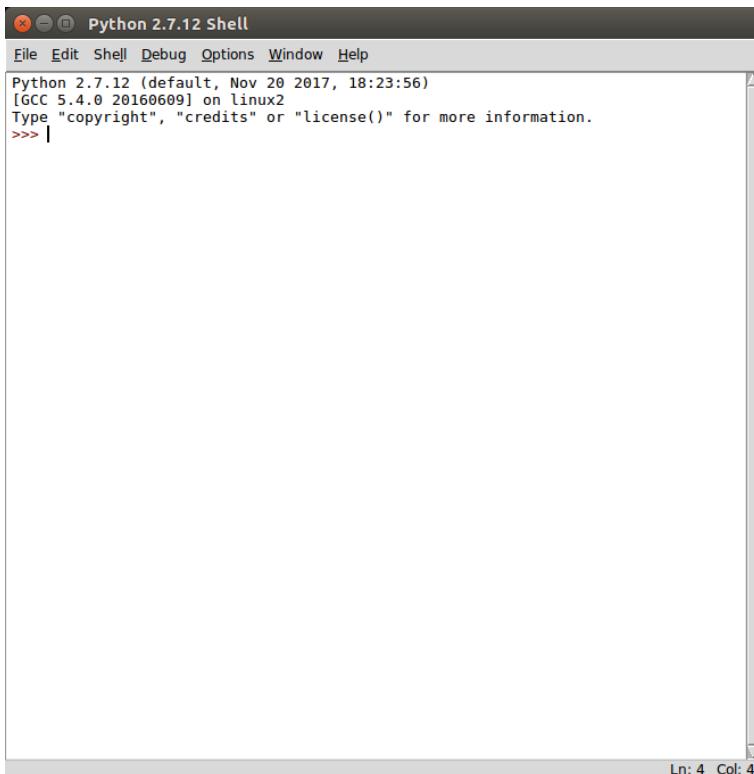


Figure 1.2: IDLE editor in UBUNTU

need to execute *Ctl + p* or *Ctl + n* these short-cuts really saves your time. If you are Linux user you must be having a tendency to use upper arrow key to browse through *history*, but that doesn't work. Be cautious, use *Ctl + p* for history. And to go ahead in the history, use *Ctl + n*. This saves lot of your time.

The other two short-cut commands that you need to know are *Ctl + c* and *Ctl + d*. *Ctl + c* helps us to close the execution and *Ctl + d* closes the entire windows. Though there isn't much description for menus in this section but have a look at the menu "options". This menu helps configure the IDE.

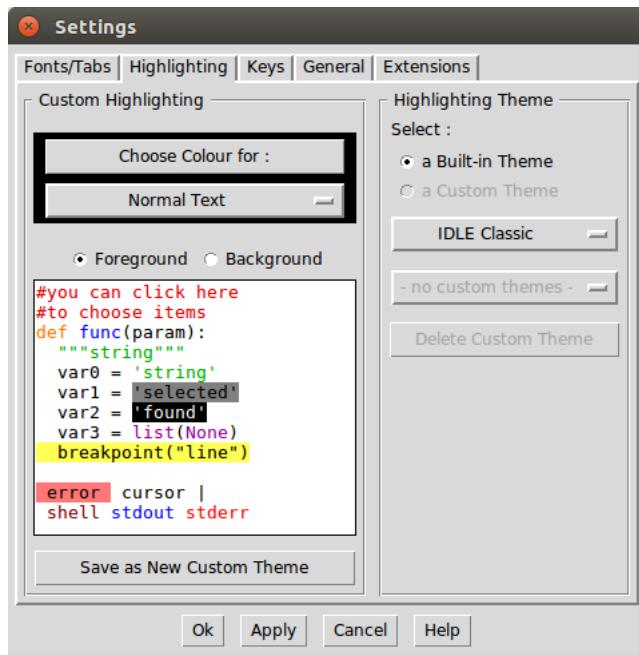


Figure 1.3: IDLE Options menu

In Linux you can also start IDLE from the Terminal by executing `idle-python2.7`, if you are using version 3 use `idle-python3`. You don't need to write the entire command just use TAB button. If you

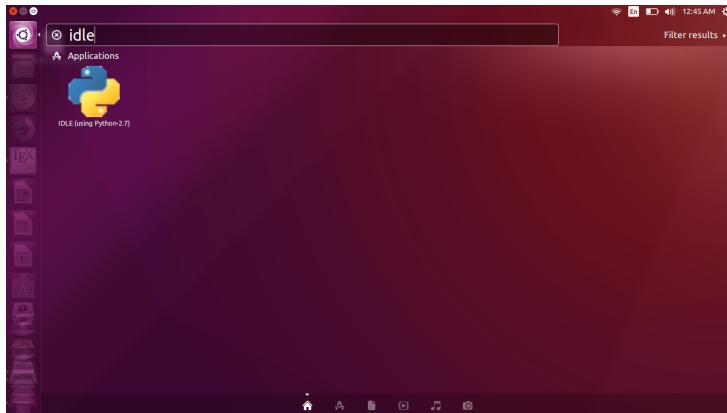


Figure 1.4: IDLE in UBUNTU

have a Ubuntu like Linux distribution just use DASH. Look at the figure 1.4

IDLE appearance and menus are all same irrespective of the platform i.e. OS. Perhaps, font may be different and might depend on host OS. In Windows the default font for these systems is *Courier New* but in Linux it might be *Lucida Console*. However, this has nothing to do with programming and functioning. Throughout this book we will be using IDLE for writing code.

To know Python version in IDLE:

```
>>> import sys
>>> print(sys.version)
2.7.12 (default, Dec 4 2017, 14:50:18)
[GCC 5.4.0 20160609]
```

In the above code `sys` is a built-in module, and it is part of Python's standard library. There are 325 such built-in modules and all of them are freely setup through Python default installation.<sup>21</sup> We use the command `import` to call modules or packages in Python user session. So, the first statement `import sys` imports/invoices the module `sys` into user session. How do you know that any module is invoked or

imported to user session? Yes, the answer is; to list out all objects in the **Namespace**. One of the very useful commands available to list out all, both system and user created, objects is `dir()`.

```
>>> import sys
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__
    ↪ __loader__', '__name__', '__package__', '__
    ↪ __spec__', 'sys']
>>>
```

You will find the name “`sys`” listed in the user session, along with other objects, also known as *namespace*. There are few names enclosed with double underscore, also known as “dunder”. They are not very much important right now at this stage. Now going back to the topic; the second statement i.e., `print(sys.version)` prints the system version.<sup>22</sup> The same activity can be done even using another built-in module known as `platform`.

```
>>> import platform
>>> platform.python_version()
'2.7.12'
```

Most of the code developed through this book is done only using IDLE. In IDLE it is possible to open a new Python script file from the *FILE* menu. The path for the same is *FILE* – > *NEWFILE*, just as shown in the Figure 1.5. Start writing the code just as few **User Defined Functions (UDF)** using Python most favorite `def` command. Save the file somewhere in certain location anywhere in you computer disk. The file which is saved will be useful for you now as module, but it need to be compiled using *RUN* – > *RUNModule* or by simply pressing F5. All those functions written in this script will serve as methods for your work. You can just *import* this newly created module just as any other one like `sys`, `os`, `random` etc. Read more about this in *Creating Packages* at Section 2.5 in Chapter 2.

### 1.5.2 IPython

There is another rather very handy Editor for learners known as *IPython*.<sup>23</sup> There are really very big hard-core fans for IPython.

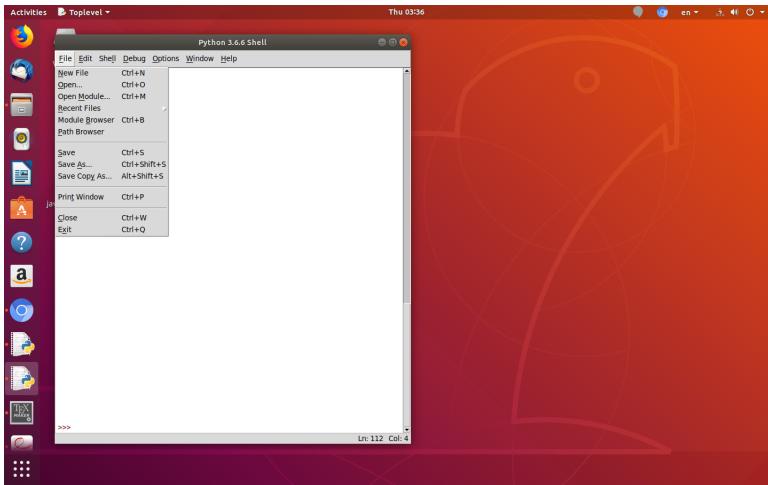


Figure 1.5: IDLE File menu in UBUNTU

You may have to visit <https://ipython.org/> if you want to use IPython. IPython is a growing project, with increasingly language-agnostic components. IPython 3.x was the last monolithic release of IPython, containing the notebook server, qtconsole, etc. As of IPython 4.0, the language-agnostic parts of the project: the notebook format, message protocol, qtconsole, notebook web application, etc. have moved to new projects under the name Jupyter. IPython itself is focused on interactive Python, part of which is providing a Python kernel for Jupyter.<sup>24</sup> Some of the interesting features of IPython includes:

1. Interactive shells (terminal and Qt-based).
2. A browser-based notebook with support for code, text, mathematical expressions, inline plots and other media.
3. Support for interactive data visualization and use of GUI tools.
4. Flexible, embeddable interpreters to load into one's own projects.

## 5. Tools for parallel computing.

Installing IPython is simple. The below description shows the way we need to follow to install IPython in Windows.

```
C:\>pip install ipython
Collecting ipython
  Downloading ipython-5.5.0-py2-none-any.whl (758kB)
    100% |#####| 768kB 383kB/s

.....
.....
.....

Installing collected packages: decorator, ipython-genutils,
  ↪ traitlets, scandir,
.....
.....
.....
```

If you are linux user and use Debian distro. use `pip install ipython`. If you are suffering due to proxy then you may execute `sudo apt-get install ipython`. Linux really comes as handy when you are suffering from proxy problems at your work. This method really rescue huge amount of time and you can really escape from the tragedy of using proxy settings while installing Python packages through `pip` by being behind proxy server. And when you start IPython (just execute command `ipython` in the Console) you will find the following prompt in Windows. Look at the figure 1.6. The process is same even in Linux. Execute `exit()` to quit the editor.

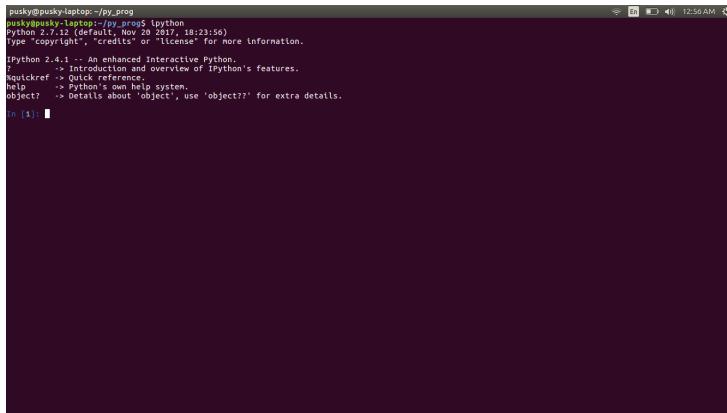
### 1.5.3 Spyder

Of course, the other editor that I like is Spyder. One of the portals for Spyder boasts as

“Spyder is the *Scientific PYthon Development EnviRonment*”<sup>25</sup>

Spyder is also useful as:

1. A powerful interactive development environment for the Python language with advanced editing, interactive testing, debugging

A screenshot of a terminal window on an Ubuntu Linux desktop. The title bar says "Terminal". The window contains the following text:

```
pusky@pusky-laptop:~/py_prog
Python 2.7.12 (default, Nov 20 2017, 18:23:56)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?      :> Introduction and overview of IPython's features.
quickref :> Python's own help system.
help   :> IPython's own help system.
object? :> Details about 'object', use 'object??' for extra details.

In [1]:
```

The terminal window has a dark background and light-colored text. The title bar includes standard icons for window control and a timestamp "12:56 AM".

Figure 1.6: IPython in Ubuntu Linux Terminal

and introspection features

2. A numerical computing environment thanks to the support of IPython (enhanced interactive Python interpreter) and popular Python libraries such as NumPy (linear algebra), SciPy (signal and image processing) or matplotlib (interactive 2D/3D plotting).

Spyder may also be used as a library providing powerful console related widgets for your PyQt-based applications. For example, it may be used to integrate a debugging console directly in the layout of your graphical user interface. You may visit <https://pypi.python.org/pypi/spyder> for spyder package available at PyPI. There is also nice description for the same at Wikipedia, visit [https://en.wikipedia.org/wiki/Spyder\\_\(software\)](https://en.wikipedia.org/wiki/Spyder_(software)) for more information.

## Installation

Spyder is quite easy to install on Windows, Linux and MacOS X. Just the read the following instructions with care. Spyder is already included in these Python Scientific Distributions such as *Anaconda*, *WinPython* and *Python(x,y)*. However, if you are using default Python

installation from raw Console in Linux machine, you might need to install Spyder manually. Open Terminal (Alt + Ctl + T) in Linux and execute the following (if you are using Debian based Linux distribution like Ubuntu).

```
sudo apt-get install spyder
spyder
```

Be careful that you may not be able to install as python package as we usually do in Linux. Once after installing open using command `spyder` in the Terminal. Spyder has nice sub-windows such as panes. The left side pane is for writing scripts. The right side you may be having three panes the top corner for *help*, just below that you may be having *Python Console or Shell*, the bottom one is for *history*. What else we need? The panes are flexible it is possible to rearrange one below the other by easily dragging here and there. I always keep *history* pane always just below *script* pane.

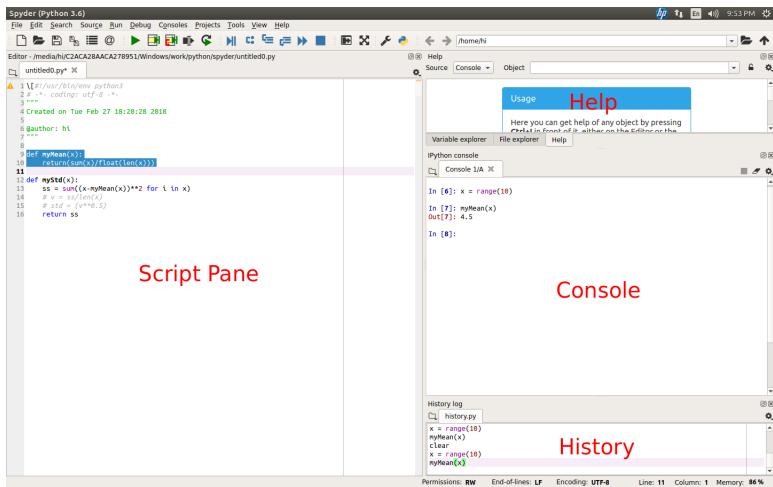


Figure 1.7: Spyder IDE in Ubuntu

Spyder is really helps and makes your work rather more easy if you really come from or having R like background.

### 1.5.4 Sublime Text Editor

I must be cruel if don't mention about *sublime text editor* the reason being that it is the darling of advanced Python programmers and it is very much used in web development. Sublime Text is a proprietary cross-platform source code editor with a Python application programming interface (API). It natively supports many programming languages and markup languages, and functions can be added by users with plugins, typically community-built and maintained under free-software licenses.<sup>26</sup>

#### Installation

Installation of sublime text editor is very much easy in Windows. It is just about getting .exe files and installing just like any other software. Sublime text is available for all platforms *aka* Windows, Linux, and MacOS. Just obtain executable libraries from <http://www.sublimetext.com/3>.

There are few ways to install sublime text in Linux based OSes. It is possible to install through (1) Terminal, (2) by obtaining distribution based pre-compiled libraries, and (3) software repositories.

#### Through Terminal

Suppose if you want to install sublime text through Ubuntu Terminal. Execute following commands in the terminal (Ctl + Alt + T).

```
wget -qO - https://download.sublimetext.com/
    ↪ sublimetext-pub.gpg | sudo apt-key add -
echo "deb https://download.sublimetext.com/ apt/
    ↪ stable/" | sudo tee /etc/apt/sources.list.d/
    ↪ sublime-text.list
sudo apt-get update
sudo apt-get install sublime-text
```

To uninstall use the following statement in the terminal

```
sudo apt-get remove sublime-text && sudo apt-get
    ↪ autoremove
```

## Through Repositories

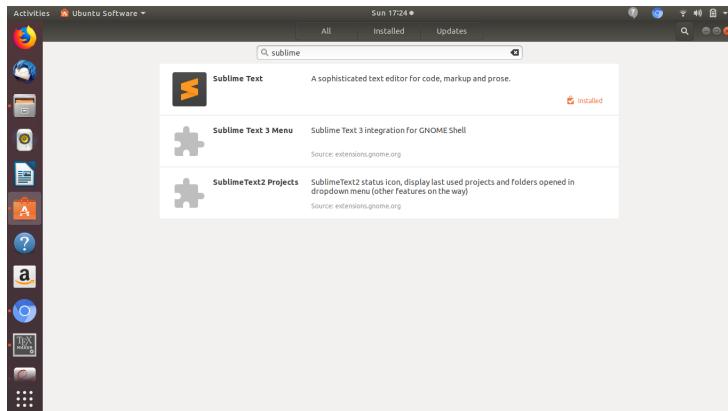


Figure 1.8: Sublime Text Editor in Ubuntu Software Center

Suppose if you are using GNU/Linux like Ubuntu, then you must be having a software installer called “Ubuntu Software”, such as the one shown in Figure 1.8. You just have to press install that all! Go to Terminal and execute `subl` to see the sublime editor. **One fundamental disadvantage of sublime is that it is proprietary software.** If you are a hard-core fan of open source I surely don’t recommend this editor for your needs.

### 1.5.5 Atom

As I mentioned before if you are open source evangelist or advocate or at least enthusiast, then there are few alternatives to sublime text editor. One of the editors that I love most is *Atom*. Atom has a beautiful website visit <https://atom.io/> and read more about documentation at <https://atom.io/docs>. The fundamental philosophy of atom is that it is not only open but *hackable to the core!* You will find ***A hackable text editor for the 21st Century*** as baseline at their website.

Most interestingly Atom is available for quite a few platforms. Visit <https://github.com/atom/atom/releases/tag/v1.30.0> for more

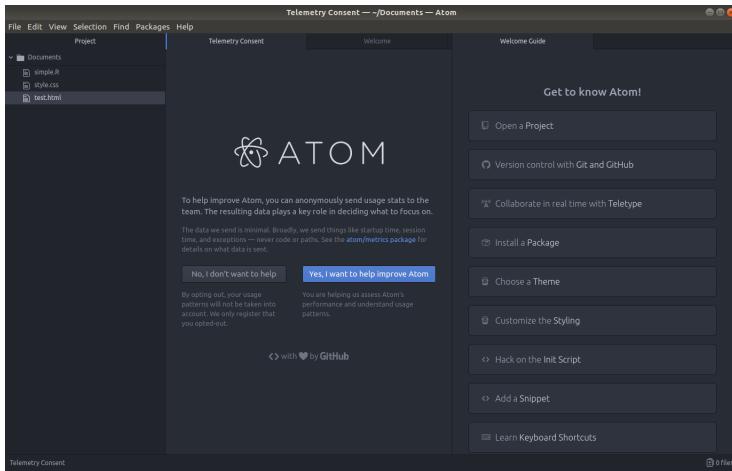


Figure 1.9: Atom Editor

information. There `.deb` package for Debian based distributions like Ubuntu. You can download `.deb` file from the website and open it with software center for installation. Open using Unity Search (Alt + F1). Atom looks like the one in Figure 1.9 when you open at first time. Atom is full featured mult tab Editor for Python.

### 1.5.6 Jupyter

It would be a big mistake, if I don't mention about . As it was mentioned earlier, IPython has revamped most of its project under the name Jupyter. Jupyter, as in whole, is an open-source project. Supports interactive computing and almost a dozen programming languages. Jupyter notebook interface is highly intuitive and user-friendly.

The Jupyter is a web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Best of all, user has enough alternatives to make output reports.

I personally like Jupyter for its support to *markdown*. Probably, you may understand it's significance, if you are into and . One best

place to learn is <https://www.markdownguide.org/>. It has highly sophisticated yet efficient methods to do literate programming.

## Installation

Installing Jupyter is very simple. The method is same in both Windows and Linux. Just install Jupyter as pip package.

```
pip install jupyter  
jupyter notebook
```

The above procedure is same in both Windows and Linux. The second statement `jupyter notebook` opens Jupyter in web browser. In case if you are using Debian platform you may do as shown below:

```
sudo apt-get install python3-jupyter
```

This will make your package universal. Few packages, needs to be installed using `Aptitude` in Debian based platforms. There is a package called `Pweave`, which is highly useful for literate programming in Python community. I had to install using `aptitude`, in spite of it being installed using `pip`.<sup>27</sup>

## 1.6 PyPI

“In real open source, you have the right to control your own destiny.”

- Linus Torvalds<sup>28</sup>

*PyPI* stands for *Python Package Index*. The URL for the same is <https://pypi.python.org/pypi>. To use a package from this index (PyPI), either use `pip install package` or download, unpack and `python setup.py install` it. In both cases, `pip` is a prerequisite. Both in Windows and Linux, `pip` will be installed automatically during Python installation. You may test it by merely executing `pip` in CMD in Windows or Terminal in Linux.

```
C:\>pip
```

**Usage:**

```
pip <command> [options]
```

**Commands:**

install	Install packages.
download	Download packages.
uninstall	Uninstall packages.

and more ...

You may verify the version of pip by using -V verbose in the CMD.

```
C:\>pip -V  
pip 9.0.1 from c:\python27\lib\site-packages (  
    ↪ python 2.7)
```

The other important *command* or *option* that you might need more often is **help**. For instance use `pip help install` and see what happens. Console might through lot of output, however you may find essential information from the very first and few lines.

```
C:\>pip help install
```

**Usage:**

```
pip install [options] <requirement specifier> [  
    ↪ package-index-options] ...  
pip install [options] -r <requirements file> [  
    ↪ package-index-options] ...  
pip install [options] [-e] <vcs project url> ...  
pip install [options] [-e] <local project path>  
    ↪ ...  
pip install [options] <archive url/path> ...
```

**Description:**

```
Install packages from:
```

The above output is only first few lines from the Console for the aforementioned command. The other useful command, of course, is **list**,

show, search.... Suppose to see if some package say `scrapy` is installed or not, we can do as below.

```
C:\>pip list

DEPRECATION: The default fo
e --format=(legacy|columns)
f under the [list] section)
asn1crypto (0.24.0)
attrs (17.4.0)
Automat (0.6.0)
.....

C:\>pip search scrapy | more

scrapy-amazon-robot-middleware-jondot (0.2.5) -
    ↪ Scrapy middleware module which uses image
    ↪ parsing to submit a captcha response to
    ↪ amazon.
scrapy-amazon-robot-middleware3 (0.3.3) -
    ↪ Scrapy middleware module which uses image
    ↪ parsing to submit a captcha response to
    ↪ amazon.
python-scrapyd-api (2.0.1) - A
    ↪ Python wrapper for working with the Scrapyd
    ↪ API
.....

C:\>pip install scrapy
```

The above code show three different statements. `list` shows the existing list of packages that were available from your computer. `search` searches the requested package (`scrapy` here) in PyPI repo. Finally `install` will install the requested package alongside fetching and executing all dependencies.

## In Linux

If you are using (GNU)Linux like OS, your job will be much more easy. BASH is a wonderful entity in Linux OS.<sup>29</sup> BASH is an acronym for *Born-Again SHeLL*. You will find the following lines of description for BASH at Wikipedia.

“BASH is a Unix shell and command language written by *Brian Fox* for the GNU Project as a free software replacement for the Bourne shell. First released in 1989, it has been distributed widely as the default login shell for most Linux distributions and Apple’s macOS (formerly OS X). A version is also available for Windows 10.”

BASH has very nice mechanism to work with Linux OS. You can literally accomplish many tasks merely using Terminal (also known as Console).<sup>30</sup>

Now coming back to Python, if you are in Linux you may execute the following code to list and search for a *particular* package name. Such as ...

```
pip list | grep xlwt
xlwt (0.7.5)
```

And at best, you don’t need to *MARK* the terminal to copy the code. I mean, If you are in Windows, it requires to *MARK* to copy the code in order to reuse it elsewhere. In Linux, copying text from BASH is rather strait, i.e. just copy.

### 1.6.1 Playing safe with packages

After we open editor we got a couple of things to know while workign with packages. One very first thing that every Python programmer is expected know is the entity `dir()`. Python has a set of *built-in-functions*. There are roughly 76 and odd such functions. Experts mention that Python has very dynamic and intelligent design, where the built-in-functions tend to play their game very safely without disturbing other functions. This is one of the important aspects of scripting languages. We will discuss more about *built-in-functions* later, but

for now, we need to know a particular function known as `dir()` which is important while working with functions. Since, we are working with packages, it is our duty to know as how to play safe with a package while working in Python.

This particular function i.e. `dir()` helps in identifying functions that were available in certain workspace. I say functions, but actual purpose of `dir()` is to find attributes of Workspace. A Workspace can be consisting of anything just as classes, objects, packages and etc. You will know about this in subsequent section. For instance, how do you know if you have imported<sup>31</sup> a particular package. Of course if you have imported; whether is it available for coding?

Use plain `dir()` to know all the packages and if you keep any package name of your interest within those parentheses then you may get all the functions which are native to that particular package. For instance, `dir(os)` gives all functions in it.

```
>>> dir

<built-in function dir>
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']

>>> dir(os)

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    dir(os)
NameError: name 'os' is not defined

>>> import os

>>> dir(os)
[...,
 '_Environ', '__all__', '__builtins__', '__doc__',

 ....
 ....

'_urandom', 'utime', 'waitpid', 'walk', 'write']

>>> dir()

['__builtins__', '__doc__', '__name__', '__package__', 'os']
```

The very first statement i.e. `dir` yields nothing but the type of the function. The function `dir` is, in deed, a built-in function. The second function `dir(os)` throws error, because we really not *imported* it yet. So we had imported or attached the libraries of `os` package in subsequent statement using the command `import os`. Now, the statement `dir(os)` shows certain valid output, in which you may be able to find all the commands or functions available from this package. Finally when you execute `dir()`, with empty parentheses, Python shows you `os` package in the list, which was not in the earlier when `dir()` was used first time at the beginning of the exercise.

We really don't need to learn much about in-built functions for a simple reason that we are not going to involve in development of software neither in applications. The purpose of this book is to give you a very basic idea that as how to use Python for data analysis. You got lot of material online, in case if you are excessively interested in these functions. One particular method to know about built-in functions is `dir(__builtin__)`. This statement enlist all the built-in functions of Python. You may use `help("name_of_the_function")` to know more about any function.

## Namespace in Python

Though it is not necessary but awareness on *namespace* is important in programming. Though, there is not much content on programming in this book, yet understanding language design gives sufficient mileage. There are three things we got to know while getting ourselves familiar with programming especially with a tool like Python. They are (1) Workspace, (2) Namespace and (3) Scoping. Today, there is lot of interest bubbling around these concepts but importance is limited to only programming cults. Cults are those who has similar level of cogitation and dwelling across community.

Coming to these concepts; *Workspace* is, more or less, an environment. The environment is composed of certain special features connected to *version*, *distributions*, *interpreters*, ... etc. There is certain interesting discussion about version in Python community. For that matter, whenever there is new version there will be certain level of comparison with its default version which is 2.7. The Python 2 has remarkable

presence in community from its inception. This particular version was started in the year 2000 somewhere in October. So it is a millennium edition. There is lot of fan-following to this version. Python 3 was introduced in the year 2008 and in December. Albeit of the confusion there is certain interesting discussions in the community that Python 2 is classic and Python 3 is future. The reason is that there was lot of development around Python 2 so much so that few of packages developed during Python 2 are not still available for Python 3.

In laymen language, a *namespace* is a space meant for name. In Python, for that matter in any programming language, a namespace is a practical approach to define the scope, and it helps to avoid name conflicts. The namespace is a fundamental idea to structure and organize the code, especially more useful in large projects. However, it could be a bit difficult concept to grasp.

A namespace is a simple system to control the names in a program. It ensures that names are unique and won't lead to any conflict. Name (also called identifier) is simply a name given to objects. Everything in Python is an object. Name is a way to access the underlying object. To simply put it, namespace is a collection of names. Mapping of every name with its corresponding object and collection of such maps represents a namespace. So a namespace is also a collection of maps where there are certain groups of objects with their names defined for a definite purpose. At any given point in time, there might be different namespaces available perfectly isolated fashion. In Python, each module creates its own global namespace.

A simple schema could be

Workspace	Built-in Namespace	+	Global Namespace	+	Local namespace
-----------	--------------------	---	------------------	---	-----------------

*Local Namespace* is the namespace that covers the local names inside a function. Python creates this namespace for every function called in a program. It remains active until the function returns. *Global Namespace* is the namespace covers the names from various imported modules used in a project. Python creates this namespace for every

module included in your program. It'll last until the program ends. *Built-in Namespace* is the namespace covers the built-in functions and built-in exception names. Python creates it as the interpreter starts and keeps it until you exit.

### What is scoping?

Namespaces make our programs immune from name conflicts. Python restricts names to be bound by specific rules known as a scope. The scope determines the parts of the program where it can be used without any prefix. Python outlines different scopes for locals, function, modules, and built-ins. A *local scope*, also known as the innermost scope, holds the list of all local names available in the current function. A scope for all the enclosing functions, it finds a name from the nearest enclosing scope and goes outwards. A module level scope, takes care of all the *global names* from the current module. The outermost scope which manages the list of all the *built-in names*. It is the last place to search for a name that you cited in the program.

```
x = 4
y = 6

def outer_fun():
    global x
    x = 2
    y = 8
    def inner_fun():
        global x
        x = 3
        y = 7
        print('x inside inner_fun :', x)
        print('y inside inner_fun :', y)
    inner_fun()
    print('x inside outer_fun :', x)
    print('y inside outer_fun :', y)

outer_fun()
print('x outside all functions :', x)
print('y outside all functions :', y)
```

The result of this script is as follows:

```
x inside inner_fun : 3
```

```
y inside inner_fun : 7
x inside outer_fun : 3
y inside outer_fun : 8
x outside all functions : 3
y outside all functions : 6
```

Don't be worried. We didn't work much on Python *definitions*. In python the `def()` statement is used to write *User Defined Functions* (UDF). We will discuss more on Python functions in forthcoming chapter.

While coming to above code chunk, the scope for variable `x` is not changing, while that of `b` is changing. This is what is called scoping. The value of `x` is not changing because the scope was redefined everywhere as *global*. The value tends to be same, irrespective of its value inside every inner function.

### Modular scope

Scoping is also important while importing modules or packages. It is very likely that modules are imported while programming. There are different ways to import such modules. For instance, few ways of importing modules are goes as below:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '
    ↪ __name__', '__package__', '
    ↪ __spec__']

>>> import os
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '
    ↪ __name__', '__package__', '
    ↪ __spec__', 'os']

>>> from os import getcwd, chdir
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '
    ↪ __name__', '__package__', '
    ↪ __spec__', 'chdir', 'getcwd'
    ↪ , 'os']
```

The very first implementation of `dir()` yields all the information of basic namespace. The same information after importing `os` module is

different. Functions of `os` need to be used by associating `os`.

```
>>> os.getcwd()
'/home/mk'
```

However, it is also possible to import functions from a module, such as the one shows in the above code. We imported two of the functions namely `getcwd` and `chdir` from `os` using `from os import getcwd, chdir` statement. Whenever, functions are imported there is no necessity to use name of the package along with the function. But there is a problem with this approach. Let us work with other module `math`.

```
>>> from math import sqrt
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>> sqrt(4)
2.0
>>> from cmath import sqrt
>>> sqrt(4)
(2+0j)
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
```

When the function imported from second module `cmath`, the earlier function which was imported from `math` is not working. Python is trying to use the functions from the latest namespace. So, the best way is:

```
>>> import math
>>> import cmath
>>> math.sqrt(4)
2.0
>>> cmath.sqrt(4)
(2+0j)
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
```

Importing modules by name is always safe in stead of functions.

## 1.7 A bit about `sys` module

The `sys` module provides information about constants, functions and methods of the Python interpreter. In brief, it gives a summary of the available constants, functions and methods. This module is used to know system specific parameters and functions.<sup>32</sup>

```
>>> dir(os)
['F_OK', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', ....

>>> for fun in dir(os):
    if fun == "chdir":
        print fun

chdir
>>>
```

The second operation in which there is a `for` loop contained `if` statement in it. This operation or set of statements helps in identifying whether there exists any function with a name `chdir`. We will be having more idea and discussion on *controls* in forthcoming sections.

```
>>> 'unicodedata' in sys.modules
False
>>> import unicodedata
>>> 'unicodedata' in sys.modules
True
```

The above exercise shows as whether the module `unicodedata` is available in the namespace `sys`. To know about all the modules that are installed execute use `sys.modules()` either with *values* or with *keys*.

```
for mod in sys.modules.values():
    print(mod)

<module 'heapq' from 'C:\Python27\lib\heapq.pyc'>
<module 'code' from 'C:\Python27\lib\code.pyc'>
<module 'tkFileDialog' from 'C:\Python27\lib\tk\
          ↪ tkFileDialog.pyc'>
<module 'functools' from 'C:\Python27\lib\functools.pyc'>
<module 'random' from 'C:\Python27\lib\random.pyc'>
```

```
<module 'unicodedata' from 'C:\Python27\DLLs\unicodedata.pyd'>
None
```

This will throw information about all those modules that are installed in the system. The name that immediately follows after `<module` is the name of the module. We will know more about `if` statement together with `for` loops in forthcoming sections.

## 1.8 Understanding OS through Python

Operating System (OS) is host for Python. I don't know whether this statement makes any sense or not. By mentioning *host* I mean, we have been working in certain environment. If you are using Windows you got to know as how to interact with your OS using Python. We got certain package called `os`. Visit <https://docs.python.org/2/library/os.html> for full documentation. Perhaps, the very first command to start with `os` is `os.name`. This return different values such as `'posix'`, `'nt'`, `'os2'`, `'ce'`, `'java'`, `'riscos'`. Suppose, if you execute this command in Windows, it might return `'nt'`. The other important command is `os.environ` this will eject entire list of paths in the system.

```
>>> os.environ
{'TMP': 'C:\\\\Users\\\\ibm\\\\AppData\\\\Local\\\\Temp', .....
>>> len(os.environ)
47
```

The output from this operation, perhaps, is a *dictionary*. We will be discussing about this data type later but for now, suppose, if you want to know a particular element of a dictionary, you might be able to use a couple of options such as `list`, `keys`, just like:

```
>>> list(os.environ.keys())
['TMP', 'COMPUTERNAME', 'USERDOMAIN', 'PSMODULEPATH', .....
```

A dictionary as by definition has a key and a value. So, there must be a way to retrieve either value or key or both. For instance, to know temporary path (TMP) I might be able to execute.

```
>>> os.environ['TMP']
'C:\\\\Users\\\\ibm\\\\AppData\\\\Local\\\\Temp'
```

So the path for *temporary* folder in my computer is `C:\\\\Users\\\\ibm\\\\AppData\\\\Local\\\\Temp`. Two very important commands which we often require are `os.getcwd()`, `os.chdir()`. The first, helps us know the current working directory and the second to change the directory to any other path or place. It is highly advisable to keep a different path which might be other than the one which is default and it is mostly your `C:/` directory. In Windows, you may use these commands as:

```
>>> os.getcwd()
'C:\\\\Python27'
>>> os.chdir("D:\\\\Windows\\\\work\\\\python")
>>> os.getcwd()
'D:\\\\Windows\\\\work\\\\python'
```

Fortunately if you are using an editor like *IDLE*, the editor will pop-up all possibilities for user inputs. Such as the one shown in the below figure 1.10.

All above code ***In Linux:***

```
>>> import os
>>> os.getcwd()
'/home/pusky/py_prog'
>>> os.chdir("/media/pusky/9C0C019DOC017394/WINDOWS/OLD/Work/
           ↪ python")
>>> os.getcwd()
'/media/pusky/9C0C019DOC017394/WINDOWS/OLD/Work/python'
```

To go back to old or default directory use the path with `os.chdir()`. One important advantage of using Linux is that you can open IDLE from *present working directory* (`pwd`) in Terminal.

***In Linux Terminal***

```
$~/py_prog$ pwd
/home/pusky/py_prog
```

As it is clear from the code; `\home\pusky\py_prog` is my present working director.

## In IDLE

```
>>> import os
>>> os.getcwd()
'/home/pusky/py_prog'
```

You see the path is same. This make our work rather convienient to work with. You may understand as why I am trying to mention all these things, well ahead of its real usage, later when we try writing *modules* or *packages*. Python has interestingly very simple mechanisms to development and use modules.

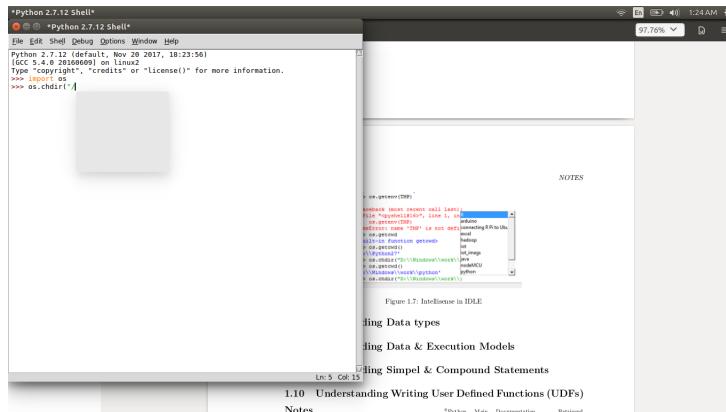


Figure 1.7: Intellisense in IDLE

When you are working with paths; if you press *tab* button on the keyboard immediately after the "\...", IDLE try to show all alternative ways for a file or directory name. In case, if you choose to enter a particular name and try after a couple of letters, IDLE takes the entire name at a shot, in stead of showing alternative names.

## Notes

load it from <https://www.libreoffice.org/download/download/>

<sup>1</sup> *calc* is a short name for Libre Office Calc which is a potential alternatives to MS Office Excel. You may freely down-

<sup>2</sup> The statistics are as of December, 2017. Source <http://gs.statcounter.com/os-market-share>. The first being

Android. Android is, of course, leading in mobile phones and other hand-held gadgets.

<sup>3</sup>Visit [https://papers.ssrn.com/sol3/papers.cfm?abstract\\_id=1995233](https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1995233) for my paper titled “User Perception Towards Open Source Operating Systems with Special Reference to GNU/Linux: A Chi-Square Analysis on User Expertise and its Influence on Technical Factors”

<sup>4</sup>Python Main Documentation. Retrieved from <https://docs.python.org/2/faq/general.html#why-is-it-called-python>

<sup>5</sup>Bill Venners, The Making of Python, January 13, 2003. Retrieved from <http://www.artima.com/intv/pythonP.html>

<sup>6</sup>Guido van van Rossum, Origin of BDFL, July 31, 2008. Retrieved from <http://www.artima.com/weblogs/vietwpost.jsp?thread=235725>

<sup>7</sup><https://hub.packtpub.com/python-founder-guido-van-rossum-goes-on-a-permanent-vacation-from-being-bdfl/>

<sup>8</sup>A.M. Kuchling and Moshe Zadka. “What’s New in Python 2.0”. Archived from the original on December 14, 2009. Retrieved March 22, 2007.

<sup>9</sup>Read more about Python from [http://en.wikipedia.org/wiki/Python\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Python_(programming_language))

<sup>10</sup>IDLE is a simple and sleek editor written purring in Python for Python programming. Visit <https://docs.python.org/3/library/idle.html>

<sup>11</sup>Visit [https://en.wikipedia.org/wiki/Read%20%93eval%20%93print\\_loop](https://en.wikipedia.org/wiki/Read%20%93eval%20%93print_loop) for more information

<sup>12</sup>In Windows OS executing .exe file

can be done just double clicking the same file.

<sup>13</sup>Python 3.6.4 was latest version at the time of writing this book. The learner is advised to check <https://www.python.org/> for regular updates.

<sup>14</sup>The very first release of Python version was 0.9 launched in 20-12-1991, however official support started from 29-07-1993. Read more about Python versions either at [https://en.wikipedia.org/wiki/History\\_of\\_Python](https://en.wikipedia.org/wiki/History_of_Python) or at [https://en.wikibooks.org/wiki/Python\\_Programming/Version\\_history](https://en.wikibooks.org/wiki/Python_Programming/Version_history).

<sup>15</sup>Python3 shows different information.

<sup>16</sup>Read more about *Prompt* and *Shell* at <https://techterms.com/definition>.

<sup>17</sup>In a typical system like UBUNTU the text editor can be opened in two ways. One from the terminal. Two from the *DASH*. To open any application from *DASH*, first press *Windows Start Key* also known as *Super Button/Key*. In Windows you may start *NOTE PAD* like application by executing *start notepad* in the Shell/CMD.

<sup>18</sup>There are number of text editors available in Linux. *Gedit* is one of the user friendly **Graphic User Interface (GUI)** based text editor, which seems like *notepad* of Windows. There are other terminal text editors

<sup>19</sup>Obtained from Python Wiki on Editors at <https://wiki.python.org/moin/PythonEditors>

<sup>20</sup>Retrieved from official Python documentation. Read more about IDLE at <https://docs.python.org/3/library/idle.html>

<sup>21</sup>Read more about Python's built-in modules at <https://docs.python.org/3/py-modindex.html>.

<sup>22</sup>If you open IDLE for Python3 the version number may be different.

<sup>23</sup>Read more about IPython at <https://ipython.org/>

<sup>24</sup>Retrieved from <https://ipython.org/>

<sup>25</sup>Source <https://pythonhosted.org/spyder/>

<sup>26</sup>Read more about Sublime Text at [https://en.wikipedia.org/wiki/Sublime\\_Text](https://en.wikipedia.org/wiki/Sublime_Text)

<sup>27</sup>Matti Pastell, Associate Professor, Dr. of Agriculture and Forestry Natural Resources Institute Finland had developed *Pweave*, which mimics like *Sweave* used by R community. Matti has developed few other open source projects. Visit his website <http://mpastell.com/> for more but very interesting details of his work.

<sup>28</sup>Read more at [https://www.brainyquote.com/topics/open\\_source](https://www.brainyquote.com/topics/open_source)

<sup>29</sup>Advised to take word Linux *in lie* of GNU/Linux. You may please read GNU/Linux naming controversy at [https://en.wikipedia.org/wiki/GNU/Linux\\_naming\\_controversy](https://en.wikipedia.org/wiki/GNU/Linux_naming_controversy) for futher details. There is also an interesting blog written by one *Micah Lee* at <https://micahflee.com/2015/09/why-i-say-linux-instead-of-gnulinus/>. I don't have any ego, to me it is just a matter of saving my time while writing about Linux. It is easy to me to write Linux in stead of GNU/Linux.

<sup>30</sup>Read Machtelt Garrels's online tutorial on BASH at <https://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html>

<sup>31</sup>Usually packages are imported using a command `import` in Python.

<sup>32</sup>Find documentation for `sys` module at <https://docs.python.org/2/library/sys.html>



## Chapter 2

# Programming in Python

“Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter. ”

- Eric Raymond

What a beautiful statement! I am not a hard-core programmer because I developed very few applications that too only for academics. Developing applications and deploying them for few good reasons is what is expected from a programmer. This Chapter deals with highly important aspects of Python programming. Learning Python, of course, is easy but one need to understand the building blocks of the language design. Python is very rich in terms of data types and structures. These data types are rudimentary aspects of Python which are used to manipulate data. Data can be manipulated using various operators viz arithmetic, logical, comparison, boolean and etc. Few data types such as tuples, lists, dictionaries, sets are highly powerful in manipulating data while executing logic. This Chapter has brainy content which might helps learner understanding the very soul of Python. Hence, learners are requested to posses little patience and also quite a bit of passion while going through the content. As a learner you

might feel boring while navigating through the sections of this Chapter. However, it is worth putting effort in reading and also assimilating concepts such as control flow, functions and matrix operations. These concepts will continue to exists in one or other programs till the end of this book. There are few statistical analyses in next Chapter, I will be using few functions developed in this Chapter for simulating input data sets for those analyses. So, I earnestly request learner to have enhanced levels of patience while reading content of this Chapter.

Above all, this Chapter, is rather more important compared to other Chapters in the whole text book. The reason being, it has content related to very fundamental nature of Python's work flow. Few of the aspects of this Chapter which are important for Python programming are as follows.

1. Data types and structures
2. Control flow
3. Functional programming
4. Matrix manipulations
5. Creating packages

Data types or structures are building blocks of programming. Data can be numerical, textual, pictorial or may be video graphic. Every language has precisely defined procedures to deal with data. A language need to have sufficient stamina to deal with such heterogeneous data. Python has beautiful methods to deal with all and such data rather more efficiently compared to rest of the languages say Java, Ruby, R, etc. Coming to Control Flow, it is not possible to process data without flow of control. A program is a set of instructions written as what is to be done on data. These instructions has a control such as directing the logic to and from within the program. Python has wonderful ways to control the flow using conditional statements mixed with loops.

Python is also known for its stamina writing scripts. For that matter there is one very strong fundamental belief that Python is primordially a scripting language. A scripting language is any language which is used to write back-end programs. Few languages like PHP, Perl, R and

Python has very strong hold in server-side programming. One of the other popular scripting languages which is default option for front-end applications is of course Java Script also known as JS. These scripting languages are popular for one thing that is “functions”. Functions are building blocks of a script. Functions or functional programming is, in fact, one of the strengths of Python. Python built or architecture is pretty much depends on functions. For that matter, the rudimentary aspects of Object Orient Programming (OOP) are functions. In fact, all those enhanced methods such as *class*, *static*, *magic* and few other aspects such as *variables*, *decorators* are all part of the very rich Python OOP architecture. All these aspects or concepts are basically functions. Writing a function in Python is dead easy. Information related to functional programming is given under the Section 2.3.2.

Coming to Matrix Manipulations; this topic is highly influential in data science and analytics. The reason being, everything is a matrix in data analytics. A matrix is an arrangement of data in rows and columns. Such matrices play very vital role in performing few numerical analysis. We may often encounter few data processing activities, say row-wise data processing, column-wise data processing while performing statistical analyses. Python has a package known as *numpy* which has very efficient methods to process matrices. However, I will be showing you as how to process matrices using conditional statements and loops. The last topic, of course, is Creating Packages. Python has really wonderful infrastructure to organize the code into modules and packages. This topic is starting point for developing software.

## Strategy

I developed a text book companion package named *PfDSaA*. I will be creating few functions while explaining aforementioned topics in this Chapter. All the source code (programs) created through this text book are organized in different modules in side this package. The structure of the package is as follows.

```
mk@mk-hp:.../PfDSaA\$ tree
.
  Pythonprogramming.py
```

As of now there a single module in this package namely *Pythonprogramming.py*. There is more description in one of the sections 2.5 which deals with creating a packages. For time being, you just have to know that we are going to organize all the code collected throughout this book in modules and there is one such module by name *Pythonprogramming.py* in which we will organize the code related to aforementioned topics.

## 2.1 Understanding Data types

Python is very rich in terms of data types. It is in fact one of the reasons for Python's perversion into all areas of knowledge. The data types of Python allows any person adapt very quickly to irrespective of their domain of knowledge.

Python's built-in (or standard) data types can be grouped into several classes. Sticking to the hierarchy scheme used in the official Python documentation these are numeric types, sequences, sets and mappings (and a few more not discussed further here). Some of the types are only available in certain versions of the language as noted below.

1. **Boolean** This is the type of the built-in values *True* and *False*. Useful in conditional expressions, and anywhere else you want to represent the truth or falsity of some condition. Mostly interchangeable with the integers 1 and 0. In fact, conditional expressions will accept values of any type, treating special ones like boolean False, integer 0 and the empty string “ ” as equivalent to False, and all other values as equivalent to True. But for safety sake, it is best to only use boolean values in these places.

### 2. Numeric type

- (a) **int**: Integers; equivalent to C longs in Python 2.x, non-limited length in Python 3.x
- (b) **long**: Long integers of non-limited length; exists only in Python 2.x
- (c) **float**: Floating-Point numbers, equivalent to C doubles
- (d) **complex**: Complex Numbers

### 3. Sequences

- (a) **str**: String; represented as a sequence of 8-bit characters in Python 2.x, but as a sequence of Unicode characters (in the range of  $U + 0000 - U + 10FFFF$ ) in Python 3.x
- (b) **bytes**: A sequence of integers in the range of 0 – 255; only available in Python 3.x
- (c) **byte array**: Like bytes, but mutable; only available in Python 3.x
- (d) **list**: A special type that contains heterogeneous entities as in brackets [ ]. A list can have different varieties of elements such as a string, number, and a tuple. Lists are *mutable*.
- (e) **tuple**: A special type that can encapsulate both numbers and characters but *immutable*.

### 4. Sets

- (a) **set**: An unordered collection of unique objects; available as a standard type since Python 2.6
- (b) **frozen set**: like set, but immutable (see below); available as a standard type since Python 2.6

### 5. Mappings

- (a) **dict**: Python dictionaries, also called hashmaps or associative arrays, which means that an element of the list is associated with a definition, rather like a Map in Java

Most of the above data types are not used in this book. However, they are used in various ways as by the context in data science arena. Mostly first few, i.e. *boolean*, *int*, *float* are useful to manage *integers* and *decimals*; and *tuple*, *list*, *dict* might be useful to manage multivariate data sets. The data types can be understood with the help of a special function `type()` in Python.

```
>>> a = 1.5
>>> s = 'kama'
>>> a = 12
```

```
>>> f = 1.5
>>> tup = ('a', 'b', 'c', 'd', 'e')
>>> l = ['a string', 342, ("a", "abc", 123, "abcd")]
>>> type(a)
<type 'int'>
>>> type(s)
<type 'str'>
>>> type(f)
<type 'float'>
>>> type(tup)
<class 'tuple'>
>>> type(l)
<type 'list'>
>>> dic = {"kama": 41, "sukanya":38, "ammalu":12, "chikku":6}
>>> type(dic)
<type 'dict'>
>>>
```

The above code chunk provides sufficient information on *integer (int)*, *string (str)*, *float*, *tuple*, *list* and *dictionary (dict)*. In the above exercise `a`, `s`, `f`, `tup`, `l`, `dic` are objects known as *user objects*. Execute `dir()` function to know about all those user objects in user session also known as *workspace*. It needs some certain knowledge on *operators*, before getting into data types.

### 2.1.1 Types of Operator

Python language supports the following types of operators.

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Assignment Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators
7. Identity Operators

Following subsections provide very little description for the first four types of operators.

### Arithmetic Operators

Arithmetic operators are useful to perform all arithmetic operations. `+, -, *, /` are basic arithmetic operators. Python supports few other operators such as `%`, `**`, `//` known as *modulus*, *exponent* and *floor division*. Suppose if  $a = 10$  and  $b = 20$ , then:

Operator	Description	Example
<code>+</code>	Addition	$a + b = 30$
<code>-</code>	Subtraction	$a - b = -10$
<code>*</code>	Multiplication	$a * b = 200$
<code>/</code>	Division	$b/a = 0.5$
<code>%</code>	Modulus	$b \% a = 0$
<code>**</code>	Exponent	$a ** b = 1e20$
<code>//</code>	Floor Division	$a//b = 0$

### Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators. Assume variable  $a$  holds 10 and variable  $b$  holds 20.

Operator	Example
<code>==</code>	$(a == b)$ is not true.
<code>!=</code>	$(a != b)$ is true.
<code>&lt;&gt;</code>	$(a <> b)$ is true. This is similar to <code>!=</code> operator.
<code>&gt;</code>	$(a > b)$ is not true.
<code>&lt;</code>	$(a < b)$ is true.
<code>&gt;=</code>	$(a >= b)$ is not true.
<code>&lt;=</code>	$(a <= b)$ is true.

```
>>> a = 1; b = 2
>>> a; b
1
2
```

```
>>> a = 1; b = 2
>>> a > b
False
>>> b > a
True
>>> a = 1; b = 2; c = a + b; c
3
>>> a = 1.5; b = 2; c = a + b; c
3.5
>>> a = 1.5; b = 3; c = a * b; c
4.5
>>> a = 10; b = 2.5; a/b
4.0
```

So when an integer is operated by another integer the result is an integer. The result of the operation over float and integer is always an integer. However, dealing these data types through arrays and loops is not that easy. One very important thing is to know about *print statement* which has certain interesting behavior in Python.

## Assignment Operators

Assume variable a holds 10 and variable b holds 20, then then see Table 2.1.1

Operator	Example
=	c = a + b assigns value of a + b into c
+=	c += a is equivalent to c = c + a
-=	c -= a is equivalent to c = c - a
*=	c *= a is equivalent to c = c * a
/=	c /= a is equivalent to c = c / a
%=	c %= a is equivalent to c = c % a

Assignment operators are highly useful for enumerations. In on of the code chunks in section 2.2.2 it was explained as how to use counter in loops. Setting counters using assignment operators does like a trick while executing programs for numerical calculations.

```
>>> cnt = 0
```

```
>>> for i in range(1, 11):
    if i % 2 == 0:
        print(f'{i} is even number')
        cnt += 1
else:
    print(f'there are {cnt} even numbers
          ↪ between 1 to 10')

2 is even number
4 is even number
6 is even number
8 is even number
10 is even number
there are 5 even numbers between 1 to 10
```

The above code executes a program for checking even numbers. As we know, there are 5 even numbers and 5 odd numbers between 1 to 10. The above program not only finds even numbers between 1 to 10 but also makes count for them. The assignment operator `+ =` works like a charm in the above code chunk.

## Logical Operators

Following are few important logical operators.

Operator	Description	Example
AND	If both operands are true then condition becomes true	(a & b)
OR	If any of the two operands are non-zero then condition becomes true	(a   b)
IS	Just works as “==”	(a is b)
NOT	Used to reverse the logical state of its operand.	not (a)
IS NOT	Just works as “!=”	(a is not b)

For instance:

```
>>> x = True; y = False
>>> print(x and y)
False
>>> print(x or y)
True
```

```
>>> print(not(x or y))
False
>>> print(x is y)
False
>>> print(x is not y)
True
```

There is only basic knowledge on operators in this book. All the above description is very minimal information which is only sufficient to understand data types. There is lot of documentation online for rest of the other operators.<sup>33</sup> It is advised to read about rest of the operators well before proceeding into forthcoming sections.

### 2.1.2 Tuples, Lists, Sets and Dictionaries

Tuples, lists and dictionaries needs more discussion. At times these are referred to data structures in stead of data types. These entities play very important role while performing statistical analysis.

#### Tuples

A tuple is a data structure that is an immutable, or unchangeable, ordered sequence of elements. Because tuples are immutable, their values cannot be modified. Tuples are typically used for storing collections of heterogeneous data. Tuples are used for grouping data. Tuples may be constructed in a number of ways:

1. Using a pair of parentheses to denote the empty tuple: ()
2. Using a trailing comma for a singleton tuple: a, or (a,)
3. Separating items with commas: a, b, c or (a, b, c)
4. Using the tuple() built-in: tuple() or tuple(iterable)

Each element or value that is inside of a tuple is called an item. Tuples have values between parentheses ( ) separated by commas (,). Empty tuples will appear as `tup = ()`, but tuples with even one value must use a comma as in `tup = ('january', 'february', .....`).

```
>>> tup
```

```
('january', 'february', 'march', 'april', 'may', '  
    ↪ june', 'july', 'august', 'september', '  
    ↪ october', 'november', 'december')
```

When thinking about Python tuples and other data structures that are types of collections, it is useful to consider all the different collections you have on your computer: your assortment of files, your song playlists, your browser bookmarks, your emails, the collection of videos you can access on a streaming service, and more.

Tuples are similar to lists, but their values can't be modified. Because of this, when you use tuples in your code, you are conveying to others that you don't intend for there to be changes to that sequence of values. Additionally, because the values do not change, your code can be optimized through the use of tuples in Python, as the code will be slightly faster for tuples than for lists.

### *Indexing Tuples*

As an ordered sequence of elements, each item in a tuple can be called individually, through indexing. Each item corresponds to an index number, which is an integer value, starting with the index number 0. For the `tup` tuple, the index breakdown looks like this:

0	1	2	3	4	5	.....	11
---	---	---	---	---	---	-------	----

```
>>> tup[0];tup[1]  
'january'  
'february'
```

In addition to positive index numbers, we can also access items from the tuple with a negative index number, by counting backwards from the end of the tuple, starting at -1. This is especially useful if we have a long tuple and we want to pinpoint an item towards the end of a tuple. For instance, suppose that I don't know what is the last element in the `tup`,

```
>>> tup[-1]  
'december'
```

Let me introduce a special module `string` which contains a number of useful constants and classes, as well as some deprecated legacy

functions that are also available as methods on strings. In addition, Python's built-in string classes support the sequence type methods viz. `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, `xrange`, and also the string-specific methods. We use template strings or the `%` operator to output formatted strings.<sup>34</sup>

```
>>> import string

>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'

>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Now we will use `string` module to implement over tuples.

```
>>> tup = (string.ascii_uppercase)
>>> tup
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> tup[1]
'B'
>>> tup[2]
'C'
>>> tup[0]+tup[1]
'AB'
```

We might be able to use certain basic built-in functions to manage tuples. Such as `len`, `max`, `min`.

```
>>> tup = (string.ascii_uppercase)

>>> tup
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> tup[1]
'B'
```

```
>>> tup[2]
'C'

>>> tup[0]+tup[1]
'AB'

>>> tup[0]*tup[1]

Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    tup[0]*tup[1]
TypeError: can't multiply sequence by non-int of type 'str'

>>> len(tup)
26

>>> min(tup)
'A'

>>> max(tup)
'Z'

>>> min(tup)-max(tup)

Traceback (most recent call last):
  File "<pyshell#47>", line 1, in <module>
    min(tup)-max(tup)
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

You may find error while performing subtraction and multiplication. This is due to the fact that the tuple has non-numerical elements in it. When we perform addition we are actually clubbing two characters into one. So we can perform addition but other arithmetic operations might not be possible over tuples unless the tuple has numeric values.

```
>>> tup1 = (range(0, 5))
>>> tup1
```

```
[0, 1, 2, 3, 4]
>>> tup1[1]*tup[2]
'C'
>>> tup1[1]*tup1[2]
2
>>> tup1[2]*tup1[3]
6
```

## Lists

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or *iterable*, or to create a subsequence of those elements that satisfy a certain condition.

```
>>> list(string.ascii_uppercase)[0:5]
['A', 'B', 'C', 'D', 'E']

>>> alfab = list(string.ascii_uppercase)

>>> type(alfab)
<type 'list'>

>>> alfab[0]; alfab[1]
'A'
'B'

>>> alfab[0:5]
['A', 'B', 'C', 'D', 'E']
```

Now let us implement our negative indexing over list `alfab`.

```
>>> alfab[-1]; alfab[-2]
'Z'
'Y'
```

There isn't 0 for left indexing for 0 is earmarked for very first element by Python. You can use `for` loop to populate all the elements of a

tuple. Please read about looping in the next section. [2.2.2](#) It is even possible for us to exercise certain functions over tuples. Functions like `len`, `max`, `min`.

List has some special methods to execute. To know more about them execute `dir(list_object)`. Some of the methods are '`append`', '`count`', '`extend`', '`index`', '`insert`', '`pop`', '`remove`', '`reverse`', '`sort`'. For instance

```
>>> squa = []
>>> squa
[]
>>> for x in lis[0]:
squa.append(x*x)

>>> squa
[0, 1, 4, 9, 16]

>>> help(squa.count)
Help on built-in function count:

count(...)
    L.count(value) -> integer -- return number of occurrences of value

>>> squa.count(2)
0
>>> squa.count(4)
1
>>> squa.count(9)
1
```

Suppose I don't know how to use the method `count`; I can know about this particular method by using `help` just as `help(squa.count)`. Python knows that your object i.e. `squa` is a list object. Now let us experiment on other method `.extend()`.

```
>>> help(lis.extend)
Help on built-in function extend:

extend(...)
```

```
L.extend(iterable) -- extend list by appending elements from the
```

So the method `.extend()` appends the object `c`. However, the extension will not behave in the fashion list accommodates the elements as `.append()`.

```
>>> c = range(10, 20)

>>> c
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

>>> lis.extend(c)

>>> lis
[[0, 1, 2, 3, 4], [6, 7, 8, 9], 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

>>> for i in lis:
    print i

[0, 1, 2, 3, 4]
[6, 7, 8, 9]
10
11
12
....
```

There is `for` loop which helps enlist all the elements of list object (`lis`). We have not discussed anything about *loops* till now, you will know more about `for` loop in forthcoming section. Earlier the object `lis` accommodated both `a`, `b` as lists. So the object `lis` has two other lists in side of it. But the method `.extend()` updates the `lis` so that the elements in the *iterable* i.e. `c` accommodated as a unique element having separate index. Each of the numbers 10 to 20 has not only unique but also a separate index number.

```
>>> lis[1]
[6, 7, 8, 9]
>>> lis[3]
11
>>> lis[-1]
```

```
19  
>>>
```

Now let us see the other methods.

```
>>> lis2 = range(1, 10)  
  
>>> lis2  
[1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
>>> max(lis2)  
9  
  
>>> min(lis2)  
1  
  
>>> lis2.insert(-1, e)  
  
>>> lis2  
[1, 2, 3, 4, 5, 6, 7, 8, 'BCDE', 9]  
  
>>> help(lis.sort)  
Help on built-in function sort:  
  
sort(...)  
    L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*  
    cmp(x, y) -> -1, 0, 1  
  
>>> lis2.sort  
<built-in method sort of list object at 0x7fa7394ca878>  
  
>>> lis2.sort()  
  
>>> lis2  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 'BCDE']  
>>>
```

`lis2` is another object of the type list. The second statement creates numbers 1 to 9 in `lis2`. The third statement *inserts* another

object BCDE, for which the method `.insert()` is used. The command `help()` always comes handy in knowing about objects in Python. The statement `lis2.sort` sort out the `lis2`. The `.sort()` method need not any arguments. However, it is possible to implement `.sort()` in reverse order by using argument `reverse=True`.

```
>>> lis2.sort(reverse=True)
>>> lis2
['BCDE', 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>>
```

Compare this with earlier result. In previous example the alphabets are in last position. However, when the list is reversed, the alphabets takes the first position. Someextent, we might be due to memory address of the object. For instance, use a specific and built-in function `id` over different objects. For time being we use this object over numbers and alphabets.

```
>>> help(id)
Help on built-in function id in module __builtin__:

id(...)
    id(object) -> integer

    Return the identity of an object. This is guaranteed to be unique
    simultaneously existing objects. (Hint: it's the object's memor...
```

```
>>> id(1)
30114200

>>> id(2)
30114176

>>> id(a)
140356197623064

>>> id(b)
140356197622632
```

```
>>> id(1) < id(a)
True
```

Intuitively it is clear that the memory address of numbers are lesser than that of alphabets. I don't guarantee the concept but it makes sense to me. There might be other reasons for this issue, you may google around for the reason.

## Sets

Python also includes a data type/structure for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets. However, to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary, a data structure that we discuss in the next section.<sup>35</sup> For instance:

```
>>> measures = {'mu', 'sd', 'var'}
>>> measures
{'sd', 'var', 'mu'}
>>> 'mu' in measures
True
>>> 'pvalue' in measures
False
```

We can also practice *list comprehensions* though.

```
>>> mct = measures
>>> md = {"var", "mean dev.", "sd", "qd"}
>>> dif = {x for x in mct if x not in md}
>>> dif
{'mu'}
>>> dif = {x for x in md if x not in mct}
>>> dif
{'mean dev.', 'qd'}
```

```
>>>
```

There is not any description on list comprehensions till now. There is some certain discussion in forthcoming section [2.4.2](#)

## Dictionaries

Another useful data type/structure in Python is the *dictionary*. Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as a set of *key:value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of *key:value* pairs within the braces adds initial *key:value* pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a *key:value* pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the `in` keyword. <sup>36</sup>

Here is a small example using a dictionary:

```
>>> mct = {'mean': 32, 'sd': 2.5, 'var': 22.5, 'UCL': 33, 'LCL': 12}
>>> mct
{'mean': 32, 'sd': 2.5, 'var': 22.5, 'UCL': 33, 'LCL': 12}
>>> mct.keys()
```

```
dict_keys(['mean', 'sd', 'var', 'UCL', 'LCL'])
>>> mct.values()
dict_values([32, 2.5, 22.5, 33, 12])
>>> mct['mean']
32
>>> list(mct)
['mean', 'sd', 'var', 'UCL', 'LCL']
>>> list(mct)[1]
'sd'
>>> list(mct)[0]
'mean'
>>> 'mean' in mct
True
>>> 'kurt' not in mct
True
>>>
```

Looping *dictionaries* is very interesting. Suppose we have few names of statistics in keys and numbers are in values, we might be able to use for loop to *format* properly to arrive at user defined formatted output.

```
>>> stat = dict(mean = 12, sd = 12.5)
>>> stat
{'mean': 12, 'sd': 12.5}

# loops

>>> for k, v in stat.items():
print(k, v)

mean 12
sd 12.5
>>> for k, v in stat.items():
print('{0} value is {1}'.format(k, v))

mean value is 12
sd value is 12.5
>>>
```

Dictionaries are very much useful while developing code for various statistical measures and tests. Suppose if there is a need to print output for various measures such as a statistic along with P Value, then dictionaries are best alternatives. We will see more about this in forthcoming section while writing code for various statistical techniques such as *t test*, *chi-square test*, *f test* and etc.

## 2.2 Control Flow in Python

In data science, what we need is maximum usage of control flow. To me control flow in simple words is all about using loops coupled with conditions. So it is highly advisable to spend more time on implementing logic using loops like `for` and conditions through `if` statements.

It is very important to understand *control flow* well before the concept of *data types* in Python. Whatever, we may do on data it one way or other way depends on control flow. We really don't have any other mechanisms to get the data which is in the form of characters or numbers into the console without using control flow. When I refer to control flow I essentially mean to mention loops together with conditions. In python we have very few statements to take care of loops such as `for`, `while`. And coming to conditional statements we just have `if` statement to take care of flow of logic.

### 2.2.1 if statement

Use `help('if')`, it will give *syntax* and a very little description to the function. I got following description in Python 3.6 executed in IDLE in Ubuntu 18.04.

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section Boolean operations for the definition of true and false); then that suite is executed (and no other part of the "if" statement is executed or evaluated). If all expressions are false, the suite of the "else" clause, if present, is executed.

Using conditional statements in Python is a fun. It is unlike other programming language. Python has very radical design patters. Using

control flow in Python is always unappealing to someone coming from C or Java like environments. In Python to use `if`, we just have to use the word `if` that all. Whatever, the statement (Truth/False) followed by this statement is evaluated. We don't need to use parenthesis just as in other languages. Suppose if  $a = 10$  and  $b = 20$  then the difference  $b - a$  could be positive.

```
>>> a = 10; b = 20
>>> if b > a:
    print("positive")

positive
```

In the above code `if` is Python's conditional statement. The expression `b>a` is *truth statement*. We don't need to do much ado like using starting parenthesis and closing parenthesis either before or after `if` statement. What happens if we use *false statement*? Python becomes silent. Because the response for alternative action is not given to Python.

### Nested `if` condition

In few situations logic might require few conditions which need to be checked together. One for truth and the other for false. Such mechanisms are known as *if else conditions*. Suppose, we want to evaluate the above example i.e.  $a,b$  not only for truth statement but also for false statement. Then the code might look as below:

```
>>> a = 10; b = 20
>>> if a > b:
    print('b is greater than a')
else:
    print('b is less than a & this is false statement')

b is less than a & this is false statement
>>> a > b
False
```

Python never knows what is  $a$  and  $b$  as soon as it encounters logic. The very first priority of Python is to help the programmer in implementing

his or her idea. A language is slave and good subordinate of the programmer. It is upto programmer to frame logic. This is how `if` `else` statement works in Python. Apart from `if`, `if-else` there is another context known as `nested if`.

```
>>> if a > b:  
    print(f'the difference is negative {a-b}')  
elif b > a:  
    print(f'the difference is positive {b-a}')  
else:  
print('I don\'t know')  
  
the difference is positive 10
```

In the above code there are two conditions, for both true and false conditions, and one default statement i.e. `else`. The *default* condition will be evaluated if both true or false are not possible. In the code `f` strings were used inside `print` statements. These statements are introduced in Python 3 thorough *PEP 498 – Literal String Interpolation*.<sup>37</sup> This culture has come from web development communities to accommodate variables within the code.

### 2.2.2 Loops

Using loops in Python is a fun, just like `if` statement. I repeat again, users from other platforms such as C and Java always find it weird while using loops in Python. Several times, I get unlikely questions and find learners flummoxed during teaching loops in Python. There might be several ways to implement loops in Python. I would like to bring two of the most sought after statements i.e. `for` and `while` to explain loops. It is possible to use these two statements in different ways. The following are few possibilities of using loops in Python.

1. `for` loop
  - (a) Iterating by index
  - (b) `for-else`
2. `while` loop

- (a) Single statement `while` block
- (b) `while-else`
- 3. Nested loops
- 4. Loop controls
  - (a) `pass`
  - (b) `break`
  - (c) `continue`

The last possibility i.e. *loop controls* is known as *control flow* in programming.

### **for loops**

Use `help('for')` in Python console to know more about *for* loops. *for* loops are used to traverse elements of an entity. The entity can be valid Python object, say a string, list, tuple etc. Suppose there is a number series of the values from 0 to 10, which we can create using our classical base function `range()`. Using `for` loop over such ranges can make a number series.

```
>>> a = range(10)
>>> for i in a:
print(i)
0
1
2
3
4
....
```

The above code chunk shows a very simple way of using `for` loop. The same activity can be done using `list()` method over above object i.e. `a`.

```
>>> a
range(0, 10)
>>> type(a)
```

```
<class 'range'>
>>> list(a)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We may not be able to write `a` and press return button on keyboard for elements of `a` in console, just as in other languages such as R. In case if you do so, Python let you know the type of the object, such as `range(0, 10)` as in the above code. Loops like `for`, `while` are essential to output values of any data variables in Python. The another example we can process strings as shown below.

```
>>> import string
>>> alfab = string.ascii_uppercase
>>> for let in alfab:
    print let

A
B
C
D
E
F
G
H
....
```

These type of data processing is very much essential in forthcoming chapters in this book. More specifically, `for` loops are very much important to make data using numerical simulations. Next Chapter we will be learning *pandas*, a special package which helps in dealing with data frames. If you are reading this book particularly this section, then certainly you must be knowing about data analysis. Data analyst do deal with two types of constructs while manipulating data, viz. *data matrix* and *data frame*. Data matrices are special data sets of the type *static array*; data frames are special data sets of the type *dynamic array*. Suppose if you have two lists one with responses such as *marital status* of respondents and the other *frequencies*, then we might be able to output these lists as data set using `for` loop just as below.

```
>>> a = ['married', 'unmarried', 'other']
>>> b = [22, 33, 44]
>>> for i in range(len(a)):
print(a[i], b[i])

married 22
unmarried 33
other 44
```

This seems rather not properly formatted; we will use a text formatting command `\t`, which helps us put a *tab* space between numbers and words.

```
>>> for i in range(len(a)):
print(b[i], '\t', a[i])
```

```
22    married
33    unmarried
44    other
```

We got a clean numeric vector of **a** and character vector **b**. It is very early to discussion about data analytics. There is all that discussion in Chapter 4.

### for-else loops

There is one important aspect of Python, it is also possible to combine `else` statement with for loop. `for` loops also have an `else` clause which most of us are unfamiliar with. The `else` clause executes *after the loop completes normally*. This means that the logic should be completely executed well before program encounters `else`. They are highly useful but needs to be understood well. Many don't know about this feature, and I happened to read about it later. This feature is highly useful while developing applications in stead of processing numerical data.

Let us assume that there are few *employees* who need to be checked for bonus based on job *performance*. I am using dummy names like '`emp1`', '`emp2`', etc. Assume that *performance* is measured using some

scale which range from 1 to 5, 1 stands for very poor and 5 stands for very high.

```
>>> employees = ['emp1', 'emp2', 'emp3', 'emp4', '
    ↪ emp5']
>>> performance = [1, 2, 4, 5, 4]
>>> cnt = 0

>>> for emp, p in zip(employees, performance):
    if p > 3:
        print(f'Select {emp} for bonus')
        cnt += 1
else:
    print(f'{cnt} of employees were selected
        ↪ for bonus')

Select emp3 for bonus
Select emp4 for bonus
Select emp5 for bonus
3 of employees were selected for bonus
```

Both *employees* and *performance* were created by using lists. The object *cnt* is a counter which helps us to count number employees who are eligible for bonus based on criteria *performance*  $> 3$ . From the result, it is clear that there are three employees eligible for bonus and they are '*emp3*', '*emp4*', '*emp5*'. The *else* statement in the above code chunk is used for *for* loop not for *if* statement.

### Single statement **while** loop

In python, **while** loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in program is executed. This means **while** has stamina to check conditions (*if*) for few iterations (*for*). In a very simple way **while** statement can be used as following:

```
>>> cnt = 0
>>> while cnt <= 5:
```

```
print cnt
cnt = cnt + 1

0
1
2
3
4
5
```

We must not forget to use the increment step after `print` statement, otherwise `while` statement loops forever. One important advantage of loops in Python is that we can use `else` parameter while using loops.

```
>>> while True:
    num = input("Enter a number: ")
    if num == str(5):
        print("\n")
        print("Good bye! you entered 5")
        break
    else:
        print("You entered: ", num)

Enter a number: 1
Enter a number: 2
Enter a number: 5

Good bye! you entered 5
```

The `else` statement in the above code is not `if` statement but for `while`. The surprising part is that the program continues to execute `else` part until user inputs 5. Whenever, 5 is pressed the `if` condition is executed and the program breaks with the help of `break` statement. Why I said surprising is that usually `else` is always used for exiting program just as the one explained for `if-else` in section [2.2.2](#).

### Nested loop

Python allows to use one loop inside another loop. Following code can illustrate the nested loop using `for` statement.

```
>>> import string
>>> for letter in string.ascii_uppercase[0:3]:
    for i in range(3):
        print(letter, i)

A 0
A 1
A 2
B 0
B 1
B 2
C 0
C 1
C 2
```

Outer `for` loop creates first three uppercase alphabets and the outer loop creates numbers 0 to 2. The inner loop for `range(3)` is executed for outer loop `string.ascii_uppercase[0:3]`. The program prints the vector 0, 1 and 2 for each of the alphabet. This type of code chunks are very important while simulating data sets in Python. You will know utility of such logic in forthcoming chapter in this book.

### Loop controls

While there is not much to discuss about loop controls, sometimes they do help regulating control flow in programs. The `pass` statement is useful many times to make dummy functions. This statement just pass the control without executing anything. In other words, `pass` is used simply to pass the control in the program.

```
>>> while True:
    x = input('Input value :')
    if x < str(3):
        pass
    else:
```

```
        print('The input value is >= 3')

Input value :1
Input value :2
Input value :3
The input value is >= 3
Input value :
```

The whole logic in the program is to *pass* for any value which is less than or equal to 3. The program executes logic only if it encounters the number 3. However, if you observe, the program continues regardless of the logic. This program continues until programmer issues **break** statement otherwise, the only way to stop this program is by executing *Cntl + c* shortcut buttons on the keyboard.

```
>>> while True:
        x = input('Input value :')
        if x < str(3):
            pass
        else:
            print('The input value is
                  ↪ >= 3')
            break

Input value :1
Input value :2
Input value :3
The input value is >= 3

>>>
```

This time the prompt (**>>>**) gets back at the end of the execution. Because, now this code has **break** statement. The program not only executes logic but also *breaks* control.

The **continue** statement is a bit interesting. This statement seems to behave like **pass** but different. Sometimes, I really think of saluting *Guido* for his prowess in designing Python. Observe the following code.

```
>>> program = ['bba', 'bse', 'bcom', 'mba', 'bba',
   ↪   'bcom', 'mba']
>>> for student in program:
    if student is 'bba' or student is 'mba':
        continue
    print(student, '-', 'found non management
   ↪   student')

bse - found non management student
bcom - found non management student
bcom - found non management student
```

I want to find a non-management students from a list of students. Their programmes were given (`program` list). The statement `continue` pass the control when it encounters only either `bba` or `mba`. The `print` statement is executed for rest of the words. Let's see what `pass` does.

```
>>> for student in program:
    if student is 'bba' or student is 'mba':
        pass
    print(student, '-', 'found non management
   ↪   student')

bba - found non management student
bse - found non management student
bcom - found non management student
mba - found non management student
bba - found non management student
bcom - found non management student
mba - found non management student
```

Oh! Something went wrong. Rather very grave mistake. This type of logic harms systems. The programme identifies, everyone as *non-management* student. Because, the `pass` statement just passes the control instead of executing logic.

You see what is happening? The statement `continue` is reverse execution of plain `if` statement.

```
>>> for student in program:
```

```

if student is 'bba' or student is 'mba':
    print(student, '-', 'found
        ↪ management student')

bba - found management student
mba - found management student
bba - found management student
mba - found management student

```

The statement `if` finds out all those students who belongs to either *bba* or *mba*, whereas the statement `continue` is useful for identifying students who does not belong to either of these programmes. So, as a programmer, one need to be very alert and must posses right understanding on concepts.

## 2.3 Functional Programming in Python

“Whereas some declarative programmers only pay lip service to equational reasoning, users of functional languages exploit them every time they run a compiler, whether they notice it or not. ”

- Philip Wadler, *How to declare an imperative*

Functional programming is a *programming paradigm* - a style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions or declarations instead of statements. Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms. Few of them are as follows: <sup>38</sup>

1. *Imperative* which allows side effects,

- *Object-oriented* which groups code together with the state the code modifies,
- *Procedural* which groups code into procedures,

2. *Declarative* which does not state the order in which operations execute,
  - **Functional** which disallows side effects,
  - *Logic* which has a particular style of execution model coupled to a particular style of syntax and grammar, and
3. *Symbolic* programming which has a particular style of syntax and grammar.

### 2.3.1 Modules

Functional programming is famously successful owing to its modular approach. In short, a *module* is a collection of functions, and a *package* is collection of modules. Modules are created as a matter of growing needs of programming. A *script* is collection of *routines*. A routine is composed of attributes and methods. An *attribute* can be a statement composed of text and other *literals*. While coming to *method*, it is a plain definition of an expression. An *expression* can be collection of literals.

Let me put it in simple words. Suppose, you write few functions in Python Console, just like the one IDLE, and if you quit from the Python interpreter and enter it again, the definitions (`def`) you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. *This is known as creating a script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; definitions from a module can be imported into other modules or into the main module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

Precisely, a module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`.

Creating a module is just making a text file but with `.py` file extension. Suppose, I want to create a script with a name `chapter_2`. In Linux, it can be done in Terminal just as below:

```
sudo gedit chapter_2.py
```

You can use `nano`, `vim` or any other text editor in stead of `gedit` in Linux. In windows, you may create a script using `notepad` like text editor.

### 2.3.2 Functions

In Python, function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. Furthermore, it avoids repetition and makes code reusable.<sup>39</sup>

Basically, we can divide functions into the following two types:

1. Built-in Functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

For instance, `range()` is in-built function whereas `myrange()` [2.3.2](#) is User-defined function.

#### Syntax of Function

Usually the following syntax is used to write functions in Python.

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

Description:

1. Keyword `def` marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.<sup>40</sup>
3. Parameters (arguments) through which we pass values to a function. They are optional. A colon (`:`) to mark the end of function header.
4. Optional documentation string (*docstring*) to describe what the function does.
5. One or more valid python statements that make up the function body. Statements must have same *indentation* level (usually 4 spaces).
6. An optional `return` statement to return a value from the function.

A simple example: since we are already familiar with loops and conditions. Let us develop a definition to output a linear series of numbers i.e. from 1 to some  $n$ . Just as our default `range()` function. Since, there is already a function by name `range` I shall try using `myrange`.

```
>>> def myrange(n):
    i = 0
    while i <= n:
        print(i)
        i += 1

>>> myrange(5)
0
1
2
3
4
5
```

So, what do you understand? The function range is defined using `while` loop. Whenever, we use this function the while loop prints numbers from zero to given parameter ( $n$ ). I often make little fun with Python by creating few UDF. There is no command for clearing

the editor (like IDLE). We can define a function to clear the console (in IDLE) just as below.

```
>>> def cls(n):
    print('\n'*n)
```

Now console gets cleared whenever this function, `cls()`, is executed, but based on the the value supplied to `n` which is one the essential parameters/arguments to this function. These type of functions can be referred to utility functions. Let us try another utility function such as checking whether given input is integer or not.

```
def checkNum(n):
    if type(n) == int:
        print("the val is int")
    else:
        print("the val is not int
              ↪ ")
```

This function uses `if-else` statements and evaluates if the given input is integer or anything else. Let us try the function.

```
>>> checkNum(2)
the val is int
>>> checkNum("a")
the val is not int
>>>
```

This way we can just create any number of functions using Python's `def` statement.

### 2.3.3 lambda, map, filter, and reduce

The `lambda` operator or `lambda` function is a way to create small anonymous functions, i.e. functions without a name. These functions are throw-away functions, i.e. they are just needed where they have been created. Lambda functions are mainly used in combination with the functions `filter()`, `map()` and `reduce()`. The `lambda` feature was added to Python due to the demand from Lisp programmers.<sup>41</sup>

The general syntax of a lambda function is quite simple:

```
lambda argument_list: expression
```

The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name. The following example of a lambda function returns the sum of its two arguments:

```
>>> f = lambda x, y: x + y
>>> f(1, 1)
2
>>> f=lambda x, y: x*y
>>> f(1, 2)
2
>>> f(3, 4)
12
>>> f([1, 2, 3, 4, 5])

Traceback (most recent call last):
  File "<pyshell#337>", line 1, in <module>
    f([1, 2, 3, 4, 5])
TypeError: <lambda>() takes exactly 2 arguments (1 given)
```

The lambda function fails to return required result. We need to use `reduce` together with `lambda` to achieve the result as shown in previous section.

```
>>> range(1, 5)
[1, 2, 3, 4]
>>> reduce(lambda x, y: x * y, range(1, 5))
24
```

This way we can find the product of the elements. There is no other way in Python to multiply consequent elements available from a vector. This type of operations are very much useful in forthcoming section i.e. statistical analysis.

Now let us know little bit about the other two function i.e. `map` and `filter`. The `map()` function applies a given function to each item of an iterable (list, tuple etc.) and returns a list of the results. We might

perform same multiplication using same function. Let's see what is going to happen.

```
>>> f = lambda x, y: x * y
>>> res = map(f, range(1, 5))
>>> print(res)
<map object at 0x7f924b586b38>
```

We don't get result. Suppose we try with `list`, `set` etc.

```
>>> list(res)
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    list(res)
TypeError: <lambda>() missing 1 required positional argument: 'y'
>>> set(res)
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    set(res)
TypeError: <lambda>() missing 1 required positional argument: 'y'
```

The error message clearly shows that the argument is missing. Do you remember there are two arguments in `lambda` function namely `x`, `y`. `map()` treats them two different arguments and expects two different inputs. Suppose if we try with two arguments we may be able to solve the problem.

```
>>> res = map(lambda x, y: x * y, range(1, 5), range(1, 5))
>>> res
<map object at 0x7f924b586ac8>
>>> print(set(res))
{16, 1, 4, 9}
>>>
```

We get result as by number. Not an aggregate. This proves that the purpose of `map()` is to get element wise manipulation. And this is equal to the below operation.

```
>>> for i in range(1, 5):
```

```
print(i**2)
1
4
9
16
```

Now coming to `filter()`; as the name suggests, filter creates a list of elements for which a function returns true. Suppose we would like to create a filter for a list elements that are negative, we might use `filter()`.

```
>>> res = filter(lambda x: x < 0, range(-5, 5))
>>> list(res)
[-5, -4, -3, -2, -1]
```

The other example can be

```
>>> res = filter(lambda x: x >= -3 and x <= 3, range(-5, 5))
>>> list(res)
[-3, -2, -1, 0, 1, 2, 3]
```

Though these functions looks to be a bit confusing, but they all have very nice applications whiling dealing with statistical applications. As it was mentioned earlier, `lambda()` is very elegant to make pretty calculations in jiffy. Coming to `reduce()` and `map()`, they have their own utility in creating aggregations or lists for such aggregations. `filter()` could be greatly helpful in truncating any data variable by certain condition.

### 2.3.4 Arguments in Functions

Passing arguments is one of the essential features of functional programming. At base level a function can be an *empty function*, which means a function can be with no arguments. While coming to arguments, they are used in three different ways in Python.

- Just by names (ex. `arg1, arg2, ...`) also known as *required arguments*
- Using `*args` syntax, these are known as *positional arguments*
- Using `*kwargs` syntax, these are known as *keyword arguments*

Required arguments are required as name implies. Functions will not work if these arguments are missing. Positional arguments are also known as *static* arguments by virtue of their nature. The third type i.e. keyword arguments which are represented by `**kwargs` syntax are more optional in nature. These arguments are defined by a unique keywords and highly dynamic. In other words, program can accept, *if and only if validated by logic*, any parameter provided by the programmer. *One very important rule is keyword arguments always follows positional arguments.* <sup>42</sup>

### Empty functions

As it was mentioned earlier, a functions can be empty. For instance, let us create a dummy function which does nothing using Python statement `pass` in our module `chapter_2` which we created in section 2.3.1.

```
def passFun():
    ''' just pass the function '''
    pass
```

The above function `passFun` just does nothing but passing the control. Let us execute this function by importing the same from `chapter_2` which is a module.

```
>>> from chapter_2 import passFun
>>> help(passFun)
Help on function passFun in module chapter_2:

passFun()
    just pass the function

>>> passFun()
>>>
```

The function `help()` retrieves the Python's *docstring*, and when it was executed it just does nothing. The control returns to Python prompt `>>>`.

## Required arguments

In section 2.2.2 we learned how to create a data frame with two columns possibly one a numeric vector and the other character vector also known as factor. Since we have a vector `a` which is composing of numbers so it is possible for us to do any numerical computations. For instance, if we choose to define arithmetic mean as  $\frac{\text{sum}}{\text{number}}$  we might create a function as below:

```
>>> def myMean(a):
    return sum(a)/len(a)

>>> myMean(50)
24
```

The variable `a` in `myMean()` is known as *required argument* and here it is *compulsory argument*. Suppose I would like to define a function with a couple of summary statistics in which I would like to have few other measures like maximum and minimum. May be I can try like this.

## Positional arguments

Positional arguments are any arguments which user could use inside a function without having associated with names. In other words, positional arguments does not required to be defined with names. However, positional arguments are used inside the function syntactically with the help of `*args` statement. This statement must be used inside the parentheses of function. Look at the following function.

```
def argExample(*args):
    ''' example for arguments '''

    for arg in args:
        print(arg)
```

And its usage is

```
>>> argExample('hi')
hi
```

```
>>> argExample('hello')
hello
```

The function accepts any value as arguments. Now let us look into more meaningful example. Suppose we would like to create a function which return either *maximum* or *minimum* along with *mean* for any given vector of values, then we may be able to develop function as the one below:

```
def meanMaxMin(vec, *args):
    ''' This function creates Maximum and
        ↪ Minimum along with Mean as by user
        ↪ arguments.

    Arguments:
        vec: Input vector
        *args: positional arguments. eg. 'min'
            ↪ and 'max'

    Output:
        Returns default value: mean and either
            ↪ minumum or maximum by argument.

    ...
    out1 = myMean(vec)
    if 'max' in args:
        out2 = max(vec)
        return({'mean': out1, 'max':out2})
    if 'min' in args:
        out2 = min(vec)
        return({'mean': out1, 'min':out2})
```

The above function is just an extension to `myMean`. The following is the construction of the function.

1. The statement enclosed in `'''` creates what is known as `docstring`, which is useful to create little description, *usage information*, for

`help()` statement.

2. Elements such as `out1`, `out2` are dummy objects used to store the results.
3. Outputs are returned using *dictionary*.

While coming to the logic; the function `meanMaxMin` computes *arithmetic mean* by default and returns either *mean*, *minimum* or *mean, maximum* as by the *arguments* through python special syntax `*args`.

```
>>> from chapter_2 import meanMaxMin
>>> help(meanMaxMin)
Help on function meanMaxMin in module chapter_2:

meanMaxMin(vec, *args)
    This function creates Maximum and Minimum
    ↪ along with Mean as by user arguments.

Arguments:
vec: Input vector
*args: positional arguments. eg. 'min' and '
    ↪ max'

Output:
Returns default value: mean and either minumum
    ↪ or maximum by argument.

>>> meanMaxMin(range(1, 11), 'min')
{'mean': 5.5, 'min': 1}

>>> meanMaxMin(range(1, 11), 'max')
{'mean': 5.5, 'max': 10}
```

*Docstring* can be retrieved by executing `meanMaxMin.__doc__`. This statement just returns entire usage information of this function but as plain statement. The resultant output has information as in dictionary. First key is information for mean, and the second key is information for either minimum or maximum.

## Keyword arguments

Keyword arguments are defined by certain names associated with them. These arguments always follow positional arguments. For instance, we may be able to define keywords in any function just as below:

```
def kwargsExample(**kwargs):
    ''' example for keyword arguments '''

    for kw in kwargs:
        print(kw, ':', kwargs[kw])
```

And following is the implementation.

```
>>> kwargsExample(first='hi', second='hello')
first : hi
second : hello
```

We can also define arguments combined with keyword arguments just as below:

```
def argsAndKwargsExample(*args, **kwargs):
    ''' example for both arguments and keyword
        ↪ arguments '''
    for arg in args:
        print(arg)

    for kw in kwargs:
        print(kw, ':', kwargs[kw])
```

The implementation of the above function is as follows:

```
>>> argsAndKwargsExample('hi', 'hello', first='hi', second='hello')
hi
hello
first : hi
second : hello
```

Let us try more meaningful example. Suppose if I want to create a

function which outputs input vector but a mean/average by its type such as 'arithmetic mean (am)', 'geometric mean (gm)' and 'harmonic mean (hm)' through keyword arguments.

```
def meanByType(vec, **kwargs):
    ''' example for means '''
    for kw in kwargs:
        if kwargs[kw] == 'am':
            print(kw, ':', myMean(vec))
        elif kwargs[kw] == 'gm':
            print(kw, ':', geoMean(vec))
        elif kwargs[kw] == 'hm':
            print(kw, ':', harMean(vec))
    else:
        return(list(vec))
```

The above function has both required as well as keyword arguments. Methods such as *myMean()*, *geoMean()* and *harMean()* are user defined methods for *arithmetic mean*, *geometric mean* and *harmonic mean*. They are all defined inside Python script *chapter\_2* with appropriate methods. Refer to Python scripts associated with this book at <https://github.com/Kamakshaiah/PfSA>.

The implementation is as follows:

```
>>> meanByType(range(1, 5))
[1, 2, 3, 4]
>>> meanByType(range(1, 5), mean='am')
mean : 2.5
[1, 2, 3, 4]
>>> meanByType(range(1, 5), mean='gm')
mean : 2.213363839400643
[1, 2, 3, 4]
>>> meanByType(range(1, 5), mean='hm')
mean : 1.9200000000000004
[1, 2, 3, 4]
```

The function returns empty input vector (data) just as it is in the absence of keyword arguments. When there is keyword argument such

as *am*, *gm* or *hm*, the function returns valid mean for that name.

## 2.4 Using loops for matrix operations

The object “matrix” is very important in data analytics. One of the data sets can be a data matrix. In mathematics, a matrix is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. In data science, every data set is a type of data matrix, because every data set has rows and columns. After all, processing data is all about set up raw data into rows and columns in a meaningful way. This is what is known as *data manipulation and organization*.

There is nothing called matrix in Python. A matrix is a set of vectors doesn’t matter whether it is set in rows or columns. Each vector is independent and serves as a row. So, there will be only row wise operations but not column wise.

### 2.4.1 NumPy for matrices

NumPy or `numpy` is the fundamental package for scientific computing with Python. It contains among other things:

1. A powerful N-dimensional array object
2. Sophisticated (broadcasting) functions
3. Tools for integrating C/C++ and Fortran code
4. Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases. Let us see as how to create arrays and matrices using NumPy. We will be seeing as how to do the same using list comprehensions and nested loops in the following sections. NumPy has very efficient methods to create arrays.

```
>>> import numpy as np  
>>> x = np.array([2,3,1,0])
```

```
>>> x
array([2, 3, 1, 0])
>>> type(x)
<class 'numpy.ndarray'>
```

The fist statement, in the above code, imports NumPy into user session. The method `array` is useful to convert any vector of numerical data into an array and when created it is known as '`numpy.ndarray`'. You can know that using `type()` command in the console. These arrays are highly powerful in Python. You will notice their significance Chapter 4 while performing statistical analysis on various data sets. Now let us see as how to create few matrices such as zero, identity and ones using NumPy.

```
>>> x = np.zeros((4, 4))
>>> x
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> type(x)
<class 'numpy.ndarray'>
>>> x = np.identity(4)
>>> x
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
>>> type(x)
<class 'numpy.ndarray'>
>>> x = np.ones((4, 5))
>>> x
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
>>> type(x)
<class 'numpy.ndarray'>
```

As I mentioned earlier. Let us see as how to create these matrices using list comprehensions and nested loops. The aim of this section is to explain efficacy of list comprehensions over nested loops. So, let us make few functions as they mimic as above NumPy functions. I will be using same module i.e. *chapter\_2.py* for this activity. To create 2 dimensional (2D) arrays also known as matrices, we may have to use nested loops. Nested loops also known as *nested list comprehensions* in Python community. The below operations creates a  $1 \times n$  matrix in simple words it is a vector. We already had some certain discussion on manipulating data vectors in previous section.

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The above exercise creates a one dimensional vector which is of the order  $1 \times n$  matrix. We can achieve the same operation using *lambda* operator.

```
>>> squares = list(map(lambda x: x**2, range(10)))
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can further simplify just using list comprehension.

```
>>> squares = [x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

## 2.4.2 List Comprehensions

Guido prefers constructs in stead of list comprehensions like map, filter, reduce and lambda. List comprehensions were added with Python 2.0. Essentially, it is Python's way of implementing a well-known notation for sets as used by mathematicians. In mathematics the square numbers of the natural numbers are for example created by  $\{x^2 | x \in N\}$  or the set of complex integers  $\{(x, y) | x \in Z \wedge y \in Z\}$ . List comprehension is an elegant way to define and create list in Python.

These lists have often the qualities of sets, but are not in all cases sets. List comprehension is a complete substitute for the lambda function as well as the other functions like `map()`, `filter()` and `reduce()`. For most people the syntax of list comprehension is easier to be grasped.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it.

43

You may read these mechanisms from the section [2.1.2](#). Now let us see as how to create 2 dimensional data sets. For instance:

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

You need to remember your secondary level school education on *sets and functions*. We are trying to map the elements of one set i.e.  $(1, 2, 3)$  with other set  $(3, 1, 4)$ , with a condition  $x \neq y$ . We can also achieve the same outcome by using nested loop.

```
>>> oc = []
>>> for x in [1, 2, 3]:
    for y in [3, 1, 4]:
        if x != y:
            oc.append((x, y))
>>> oc
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1),
 ↪ (3, 4)]
```

Do you see now! So list comprehensions really save our day. As I have said before, *a matrix is a set of vectors which are two-dimensional in nature*.

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

```
[]
>>> mat
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

It doesn't matter the shape of the input vectors, Python always ejects the output as horizontally arranged fashion. We can use the same list comprehensions to create a *zeros matrix*.<sup>44</sup>

### Zeros matrix

Zeros matrix has lot of significance in statistical analysis. They are used rampantly in regression models. Zeros matrix with ones in diagonals is known as *unit matrix*. Now we are going to see how to create zeros matrix using both list comprehension and nested loops. Following is the code for creating zeros matrix using list comprehension.

```
>>> a = [[0 for i in range(5)] for i in range(5)]
>>> a
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0,
    ↪ 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
>>> for i in a:
    print i

[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
>>>
```

We might need `for` loop to print contents of matrix in rows and columns format. *We need nested loops to create same zeros matrix.*

```
>>> a = []
>>> for x in range(5):
    row = []
    for y in range(5):
        row.append(0)
    a.append(row)
```

```
>>> a
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0,
    ↪ 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
>>> for i in a:
    print i

[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
```

## Square Matrices

Square matrix is a matrix whose number of rows are equal to number of columns. Suppose we want to produce a matrix with 3 X 3 order.

45

```
>>> mat = [[0] * 3] * 3
>>> [x for x in mat]
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> mat = [[0] * 3] * 3
>>> a = [x for x in mat]
>>> for i in a:
    print i

[0, 0, 0]
[0, 0, 0]
[0, 0, 0]
```

For that matter we can fill matrices with any elements using values as we wish. There is another attractive way of making matrices in Python using *iterator*.

```
>>> it = iter(range(9))
>>> [[next(it) for i in range(3)] for i in range
    ↪ (3)]
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

```
>>> it = iter(range(16))
>>> [[next(it) for i in range(4)] for i in range
    ↪ (4)]
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12,
    ↪ 13, 14, 15]]
```

The code above fills all numbers starting from 0 to the maximum limit as by the object `it` which is an instance of `iter`. The strategy is very simple. We just used *list comprehensions* to create *zero matrix*. Same fashion you may use `iter()` to create a matrix of linear sequence of numbers such as  $1, 2, 3, \dots, n$ . Try yourself.

### Rectangular Matrix

A matrix for which horizontal and vertical dimensions are not the same (i.e., an  $m \times n$  matrix with  $m \neq n$ ). The logic is very simple, the number in both ranges i.e. range for `i` and range for `j` should be unequal.

```
>>> zeros = [[0 for i in range(3)] for j in range
    ↪ (5)]
>>> zeros
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0,
    ↪ 0, 0]]
>>> for i in zeros:
    print i

[0, 0, 0]
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]
```

Till now we handled the concept of matrices using loops and conditional statements. Let us write or create a function which creates a *zeros matrix* since we are aware of functions in Python.<sup>46</sup>

```
>>> def zerosMatrix(m, n):
    ''' creates zeros matrix '''
```

```
mat = [[0]*n for i in range(m)]
for r in mat:
    print(r)

>>> zero_matrix(3, 3)
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]

>>> zero_matrix(5, 3)
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]

>>> zero_matrix(2, 5)
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
```

## Identity Matrix

The identity matrix, or sometimes ambiguously called a *unit matrix*, of size  $n$  is the  $nn$  square matrix with ones on the main diagonal and zeros elsewhere. It is denoted by  $I_n$ , or simply by  $I$  if the size is immaterial or can be trivially determined by the context.

```
>>> n = 5
>>> m = [[int(x==y) for x in range(n)] for y in
        range(n)]
>>> m
[[1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0,
    ↪ 0], [0, 0, 0, 1, 0], [0, 0, 0, 0, 1]]
>>> for i in m:
    print i

[1, 0, 0, 0, 0]
[0, 1, 0, 0, 0]
```

```
[0, 0, 1, 0, 0]
[0, 0, 0, 1, 0]
[0, 0, 0, 0, 1]
```

Identity matrices has lot of applications in multivariate analytics. For instance, singularity or non-singularity of any given matrix can be determined by its invertible. If a matrix is invertible then it is non-singular. If a matrix is non-singular it might be multivariate uniform distribution.<sup>47</sup> The other important application in statistics is that if A is invertible (nonsingular, non-degenerate) then its columns are *linearly independent*. Such dependencies describes the very properties of *dimensions* which has lot of significance in the theory of statistics and mathematics.<sup>48</sup>

We can define a function for identity matrix as below:

```
def identityMatrix(n):
    ''' creates identiriy matrix '''
    n = n
    m = [[int(x==y) for x in range(n)]for y in
          range(n)]
    for i in m:
        print(i)
```

The implementation is as follows:

```
>>> identityMatrix(5)
[1, 0, 0, 0, 0]
[0, 1, 0, 0, 0]
[0, 0, 1, 0, 0]
[0, 0, 0, 1, 0]
[0, 0, 0, 0, 1]
```

## Ones Matrix

A matrix of ones or all-ones matrix is a matrix over the real numbers where every element is equal to one.

$$M = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

We might be able to create a function for ones' matrix just as below:

```
def onesMatrix(n):
    ''' creates ones matrix '''
    rows = [[1] * n for i in range(n)]
    for r in rows:
        print(r)
```

The implementation is as follows:

```
>>> onesMatrix(5)
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]
```

Why are we creating UDF when we are able to get these methods through NumPy? Simple, if you want learn, learn it in hardest way. There is no simple way to learn. Learning requires *core thinking*. Going to the base and doing things from scratch is essential requirement for core thinking. Programming is intellectual activity. Intelligence arises from churning the brain. It is very essential for any programmer to know about *nitty-gritty* of programming. *If you are able to write your own programs using fundamental knowledge of a language, you are not a user but a developer.* That is the only way to grow robust in computing. Many times I write my own functions using methods in stead of using modules or packages. Why? Because that is the only way to ensure “what I do is what I know”!

### Matrices with random numbers

We can try filling elements in matrices using `random` module, in case if you are bored with zeros and ones. We may use random numbers such as below:

```
>>> [random.randint(1, 10) for i in range(5) for j
     ↪   in range(5)]
[3, 6, 1, 10, 2, 2, 9, 4, 4, 2, 3, 4, 2, 4, 10,
     ↪ 10, 8, 1, 2, 3, 7, 2, 5, 4, 10]
>>> elements = [[random.randint(1, 9) for i in
     ↪   range(5)] for j in range(5)]
>>> for i in elements:
     print(i)

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

In the above code; `random.randint(1, 10)` creates a random data point for rows using `range(5)` and columns using `range(5)`. These rows and columns were printed using list comprehension. I will try to create a function a name `randMatrix()` inside module `pythonprogramming.py` so that it can be used wherever we need to create a matrix with random numbers.

```
def randMatrix(n, a, b):
    ''' Creates a X b order matrix with random
        ↪ data '''
    import random as rnd
    mat = [[rnd.randint(1, n) for i in range(a)]
           ↪ for j in range(b)]
    return(mat)
```

This function works for three arguments namely  $n$ ,  $a$ ,  $b$  where  $n$  is the range for random integer which is used by the module `random` inside the function,  $a$  and  $b$  required to define the order of the matrix.

```
>>> from pythonprogramming import randMatrix
>>> mat = randMatrix(10, 4, 5)
>>> mat
[[4, 2, 10, 8], [7, 7, 9, 8], [6, 8, 2, 1], [9, 9,
   ↪ 10, 2], [8, 1, 10, 1]]
>>> for i in mat:
    print(i)

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

### Columnar data processing in matrices

Columns represents presentation of data vertically. There is a reason why this happens in data analytics. One important reason is columns mostly represents *attributes* while rows represents *instances*. In social science, a data set is any data frame with certain rows and columns. Each row is an instance of an entity eg. person, firm, etc. and each column is attribute of that entity eg. age, salary, education etc. While list comprehensions makes it easy creating matrices, but indexing makes possible for us to separate rows and columns from a given matrix. Let us try the matrix having some elements in rows and columns.

```
>>> a = [[random.randint(1, 9) for x in range(5)]
   ↪ for y in range(6)]
>>> a
[[3, 8, 3, 7, 3], [8, 8, 8, 1, 4], [1, 3, 1, 2,
   ↪ 2], [4, 7, 2, 1, 7], [8, 4, 7, 4, 7], [6, 6,
   ↪ 3, 5, 9]]
>>> for i in a:
    print(i)

[3, 8, 3, 7, 3]
```

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

We can separate rows using loops, as we have been doing from long. However, how to process them? is the real question. Because, in forthcoming chapters we will be dealing data processing. To process data either we might need rows or columns but not on whole data set. So to process rows we can use a particular concept call list indexing. For instance, I might be able to do as below, to sum all the elements of very first row of above matrix i.e.  $a$ ,

```
>>> for i in range(len(a[0])):
    print(a[0][i])

3
8
3
7
3
>>> x = []
>>> for i in range(len(a[0])):
    x.append(a[0][i])

>>> sum(x)
24
```

You see? the sum of  $3, 8, 3, 7, 3$  is 24. However, this is not how data processing happens in social sciences. Given a data set such as sample individuals as rows and their attributes as columns, it is required to process columns but not rows. Because, to process data we need similar data instances, but not heterogeneous. Row-wise data is always heterogeneous in nature. Suppose if the very first column of above matrix i.e.  $a$  is some relevant attribute as age. The mean of age is relevant for analysis. Now the point is as how to process columnar data.

```
>>> for i in range(len(a)):
            print(a[i][0])
3
8
1
4
8
6
```

Now we got values of very first column of matrix *a*. The trick is very simple we just have to change index. We have to change iterator *i* to first position instead of second position. Now it may be possible to process columnar data using any damn statistical measure.

```
>>> b = []
>>> for i in range(len(a)):
            b.append(a[i][0])

>>> print(b)
[3, 8, 1, 4, 8, 6]
>>> sum(b)/len(b)
5.0
```

The values  $[3, 8, 1, 4, 8, 6]$  belongs to very first column of matrix *a*. The expression `sum(b)/len(b)` calculates arithmetic average. This is how data processing is done in statistical analysis. In deed, we don't need to struggle as this, we will be processing data using a particular package called `pandas`. However, it is always worth knowing and doing columnar data processing from scratch.

### 2.4.3 Matrix Manipulation

In Python a matrix means a 2D array. Matrix manipulation is any operations such as add, subtract, multiply and divide done on two or more matrices. Python is one of the efficient programming languages that deals with matrix arithmetic or manipulation. In this section, we will see as how to perform very basic yet highly important operations on matrices. It is easy to perform matrix arithmetic using *numpy*. I

created a module named *Pythonprogramming.py* to collect few functions that were created in this Chapter. I will be using this module to develop source code related to matrix operations alongside using *numpy* for evaluation.

## Addition

Matrix addition is adding two matrices using nested loop and nested list comprehension. I will try to create two random matrices and then add them using nested loops to make addition.

```
>>> mat1 = randMatrix(10, 4, 4)
>>> mat2 = randMatrix(10, 4, 4)
>>> mat1
[[5, 8, 3, 9], [6, 4, 2, 4], [10, 3, 9, 7], [8,
    ↪ 10, 6, 6]]
>>> mat2
[[7, 1, 3, 1], [7, 3, 8, 9], [3, 1, 2, 2], [10, 5,
    ↪ 2, 3]]]
```

Now let me use nested for loops to add each element of `mat1` with that of `mat2`.

```
>>> import Pythonprogramming as pp
>>> result = pp.zerosMatrix(4, 4)
>>> for i in range(len(mat1)):
        for j in range(len(mat1[0])):
            result[i][j] = mat1[i][j] +
                ↪ mat2[i][j]
>>> result
[[18, 9, 9, 17], [18, 3, 14, 10], [19, 7, 13, 13],
    ↪ [17, 13, 6, 12]]]
```

What are `range(len(mat1))` and `range(len(mat1[0]))`? They are just linear number series starting from 0 to 3. Indirectly we are just adding for every element (*i*) of `mat1` with that (*j*) of `mat2` using `range()` statement. At last the additions of the elements, at positions *i*, *j*, were parsed into the object with a name `result`. Now let us define a function for matrix multiplication.

```

def matrixAddition(m1, m2):
    ''' Multiplies input matrices m1 and m2 '''

    m = len(m1)
    n = len(m2[0])
    result = zerosMatrix(m, n)

    result = zerosMatrix(m, m)
    for i in range(len(m1)):
        for j in range(len(m1[0])):
            result[i][j] = m1[i][j] + m2[i][j]
    return(result)

```

Objects `m`, `n` represents rows and columns for dummy matrix `result` which serves as a container for output from addition of elements. The rest of the code is all about just converting logic for addition into a Python definition using inner loops. Let us test this function.

```

>>> matrixAddition(randMatrix(10, 4, 4),
    ↪ randMatrix(10, 4, 4))
[[6, 19, 11, 13], [5, 8, 11, 10], [11, 14, 12,
    ↪ 17], [11, 16, 9, 13]]

```

## Subtraction

Matrix subtraction is pretty much like addition. We don't really need to understand subtraction technically. If we substitute minus in stead of plus in the above code it can perform matrix subtraction. The function name for matrix subtraction is going to be `matrixSubtraction()` and it is available in the module `Pythonprogramming.py` in side *PfDSaA* package.

```

def matrixSubtraction(m1, m2):
    ''' Multiplies input matrices m1 and m2 '''

    m = len(m1)
    n = len(m2[0])
    result = zerosMatrix(m, n)

```

```
for i in range(len(m1)):
    for j in range(len(m1[0])):
        result[i][j] = m1[i][j] - m2[i][j]
return(result)
```

As usual I created a dummy matrix with all zero elements, using UDF `zerosMatrix()` in the present module i.e. `Pythonprogramming.py`. I used inner loops to manipulate subtraction on two input matrices shown as `m1` and `m2`. Let us test the code.

```
>>> from Pythonprogramming import randMatrix,
       ↪ matrixSubtraction
>>> m1 = randMatrix(10, 4, 4)
>>> m2 = randMatrix(10, 4, 4)

>>> m1
[[10, 4, 7, 3], [1, 3, 3, 5], [1, 2, 2, 2], [5, 1,
   ↪ 6, 9]]
>>> m2
[[1, 4, 3, 5], [10, 3, 9, 6], [7, 5, 2, 2], [9, 1,
   ↪ 3, 7]]

>>> matrixSubtraction(m1, m2)
[[9, 0, 4, -2], [-9, 0, -6, -1], [-6, -3, 0, 0],
   ↪ [-4, 0, 3, 2]]
```

As usual, I used `randMatrix()` function, from same module i.e. `Pythonprogramming.py`, to simulate two matrices with random data elements. The resultant matrices were used to perform subtraction. The result is obvious from the code.

## Multiplication

Matrix multiplication is rather a typical topic. It has many things involved in it. There are different types of matrix multiplication namely, (1) pairwise product, (2) dot product, (2) cross product (3) inner product, (4) outer product and many more. I may not be able to explain

all these types. However, I will be able to show you as how to make two very important matrix multiplications they are *pairwise product* and *matrix product*. The pairwise product is also known as *Hadamard product* or *Schur Product* only named after their inventors. Hadamard product is used in few computer applications such as advanced computer visions and Machine Learning (ML). The other type i.e. matrix product, which is close to dot product, is used in all most all operations of multivariate statistics. We need the later mostly while performing few analyses such as multivariate correlation, multiple regression, and few other multivariate techniques in data analysis. Doing pairwise product is as simple as am arithmetic calculation in Python. The name of the function for pairwise product is `pairwiseProduct()` in `Pythonprogramming.py` module.

```
def pairwiseProduct(m1, m2):
    ''' Performs pairwise multiplications of
        ↪ elements from input matrices m1 and m2
        ↪ '''

    m = len(m1)
    n = len(m2[0])
    result = zerosMatrix(m, n)

    for i in range(len(m1)):
        for j in range(len(m2[0])):
            result[i][j] = m1[i][j] * m2[i][j]
    return(result)
```

As usual I used a dummy matrix using `onesMatrix()` function and inner loops for processing multiplication of each element of  $m1$  with corresponding element of  $m2$  in same position. The result is parsed to dummy matrix with a name `result`. Doing *matrix product* is rather typical for we need three level inner loop using `for` statement. One loop for iterating row-wise elements of  $m1$  matrix, another loop for iterating column-wise elements of  $m2$  matrix and final loop to make element wise multiplication. I will name the function as `matrixMultiplication()`.

```
def matrixMultiplication(m1, m2):
    ''' Performs matrix multiplication of
        ↪ elements from input matrices m1 and m2
        ↪ '''

    m = len(m1)
    n = len(m2[0])
    result = zerosMatrix(m, n)

    for i in range(len(m1)):
        for j in range(len(m2[0])):
            for k in range(len(m2)):
                result[i][j] += m1[i][k] * m2[k
                    ↪ ][j]

    return(result)
```

As it was mentioned, matrix product is not as easy as pairwise product. There is little logic in it. The logic is as this: suppose if we have  $ik$  and  $kj$  matrices then the matrix product produces a result of the order  $ik$ .

```
>>> m1 = randMatrix(10, 4, 5)
>>> m2 = randMatrix(10, 5, 4)

>>> matrixMultiplication(m1, m2)
[[157, 174, 145, 132, 93], [171, 206, 175, 180,
    ↪ 123], [130, 159, 153, 155, 95], [125, 152,
    ↪ 133, 158, 123], [186, 204, 188, 176, 122]]

>>> m1
[[8, 2, 1, 10], [8, 2, 7, 8], [8, 5, 6, 2], [1, 2,
    ↪ 9, 8], [9, 6, 2, 10]]
>>> m2
[[8, 10, 8, 6, 2], [4, 3, 7, 7, 5], [5, 8, 7, 10,
    ↪ 7], [8, 8, 6, 6]]
```

You may manually check the multiplication of first row of  $m1$  with

first column of  $m2$ . The result is 157.

## Division

Matrix division is not too different from addition, subtraction, multiplication. We just have to substitute division symbol ( $/$ ) in place of  $+$ ,  $-$ ,  $*$ . Apart from the operational concerns, matrix divisions are highly important while performing few key analyses in statistics. For instance, *matrix determinants can be used to solve linear systems using Cramer's rule, where the division of the determinants of two related square matrices equates to the value of each of the system's variables*. Division is one of the four arithmetic operations used in studying various properties of matrices.<sup>49</sup> Coming back to coding; the code for matrix division can be as follows.

```
def matrixDivision(m1, m2):
    ''' Divides input matrix m1 with m2 and
        ↪ provides resultant matrix '''

    m = len(m1)
    n = len(m2[0])
    result = zerosMatrix(m, n)

    for i in range(len(m1)):
        for j in range(len(m1[0])):
            result[i][j] = m1[i][j] / m2[i][j]

    return(result)
```

Testing the code.

```
>>> m1 = randMatrix(10, 4, 3)
>>> m2 = randMatrix(10, 4, 3)
>>> matrixDivision(m1, m2)
[[3.0, 1.125, 1.0, 0.2], [1.0, 1.75, 0.75,
    ↪ 0.2222222222222222], [0.1666666666666666,
    ↪ 2.6666666666666665, 1.0, 0.5]]
>>> import numpy as np
>>> np.divide(m1, m2)
```

```
array([[3.           , 1.125       , 1.           , 0.2
       ↪      ],
       [1.           , 1.75       , 0.75       ,
        ↪ 0.22222222],
       [0.16666667, 2.66666667, 1.           , 0.5
        ↪      ]])
```

Objects `m1` and `m2` are two input matrices simulated using `randMatrix()` from `Pythonprogramming.py` module. The function `matrixDivision()` divides matrix `m1` by `m2` and returns output as in matrix. Outputs for both `matrixDivision()` and `np.divide()` are same.

## Transpose

One very important matrix operation which is not in arithmetic operations is *matrix transpose*. The transpose of a matrix is an operator which flips a matrix over its diagonal, that is it switches the row and column indices of the matrix by producing another matrix. In simple words, matrix transpose is an act of converting rows into columns. It is an element wise operation. Transpose as an operation is highly important for data processing. Many statistical techniques, especially bivariate and multivariate statistics, depends on matrix transpose. Matrix transpose, as function, is highly useful to compute few measures related to Correlation, Regression, Principal Components Analysis (PCA), Factor Analysis and also in Cluster Analysis. We will try to discuss these methods in detail in Chapter 4.

For time being, let me explain as how to write code or develop function for matrix transpose. The strategy for transposing matrix is very simple. As it was said, matrix transpose is the act of converting rows into columns. This means all the rows of parent matrix will be arranged into columns in resultant or output matrix. The function `matrixTranspose()` in `Pythonprogramming.py` module is useful to perform matrix transpose.

```
def matrixTranspose(m):
    ''' Transpose matrix m '''
    a = len(m)
```

```
b = len(m[0])
result = zerosMatrix(b, a)

for j in range(a):
    for i in range(b):
        result[i][j] = m[j][i]

return(result)
```

In the above listing, `m` is input matrix and `a`, `b` are rows and columns i.e. matrix `m` has  $a \times b$  order. The object `result` is a dummy matrix with zero elements and whose order is  $b \times a$ . The logic for transposing is implemented using inner loops. The outer loop is iterated by `j` but not by `i`. This is because, we need to alter the order while parsing elements from input matrix `m` to output matrix `result`.

#### 2.4.4 Python packages for Matrices

*Numpy* package has got all the above shown methods. You may perform all other additional methods such as *outer*, *inner* what not? everything.

```
>>> m1 = randMatrix(10, 4, 4)
>>> m2 = randMatrix(10, 4, 4)
>>> import numpy as np
>>> np.matmul(m1, m2)
array([[ 53, 107, 124, 132],
       [ 40,  89, 164, 195],
       [ 73, 144, 215, 231],
       [ 32,  48, 117, 120]])
>>> np.dot(m1, m2)
array([[ 53, 107, 124, 132],
       [ 40,  89, 164, 195],
       [ 73, 144, 215, 231],
       [ 32,  48, 117, 120]])
>>> np.cross(m1, m2)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
```

```
...
...
raise ValueError(msg)
ValueError: incompatible dimensions for cross
    ↪ product
(dimension must be 2 or 3)
```

The function `matmul()` from `numpy` helps in performing matrix multiplication and it is also a dot product. You can see the results for both matrix multiplication and dot product are same. I don't know why cross product fails for  $4 \times 4$  order matrix. But it works for any matrix with less than or equal to  $3 \times 3$  order matrices.

```
>>> m1 = sml.simNumVec(3)
>>> m2 = sml.simNumVec(3)

>>> np.cross(m1, m2)
array([ 2050, -1501, -175])

>>> m1 = randMatrix(10, 3, 3)
>>> m2 = randMatrix(10, 3, 3)

>>> np.cross(m1, m2)
array([[ -19,   16,    9],
       [-33,   16,   26],
       [ 36,   45,  -58]])
```

The function `simNumVec()` appears in next Chapter ???. This function just creates a vector of random numbers. The function `randMatrix()` is available from module associated with this Chapter with a name `Pythonprogramming.py`. `numpy.cross()` works only for 2 or 3 dimensional vectors or matrices. There are also functions available from `scipy` package. `csr_matrix()` from Scipy helps us to create *Compressed Sparse Row* (CSR) matrix. Let us deal with `csr_matrix()` from Scipy to create a matrix with any random numbers.<sup>50</sup> One of the syntaxes used for creating or manipulating CSR matrix is `csr_matrix((data, (row_ind, col_ind)), [shape=(M, N)])`.

```
>>> import numpy as np
```

```
>>> from scipy.sparse import csr_matrix
>>> csr_matrix((3, 4)).toarray()
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> csr_matrix((3, 4), dtype = np.int8).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

The option `dtype` convert the float values to integer type. But the function, `zerosMatrix()`, we have written at Section 2.4.2 is far simpler than the way we did using *numpy* and *scipy*. Above all it is just a matter of confidence, which means it is always better to define own functions rather than depending on dependencies. The reason being, I can totally believe in what I've written than that of someone. Any way let us explore few other methods.

Suppose we want to make a matrix in which we think of certain values, it is not strait as we did earlier using random numbers from *random* package. We need to have a strategy. Look at the following figure 2.1.

From the figure it makes clear to you that about relative position of each of the elements in the matrix. The matrix has 9 elements namely  $a, b, c, \dots, i$ . The position of the first element i.e. **a** is  $[0, 0]$ , the position of the second element i.e. **b** is  $[0, 1]$  and so on and so forth. We need to put this plan into action to create a matrix with values of your interest. Look at the following code.

```
>>> row = np.array([0, 0, 1, 2, 2, 2])
>>> col = np.array([0, 2, 2, 0, 1, 2])
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> csr_matrix((data, (row, col)), shape = (3, 3))
    ↪ .toarray()
array([[1, 0, 2],
       [0, 0, 3],
       [4, 5, 6]])
```

User objects `row` and `col` takes care of relative positions of values. `data` takes care of values of interest. The plan is such that we want

values of our interest i.e.  $1, 2, 3, \dots, 6$  at  $[0, 0], [0, 2], [1, 2], [2, 0], [2, 1], [2, 2]$  respectively. The resultant array is the matrix of our interest.

### Determinant and Inverse of matrix

The package *numpy* is useful for quite a few of other matrix operations. I would like to explain two of the very important operations which required quite a bit of effort to develop code manually are (1) determinant, (2) inverse. Let us understand theory first and later operations.

Determinant is one of the operations used in linear algebra. The determinant is a scalar value that can be computed from the elements of a square matrix and encodes certain properties of the linear transformation described by the matrix. The determinant of a matrix A is denoted  $\det(A)$ ,  $\det A$ , or  $\text{mod } A$ . Geometrically, it can be viewed as the volume scaling factor of the linear transformation described by the matrix. In the case of a  $2 \times 2$  matrix the determinant may be defined as

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc. \quad (2.1)$$

Similarly, for a  $3 \times 3$  matrix A, its determinant is

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} = a \quad (2.2)$$

While it is easy to write a function to calculate determinant for  $2 \times 2$  matrix, but not easy to write Python code for any matrix with an order  $\geq 3 \times 3$ . I will try to discuss these methods possibly in my forthcoming book on advanced numerical analyses using Python. For time being,

matrix determinant can be calculated using `linalg.det()` function in `numpy` package. The function `randMatrix()` is used to simulated a  $4 \times 4$  order matrix. Please be cautious, the input matrix must be a square matrix.

```
>>> m = randMatrix(10, 4, 4)
>>> np.linalg.det(m)
185.9999999999974
```

As far as Matrix Inverse is concerned; The inverse of a square matrix  $A$  is a second matrix such that  $AA^{-1} = A^{-1}A = I$ , where  $I$  being the identity matrix. There are many ways to compute the inverse of a matrix, the most common being multiplying the reciprocal of the determinant of  $A$  by its *adjoint* also known as *adjugate*, which is the transpose of the *cofactor* matrix. ***A matrix is invertable if and only if the determinant is not equal to zero.***

In linear algebra, an  $n \times n$  square matrix  $A$  is called invertible (also non-singular or non-degenerate) if there exists an  $n \times n$  square matrix  $B$  such that

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I}_n \quad (2.3)$$

where  $I_n$  denotes the  $n \times n$  identity matrix and the multiplication used is ordinary matrix multiplication. If this is the case, then the matrix  $B$  is uniquely determined by  $A$  and is called the inverse of  $A$ , denoted by  $A^1$ . Matrix inversion is the process of finding the matrix  $B$  that satisfies the prior equation for a given invertible matrix  $A$ . ***A square matrix that is not invertible is called singular or degenerate. A square matrix is singular if and only if its determinant is zero. Singular matrices are rare in the sense that the probability that a square matrix whose real or complex entries are randomly selected from any finite region in the number line or complex plane is singular is 0, that is, it will “almost never” be singular.***

Performing matrix inverse using Python belongs to data science. However, using same operation on data sets is the responsibility of data analyst. Now in which way this is important in data analytics? or

How to use matrix inversion in statistical analyses? There is a particular yet highly popular statistical procedure called *Structural Equation Modeling (SEM)*, used in multivariate analysis. SEM is widely used to test theories and models. As such, SEM is not a technique but collection of techniques. One of the fundamental requirements to perform SEM is to ensure that the data set is non-singular. This means the data set used for SEM as a data matrix should have a determinant other than zero. If the determinant of input data set or matrix is zero then the relationships across the variables in that data may not be sufficient to test theory or model in hand. This error arises whenever, there is random data in hand. If the data inside a data set is distributed perfectly at random then it may not be possible to perform SEM on such data sets. Many software tools such as R, Matlab, Julia etc. rise error while performing SEM whenever, it is used on perfectly randomly distributed data sets.

I will show as how to perform matrix inverse using *numpy* package, though it is possible to write Python code for the same. *numpy* has a method `linalg.inv()` to perform matrix inversion. Given an input matrix  $m$ , the matrix inversion can be done as shown below.

```
>>> m = randMatrix(10, 4, 4)
>>> m
[[9, 6, 9, 3], [6, 4, 5, 5], [7, 5, 1, 8], [10, 9,
   ↪ 2, 3]]
>>> np.linalg.inv(m)
array([[ 1.27956989, -2.35483871,  1.16129032,
   ↪ -0.4516129 ],
       [-1.27419355,  2.29032258, -1.17741935,
   ↪  0.59677419],
       [-0.22043011,  0.64516129, -0.33870968,
   ↪  0.0483871 ],
       [-0.29569892,  0.5483871 , -0.11290323,
   ↪  0.01612903]])
```

## 2.5 Creating Packages

Packaging is beyond the scope of this book. However, let us have a very quick understanding on creating packages in Python, and that surely gives learner an added advantage. Creating packages is advanced knowledge. Usually you may feel to develop package when you have lots of code written due to not being able to trace them elsewhere. Coming to my case, I write most of the code for I don't like to depend on packages. These dependencies has quite a bit of problem, firstly, you don't know methods used by package author, albeit of the fact that mostly they are reliable. Secondly, it is always safe to have your own code for that relieves you from doubts and that is the open source way. Open source people always prefer to have their own food and they love to share it with others. It is okay to eat once in a while outside, but eating everyday outside is not a healthy idea. The three primordial characteristics of open source people are Self reliance, self dependency and self confidence. For instance, observe few open source evangelist such Richard M. Stallman, Linuz Torvald, Larry Wall, Guido Van Rossum and a couple of others; you may find these three characteristics as backdrop of their character.

Everyone starts journey as beginner. However, after a while everyone achieves mastery based on their journey and effort. You may find newer ideas, thoughts growing in your brain after sometime. For me, I have quite a few functions written for my usage and I do this whenever I find one not available from the community. I do write few functions in the area of statistical diagnosis whenever I felt that it is necessity to do so. Examples are aplenty. So, one day, you may find it necessary to write your own code in order to fill requirements for your job in the hand. Packages comes into the picture exactly at this juncture.

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name *main-mod.submod* designates a submodule named *submod* in a package named *mainmod*. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages from having to worry about each other's module names.

We don't write packages but make them. There are ways to make packages. First, you can create a package and use locally in your computer or at best you may distribute thorough some communication medium just like a mail, github or at worst make it available somewhere at your blog. Second, you can make your package available to public at least to Python community using *PyPI*, which stands for Python's Package Index. Visit <https://pypi.org/> for details. This needs registration at PyPI and you need to obtain username and password to upload your packages. There is a very detailed tutorial at <https://packaging.python.org/tutorials/packaging-projects/> as how to package Python project.

Since, creating packages is beyond the scope of this book, I can only explain the fist approach, i.e. *creating packages locally*. Making or creating a package this way is pretty simple idea in Python. All you need to do is to create a folder or directory and put all your code (.py files) in it. You just have to add a file to *initialize* your package. Initializing is all about telling Python to treat your project directory as a Python. Initialization of package requires to create a file called `__init__.py` in your project directory. That's all! Your project is ready as package now.

I will try to explain with the help of an example. Suppose we have few functions and they are related to descriptive statistics. Usually, descriptive statistics is all about summary statistics together with data visualizations. There are three types of measures in summary statistics. They are

1. Measures of central tendency: mean, median, mode
2. Measures of dispersion: variance, standard deviation
3. Measures of shape: skewness, kurtosis

I am going to write all my functions in four different .py files namely, (1) *mct.py*, (2) *md.py*, (3) *ms.py* and (4) *visuals.py*. The very first script i.e. *mct.py* is as follows:

```
def Mean(x):  
    ''' Calculates arithmetic mean '''  
    return sum(x)/float(len(x))
```

```
def Median(x):
    ''' Calculates median '''
    n = len(x)
    x.sort()

    if n % 2 == 0:
        median1 = x[n//2]
        median2 = x[n//2 - 1]
        return((median1 + median2)/2)
    else:
        median = x[n//2]
        return("Median is: " + str(median))

def Mode(x):
    ''' Calculates mode '''
    from collections import Counter

    n = len(x)

    data = Counter(x)
    get_mode = dict(data)
    mode = [k for k, v in get_mode.items() if v
            ↪ == max(list(data.values()))]

    if len(mode) == n:
        get_mode = "No mode found"
        return(get_mode)
    else:
        get_mode = "Mode is / are: " + ', '.join
            ↪ (map(str, mode))
    return(get_mode)
```

There are three functions in this script namely (1) *Mean*: to calculate arithmetic mean, (2) *Median*: to calculate median, and (3) *Mode*: to calculate mode for any given input data variable. All these three calculates univariate measures of central tendency. The second file

with name *md.py* has two very important functions which represents measures of dispersion and the code inside this file is as follows:

```
def Var(x):
    ''' Calculates variance '''

    ss = sum([(i-myMean(x))**2 for i in x])
    return float(ss)/(len(x) - 1)

def Std(x):
    ''' Calculates standard deviation '''

    return myVar(x)**0.5
```

The two of the functions related to measures of dispersion are (1) *Var*: calculates variance, and (2) *Std*: calculates standard deviation for any univariate data. The third script file *ms.py* is as follows:

```
def Skewness(x):
    numer = sum([(i-myMean(x))**3) for i in x])
    denom = sum([(i-myMean(x))**2)**float(3)/2
                 for i in x])
    return numer/denom

def Kurtosis(x):
    numer = sum([(i-myMean(x))**4 for i in x])
    numer = numer / len(x)
    denom = sum([(i-myMean(x))**2) for i in x])
    denom = (denom/len(x))**2
    return (numer/denom)-3
```

This file has two of the very essential functions which represents measures of shape and they are (1) *Skewness*: to calculate skewness and (2) *Kurtosis*: to calculate kurtosis for univariate data. The last file is for visualizations and the name of the script file is *visuals.py*. The code inside this script is as follows:

```
def plotNormal(x):
    ''' this method plots normal curve for given
```

```
    ↢  data'''
```

```
# importing packs
```

```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
```

```
# data transformation
y = norm.pdf(x, 0, 1)
plt.plot(x, y)
plt.show()
```

```
def twoByTwoPlot():
    ''' creates a visual for 2 by 2 graph grid
        requires no inputs
    '''
    import numpy as np
    import matplotlib.pyplot as plt
```

```
    for i in range(1, 5):
        plt.subplot(2, 2, i)
        x = np.random.uniform(1, 2, 100)
        y = np.random.uniform(1, 2, 100)
        plt.plot(x, y, 'o')
    plt.show()
```

```
def PDFplot():
    ''' creates a visual for PDF for normally
        ↢ distributed data
        requires no inputs
    '''

    import numpy as np
    import scipy.stats as stats
    import matplotlib.pyplot as plt
```

```
x = np.random.normal(0, 1, 100)
res = stats.probplot(x, plot=plt)
plt.show()

def CDFPlot():

    ''' creates a visual for CDF for normally
        ↪ distributed data
        requires no inputs
    '''

    import numpy as np
    import scipy.stats as stats
    import matplotlib.pyplot as plt

    y = np.random.normal(0, 1, 100)
    y.sort()
    y1 = stats.norm.cdf(y, np.mean(y), np.std(y)
        ↪ )
    x = np.linspace(-3, 3, 100)
    x1 = np.linspace(min(y1), max(y1), len(y))

    plt.scatter(x, y1, color = 'blue')
    plt.plot(x, x1, color = "red")
    plt.show()
```

This script has five functions namely (1) *plotNormal*: makes a plot for normal distribution, requires input data variable (2) *twoByTwoPlot*: makes a grid for two by two plots, does not require input data, (3) *PDFPlot*: makes a plot for Probability Distribution Function (PDF) and does not require input data, (4) *CDFPlot*: makes a plot for Cumulative Distribution Function (CDF) and does not require input data.

We reached almost to the end of the activity i.e. creating a directory and placing all these files inside of it. I am going to place these three file into a single directory with a name *summarystatistics*. Now my directory structure is as follows:

```
mk@mkg-hp:.../summarystatistics$ tree
.
  mct.py
  md.py
  ms.py
  visuals.py

0 directories, 4 files
```

The function `tree` in Linux is useful to print directory structure inside the Terminal. In Linux, you can install this utility function using your preferred installer. I use Ubuntu so I installed this utility function using `sudo apt install tree`.

You see I got three files under the directory `summarystatistics`. One last step is to create a file with a name `__init__.py` in the same directory. Actually we don't even need to do this step. Python automatically recognizes this directory as Python package, if you just open Python Console inside this directory and import any module (script) or function. Let us import any of these modules, using any of your preferred editor, and see what changes happens in this directory.

```
>>> import os
>>> os.getcwd()
'/home/mk'
>>> os.chdir('.../python_scripts')
>>> os.getcwd()
'.../python_scripts'
>>> os.listdir()
['scripts', 'scripts.zip', 'scripts_old', '
    ↪ summarystatistics', 'test.py']
```

We are just above the project directory `summarystatistics`. Let us import one of the modules from this folder say `mct.py` using IDLE.

```
>>> from summarystatistics import mct
```

Now go back to the Terminal and execute the `tree` command to observe the changes.

```
mk@mk-hp:.../summarystatistics$ tree
.
  mct.py
  md.py
  ms.py
  __pycache__
    mct.cpython-36.pyc
  visuals.py
```

Now there are changes in the directory. A new folder with a name `__pycache__` is created. Inside which there is a file (`mct.cpython-36.pyc`). This file is related to the module (`mct`) which we imported in Python console (I used IDLE). Now let us import rest of the scripts i.e. `md.py` and `ms.py` and see what changes happens in the project directory.

```
>>> from summarystatistics import md
>>> from summarystatistics import ms
```

Now go back to Linux Terminal and execute the command `tree` to see the additional changes.

```
mk@mk-hp:.../summarystatistics\$ tree
.
  mct.py
  md.py
  ms.py
  __pycache__
    mct.cpython-36.pyc
    md.cpython-36.pyc
    ms.cpython-36.pyc
  visuals.py
```

Python creates *cache* files for each module with an extension `.pyc` whenever the module is imported.

### 2.5.1 What is `__pycache__`?

Every programming language has a design strategy. As we know a *package* is a collection of modules and a *module* is collection of code

*blocks and a block is collection of statements.* The script, which is a collection of code blocks is a Python file, with .py extension, is usually known as *source* or *source code*. Every programming language compiles these sources only to get them ready for execution. This happens automatically as by default activity. Coming to Python design, the folder `__pycache__` is Python's default folder in which there is byte code which is created based on source code. These byte code files are files that are used by programming language to communicate to OS in order to accomplish user requests.

*PEP 3147* by name *PYC Repository Directories* narrates the importance of `__pycache__`. Please read the same at <https://www.python.org/dev/peps/pep-3147/#id26>. The description at this site is as follows:

This PEP describes an extension to Python's import mechanism which improves sharing of Python source code files among multiple installed different versions of the Python interpreter. It does this by allowing more than one byte compilation file (.pyc files) to be co-located with the Python source file (.py file). The extension described here can also be used to support different Python compilation caches, such as JIT output that may be produced by an Unladen Swallow enabled C Python.

### 2.5.2 What is `__init__.py`?

The `__init__.py` only required to inform Python that the directory is, in fact, a package not just a directory. Python might treat the directory to manage what is called *namespace* conflicts. The following piece of information available at certain Stack Over Flow describes about this file:<sup>51 52</sup>

Python defines two types of packages, regular packages and *namespace* packages. Regular packages are traditional packages as they existed in Python 3.2 and earlier. A regular package is typically implemented as a directory containing an `__init__.py` file. When a regular package is imported, this `__init__.py` file is implicitly executed,

and the objects it defines are bound to names in the package's namespace. The `__init__.py` file can contain the same Python code that any other module can contain, and Python will add some additional attributes to the module when it is imported.

This shows that the major idea behind `__init__.py` is only to recognize the directory as package and this is required to manage namespace conflicts. *The other utility of `__init__.py` file is of course to manage imports.* It is possible to mention as which modules needs to be imported while dealing with package. By this way Python earmarks huge amount of importance to user. This freedom allows user to define any name rather freely regardless of namespace conflicts. This shows that user can define the very *behavior* of package while importing modules. A user can set a property known as `__all__` inside `__init__.py` so as to define a list of modules that need to be imported during initialization.

Right now our package `summarystatistics` may not be able to import any module either using `import *` or `import summarystatistics`. The directory structure is reinstated to original condition.

```
mk@mk-hp:.../summarystatistics$ tree
.
mct.py
md.py
ms.py
visuals.py
```

0 directories, 4 files

Now try to import the whole directory as package.

```
>>> from summarystatistics import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__
    ↪ __loader__', '__name__', '__package__', '__
    ↪ __spec__', 'os']

>>> import summarystatistics
```

```
>>> summarystatistics.mct.myMean(range(1, 11))
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    summarystatistics.mct.myMean(range(1, 11))
AttributeError: module 'summarystatistics' has no
  ↪ attribute 'mct'

>>> from summarystatistics import mct
>>> mct.myMean(range(1, 11))
5.5
```

Neither the directory nor the \* is considered for importing. We may not be able to import the whole directory as package. Because, Python never knows that this directory is a package. The code works when we import a particular module say *mct.py* using *from summarystatistics import mct*. Let us see the directory structure.

```
mk@mk-hp:.../summarystatistics$ tree
.
  mct.py
  md.py
  ms.py
  __pycache__
    mct.cpython-36.pyc
  visuals.py

1 directory, 5 files
```

Let me delete \_\_pycache\_\_ and create \_\_init\_\_.py so as to enable Python recognize project directory (*summarystatistics*) as package. Now the directory structure is as follows:

```
mk@mk-hp:.../summarystatistics$ tree
.
  __init__.py
  mct.py
  md.py
```

```
ms.py  
visuals.py  
  
0 directories, 5 files
```

Try importing the whole directory as package.

```
>>> os.listdir()  
['scripts', 'scripts.zip', 'scripts_old', '  
 ↪ summarystatistics', 'test.py']  
>>> dir()  
['__annotations__', '__builtins__', '__doc__', '  
 ↪ __loader__', '__name__', '__package__', '  
 ↪ __spec__', 'os']  
  
>>> import summarystatistics  
>>> dir()  
['__annotations__', '__builtins__', '__doc__', '  
 ↪ __loader__', '__name__', '__package__', '  
 ↪ __spec__', 'os', 'summarystatistics']  
  
>>> from summarystatistics import *  
>>> dir()  
['__annotations__', '__builtins__', '__doc__', '  
 ↪ __loader__', '__name__', '__package__', '  
 ↪ __spec__', 'os', 'summarystatistics']
```

The statement `import summarystatistics` now successfully imports `summarystatistics` as a valid package after creating `__init__.py` file in the directory. This means Python now started recognizing the whole directory as a package. However, the statement `from summarystatistics import *` still not working as we are not able to import modules using `*`. To get `*` working let us do little work. Try to define which files need to be imported while initializing package by adding `__all__ = ['mct', 'visuals']`. This means I want Python to load only `'mct.py', 'visuals.py'` whenever we use `*`.

```
>>> dir()
```

```
[ '__annotations__', '__builtins__', '__doc__', '
    ↪ __loader__, '__name__', '__package__', '
    ↪ __spec__', 'os']
>>> from summarystatistics import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '
    ↪ __loader__, '__name__', '__package__', '
    ↪ __spec__', 'mct', 'os', 'visuals']
```

Now Python not only recognize the directory as package but loads modules as by the attribute `__all__` inside `__init__.py` file. Now verify the directory structure.

```
mk@mk-hp:.../summarystatistics\$ tree
.
├── __init__.py
├── mct.py
└── md.py
└── ms.py
└── __pycache__
    ├── __init__.cpython-36.pyc
    ├── mct.cpython-36.pyc
    └── visuals.cpython-36.pyc
└── visuals.py
```

It has all those byte-code compilations for both levels i.e. package (`__init__.cpython-36.pyc`) level and module (`mct.cpython-36.pyc` `visuals.cpython-36.pyc`) level. The other packages can be imported individually or we can define them inside `__init__.py` file using `__all__`.

```
>>> from summarystatistics import md, ms
>>> dir()
['__annotations__', '__builtins__', '__doc__', '
    ↪ __loader__, '__name__', '__package__', '
    ↪ __spec__', 'mct', 'md', 'ms', 'os', 'visuals'
    ↪ ']
```

## 2.6 Exception handling

Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong). When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash. For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A. If never handled, an error message is displayed and our program comes to a sudden unexpected halt.

### 2.6.1 Catching Exceptions in Python

In Python, exceptions can be handled using a `try` statement. The critical operation which can raise an exception is placed inside the `try` clause. The code that handles the exceptions is written in the `except` clause. We can thus choose what operations to perform once we have caught the exception. Here is a simple example.

```
import sys
try :
    "a"/2
except :
    print(sys.exc_info())
```

The above script produces below output.

```
(<class 'TypeError'>, TypeError("unsupported operand type(s)
                                ← for /: 'str' and 'int'",),
     ← traceback object at
     ← 0x03943B68>)
```

Above line is not a single line but a tuple with items separated by commas. We can try below code to get each of the element inside this tuple.

```
import sys
try :
    "a"/2
except :
```

```
for i in sys . exc_info () :
    print (i)
```

The output is as follows:

```
<class 'TypeError'>
unsupported operand type(s) for /: 'str' and 'int'
<traceback object at 0x03403B88>
```

Below can be much better example.

```
# import module sys to get the type of exception
import sys
randomList = ['a', 0 , 2]
for entry in randomList :
    try :
        print (" The entry is", entry )
        r = 1/int ( entry )
        break
    except :
        print (" Oops !", sys . exc_info ()[0], " occurred .")
```

The output

```
The entry is a
Oops ! <class 'ValueError'> occurred .
The entry is 0
Oops ! <class 'ZeroDivisionError'> occurred .
The entry is 2
```

## 2.6.2 Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of *errors*: *syntax errors* and *exceptions*.

### Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print '(Hello world )'  
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the function `print()`, since a colon (`:`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

## Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> while True print '(Hello world )'  
SyntaxError: invalid syntax

>>> 10 * (1/0)  
Traceback (most recent call last):  
  File "<pyshell#1>", line 1, in <module>  
    10 * (1/0)  
ZeroDivisionError: division by zero

>>> 4 + spam *3  
Traceback (most recent call last):  
  File "<pyshell#2>", line 1, in <module>  
    4 + spam *3  
NameError: name 'spam' is not defined

>>> '2' + 2  
Traceback (most recent call last):  
  File "<pyshell#4>", line 1, in <module>  
    '2' + 2  
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions

come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords). The rest of the line provides detail based on the type of exception and what caused it. The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

### 2.6.3 Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
    try:
        x = int ( input ("Please enter a number: "))
        break
    except ValueError :
        print ("Oops! That was no valid number. Try, again ...")

Please enter a number: 1

>>> while True:
    try:
        x = int ( input ("Please enter a number: "))
        break
    except ValueError :
        print ("Oops! That was no valid number. Try, again ...")

Please enter a number: a
Oops! That was no valid number. Try, again ...
Please enter a number:
```

The try statement works as follows.

1. The try clause (the statement(s) between the try and except keywords) is executed.
2. If no exception occurs, the except clause is skipped and execution of the try statement is finished.
3. If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
4. If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except ( RuntimeError , TypeError , NameError ):  
...     pass
```

Example

```
try:  
    f = open('myfile.txt')  
    s = f.readline()  
    i = int(s.strip())  
    print(s)  
except OSError as err:  
    print('OS error: {}'.format(err))  
except ValueError :  
    print('Could not convert data to an integer.')  
except:  
    print ('Unexpected error :', sys.exc_info())  
    raise
```

Above program returns OS error: [Errno 2] No such file or directory: 'myfile.txt' if there is no file and returns data. Suppose if there text in the file then the the result will be Could not convert data to an integer. The `try ... except` statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.

For example:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that was not raised by the code being protected by the `try ... except` statement. When an exception occurs, it may have an associated value, also known as the exception's argument. The presence and type of the argument depend on the exception type. The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`. One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
try:
    raise Exception('spam', 'eggs')
except Exception as inst:
    print(type(inst))      # the exception instance
    print(inst.args)       # arguments stored in .args
    print(inst)            # __str__ allows args to be printed
                           # directly,
                           # but may be overridden in exception
                           # subclasses
                           #
x, y = inst.args        # unpack args
```

```
print('x =', x)
print('y =', y)
```

The above code produces below output.

```
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
```

If an exception has arguments, they are printed as the last part ('detail') of the message for unhandled exceptions. Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```
def this_fails():
    x = 1/0

try:
    this_fails()
except ZeroDivisionError as err:
    print('Handling run-time error:', err)
```

The above code produces `Handling run-time error: division by zero` as output.

#### 2.6.4 Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `Exception`). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')
    raise
```

### 2.6.5 Exception Chaining

The raise statement allows an optional from which enables chaining exceptions. For example:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
def func():
    raise IOError

try:
    func()
except IOError as exc:
    raise RuntimeError('Failed to open database') from exc
```

Ouput

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
OSError

The above exception was the direct cause of the following
  ↪ exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

Exception chaining happens automatically when an exception is raised inside an `except` or `finally` section. Exception chaining can be disabled by using from `None` idiom:

```
try:
    open('database.sqlite')
except IOError:
    raise RuntimeError from None
```

Ouput

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

### 2.6.6 User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see Classes for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly. Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error
                      occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
```

```

        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition
       ↪ that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition
                   ↪ is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message

```

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions. Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter Classes.

### 2.6.7 Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```

try:
    raise KeyboardInterrupt
finally:
    print('Goodbye, world!')

```

Ouput

```

Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>

```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs

whether or not the try statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the try clause, the exception may be handled by an except clause. If the exception is not handled by an except clause, the exception is re-raised after the finally clause has been executed.
- An exception could occur during execution of an except or else clause. Again, the exception is re-raised after the finally clause has been executed.
- If the try statement reaches a break, continue or return statement, the finally clause will execute just prior to the break, continue or return statement's execution.
- If a finally clause includes a return statement, the returned value will be the one from the finally clause's return statement, not the value from the try clause's return statement.

For example:

```
def bool_return():
    try:
        return True
    finally:
        return False
```

Output

```
>>> bool_return()
False
```

A more complicated example:

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

### Ouput

```
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

### 2.6.8 Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
    print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

## 2.7 Regexpressions

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the `re` module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as “Does this string match the pattern?”, or “Is there a match for the pattern anywhere in this string?”. You can also use REs to modify a string or to split it apart in various ways.

Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn’t covered in this document, because it requires that you have a good understanding of the matching engine’s internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

We’ll start by learning about the simplest possible regular expressions. Since regular expressions are used to operate on strings, we’ll begin with the most common task: matching characters. For a detailed explanation of the computer science underlying regular expressions (deterministic and non-deterministic finite automata), you can refer to almost any textbook on writing compilers.

### 2.7.1 Matching Characters

Most letters and characters will simply match themselves. For example, the regular expression test will match the string test exactly. (You can enable a case-insensitive mode that would let this RE match Test or TEST as well; more about this later.) There are exceptions to this rule; some characters are special metacharacters, and don't match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the RE by repeating them or changing their meaning. Much of this document is devoted to discussing various metacharacters and what they do. Here's a complete list of the metacharacters; their meanings will be discussed in the rest of this HOWTO.

```
. ^ \$ * + ? { } [ ] \ | ( )
```

The first metacharacters we'll look at are [ and ]. They're used for specifying a character class, which is a set of characters that you wish to match. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a '-'. For example, [abc] will match any of the characters a, b, or c; this is the same as [a-c], which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your RE would be [a-z].

Metacharacters are not active inside classes. For example, [akm\\$] will match any of the characters 'a', 'k', 'm', or '\$'; '\$' is usually a metacharacter, but inside a character class it's stripped of its special nature. You can match the characters not listed within the class by complementing the set. This is indicated by including a '^' as the first character of the class. For example, [^5] will match any character except '5'. If the caret appears elsewhere in a character class, it does not have special meaning. For example: [5^] will match either a '5' or a '^'.

Perhaps the most important metacharacter is the backslash, \. As in Python string literals, the backslash can be followed by various characters to signal various special sequences. It's also used to escape all the metacharacters so you can still match them in patterns; for

example, if you need to match a [ or \, you can precede them with a backslash to remove their special meaning: \\[ or \\\\".

Some of the special sequences beginning with '\\' represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that is not whitespace.

Let's take an example: \\w matches any alphanumeric character. If the regex pattern is expressed in bytes, this is equivalent to the class [a-zA-Z0-9\_]. If the regex pattern is a string, \\w will match all the characters marked as letters in the Unicode database provided by the unicodedata module. You can use the more restricted definition of \\w in a string pattern by supplying the `re.ASCII` flag when compiling the regular expression. The following list of special sequences is not complete.

- \\d Matches any decimal digit; this is equivalent to the class [0-9].
- \\D Matches any non-digit character; this is equivalent to the class [^0-9].
- \\s Matches any whitespace character; this is equivalent to the class [\\t\\n\\r\\f\\v].
- \\S Matches any non-whitespace character; this is equivalent to the class [^t\\n\\r\\f\\v].
- \\w Matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9\_].
- \\W Matches any non-alphanumeric character; this is equivalent to the class [^a-zA-Z0-9\_].

These sequences can be included inside a character class. For example, [\\s,.] is a character class that will match any whitespace character, or ., or .. The final metacharacter in this section is .. It matches anything except a newline character, and there's an alternate mode (`re.DOTALL`) where it will match even a newline. . is often used where you want to match "any character".

## 2.7.2 Using Regular Expressions

Now that we've looked at some simple regular expressions, how do we actually use them in Python? The `re` module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them.

### Compiling Regular Expressions

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```
import re
p = re.compile('ab*')
p
```

`re.compile()` also accepts an optional `flags` argument, used to enable various special features and syntax variations. We'll go over the available settings later, but for now a single example will do:

```
p = re.compile('ab*', re.IGNORECASE)
```

The RE is passed to `re.compile()` as a string. REs are handled as strings because regular expressions aren't part of the core Python language, and no special syntax was created for expressing them. (There are applications that don't need REs at all, so there's no need to bloat the language specification by including them.) Instead, the `re` module is simply a C extension module included with Python, just like the `socket` or `zlib` modules. Putting REs in strings keeps the Python language simpler, but has one disadvantage which is the topic of the next section.

### The Backslash Plague

As stated earlier, regular expressions use the backslash character ('\\') to indicate special forms or to allow special characters to be used without invoking their special meaning. This conflicts with Python's usage of the same character for the same purpose in string literals.

Let's say you want to write a RE that matches the string `\section`, which might be found in a LaTeX file. To figure out what to write in the program code, start with the desired string to be matched. Next, you must escape any backslashes and other metacharacters by preceding them with a backslash, resulting in the string `\\\section`. The resulting string that must be passed to `re.compile()` must be `\\\section`. However, to express this as a Python string literal, both backslashes must be escaped again.

Characters	Stage
<code>\section</code>	Text string to be matched
<code>\\\section</code>	Escaped backslash for <code>re.compile()</code>
<code>"\\\\\\section"</code>	Escaped backslashes for a string literal

Table 2.1: \ for escape

In short, to match a literal backslash, one has to write '`\\\\"`' as the RE string, because the regular expression must be `\\"`, and each backslash must be expressed as `\\\`inside a regular Python string literal. In REs that feature backslashes repeatedly, this leads to lots of repeated backslashes and makes the resulting strings difficult to understand.

The solution is to use Python's raw string notation for regular expressions; backslashes are not handled in any special way in a string literal prefixed with 'r', so `r"\n"` is a two-character string containing '`\`' and '`n`', while `"\n"` is a one-character string containing a newline. Regular expressions will often be written in Python code using this raw string notation.

In addition, special escape sequences that are valid in regular expressions, but not valid as Python string literals, now result in a `DeprecationWarning` and will eventually become a `SyntaxError`, which means the sequences will be invalid if raw string notation or escaping the backslashes isn't used.

## Performing Matches

Once you have an object representing a compiled regular expression, what do you do with it? Pattern objects have several methods and

Regular String	Raw string
"ab*"	r"ab*"
"\\section"	r"\\"section"
"\w+\s+\1"	r"\w+\s+\1"

Table 2.2: Regular vs. raw expressions

attributes. Only the most significant ones will be covered here; consult the `re` docs for a complete listing.

Method/Attribute	Purpose
<code>match()</code>	Determine if the RE matches at the beginning of the string.
<code>search()</code>	Scan through a string, looking for any location where this RE matches.
<code>findall()</code>	Find all substrings where the RE matches, and returns them as a list.
<code>finditer()</code>	Find all substrings where the RE matches, and returns them as an iterator.

Table 2.3:

`match()` and `search()` return `None` if no match can be found. If they're successful, a match object instance is returned, containing information about the match: where it starts and ends, the substring it matched, and more. You can learn about this by interactively experimenting with the `re` module. If you have `tkinter` available, you may also want to look at <https://github.com/python/cpython/blob/3.9/Tools/demo/redemo.py>, a demonstration program included with the Python distribution. It allows you to enter REs and strings, and displays whether the RE matches or fails. `redemo.py` can be quite useful when trying to debug a complicated RE. This HOWTO uses the standard Python interpreter for its examples. First, run the Python interpreter, import the `re` module, and compile a RE:

```
import re
p = re.compile('[a-z]+')
p
```

Now, you can try matching various strings against the RE  $[a - z]^+$ . An empty string should not match at all, since `+` means ‘one or more repetitions’. `match()` should return `None` in this case, which will cause the interpreter to print no output. You can explicitly print the result of `match()` to make this clear.

```
p.match("")  
print(p.match(""))
```

Now, let’s try it on a string that it should match, such as `tempo`. In this case, `match()` will return a match object, so you should store the result in a variable for later use.

```
m = p.match('tempo')  
m
```

Now you can query the match object for information about the matching string. Match object instances also have several methods and attributes; the most important ones are:

Method/Attribute	Purpose
<code>group()</code>	Return the string matched by the RE
<code>start()</code>	Return the starting position of the match
<code>end()</code>	Return the ending position of the match
<code>span()</code>	Return a tuple containing the (start, end) positions of the match

Table 2.4:

Trying these methods will soon clarify their meaning:

```
>>> m.group()  
'tempo'  
>>> m.start(), m.end()  
(0, 5)  
>>> m.span()  
(0, 5)
```

`group()` returns the substring that was matched by the RE. `start()` and `end()` return the starting and ending index of the match. `span()` returns both start and end indexes in a single tuple. Since the `match()`

method only checks if the RE matches at the start of a string, `start()` will always be zero. However, the `search()` method of patterns scans through the string, so the match may not start at zero in that case.

```
>>> print(p.match('::: message'))
None
>>> m = p.search('::: message'); print(m)
<re.Match object; span=(4, 11), match='message'>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

In actual programs, the most common style is to store the match object in a variable, and then check if it was `None`. This usually looks like:

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print('Match found: ', m.group())
else:
    print('No match')
```

Two pattern methods return all of the matches for a pattern. `findall()` returns a list of matching strings:

```
>>> p = re.compile(r'\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords
              ↪ a-leaping')
['12', '11', '10']
```

The `r` prefix, making the literal a raw string literal, is needed in this example because escape sequences in a normal “cooked” string literal that are not recognized by Python, as opposed to regular expressions, now result in a `DeprecationWarning` and will eventually become a `SyntaxError`. See [The Backslash Plague](#). `findall()` has to create the entire list before it can be returned as the result. The `finditer()` method returns a sequence of match object instances as an iterator:

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...
                           ↪ ')
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
```

```
...     print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

### 2.7.3 Module-Level Functions

You don't have to create a pattern object and call its methods; the `re` module also provides top-level functions called `match()`, `search()`, `findall()`, `sub()`, and so forth. These functions take the same arguments as the corresponding pattern method with the RE string added as the first argument, and still return either `None` or a match object instance.

```
>>> print(re.match(r'From\s+', 'Fromage amk'))
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<re.Match object; span=(0, 5), match='From '>
```

Under the hood, these functions simply create a pattern object for you and call the appropriate method on it. They also store the compiled object in a cache, so future calls using the same RE won't need to parse the pattern again and again. Should you use these module-level functions, or should you get the pattern and call its methods yourself? If you're accessing a regex within a loop, pre-compiling it will save a few function calls. Outside of loops, there's not much difference thanks to the internal cache.

### 2.7.4 Compilation Flags

Compilation flags let you modify some aspects of how regular expressions work. Flags are available in the `re` module under two names, a long name such as `IGNORECASE` and a short, one-letter form such as `I`. (If you're familiar with Perl's pattern modifiers, the one-letter forms use the same letters; the short form of `re.VERBOSE` is `re.X`, for example.) Multiple flags can be specified by bitwise OR-ing them; `re.I | re.M` sets both the `I` and `M` flags, for example.

Here's a table of the available flags, followed by a more detailed explanation of each one.

Flag	Meaning
ASCII, A	Makes several escapes like \w, \b, \s and \d match only on ASCII characters with the respective property.
DOTALL, S	Make . match any character, including newlines.
IGNORECASE, I	Do case-insensitive matches.
LOCALE, L	Do a locale-aware match.
MULTILINE, M	Multi-line matching, affecting ^ and \$.
VERBOSE, X (for 'extended')	Enable verbose REs, which can be organized more cleanly and understandably.

Table 2.5: Flags for regex

^ Matches at the beginning of lines. Unless the MULTILINE flag has been set, this will only match at the beginning of the string. In MULTILINE mode, this also matches immediately after each newline within the string. For example, if you wish to match the word From only at the beginning of a line, the RE to use is `^From`.

```
>>> print(re.search('^From', 'From Here to Eternity'))
<re.Match object; span=(0, 4), match='From'>
>>> print(re.search('^From', 'Reciting From Memory'))
None
To match a literal '^', use \^.
```

\$ matches at the end of a line, which is defined as either the end of the string, or any location followed by a newline character.

```
>>> print(re.search('}\$\n', '{block}'))
<re.Match object; span=(6, 7), match='}'>
```

```
>>> print(re.search('}\$\n', '{block} '))
None
>>> print(re.search('}\$\n', '{block}\n'))
<re.Match object; span=(6, 7), match='}'>
To match a literal '\$', use \$ or enclose it inside a
    ↪ character class, as in [\$].
```

`\b` is used for Word boundary. This is a zero-width assertion that matches only at the beginning or end of a word. A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character. The following example matches `class` only when it's a complete word; it won't match when it's contained inside another word.

```
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<re.Match object; span=(3, 8), match='class'>
>>> print(p.search('the declassified algorithm'))
None
>>> print(p.search('one subclass is'))
None
```

There are two subtleties you should remember when using this special sequence. First, this is the worst collision between Python's string literals and regular expression sequences. In Python's string literals, `\b` is the backspace character, ASCII value 8. If you're not using raw strings, then Python will convert the `\b` to a backspace, and your RE won't match as you expect it to. The following example looks the same as our previous RE, but omits the `'r'` in front of the RE string.

```
>>> p = re.compile('\bclass\b')
>>> print(p.search('no class at all'))
None
>>> print(p.search('\b' + 'class' + '\b'))
<re.Match object; span=(0, 7), match='\x08class\x08'>
```

Second, inside a character class, where there's no use for this assertion, represents the backspace character, for compatibility with Python's string literals.

### 2.7.5 Grouping

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a ':', like this:

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
MIME-Version: 1.0
To: editor@example.com
```

This can be handled by writing a regular expression which matches an entire header line, and has one group which matches the header name, and another group which matches the header's value. Groups are marked by the '(', ')' metacharacters. '(' and ')' have much the same meaning as they do in mathematical expressions; they group together the expressions contained inside them, and you can repeat the contents of a group with a repeating qualifier, such as \*, +, ?, or m,n. For example, (ab)\* will match zero or more repetitions of ab.

```
>>> p = re.compile('(ab)*')
>>> print(p.match('abababab').span())
(0, 10)
```

Groups indicated with '(', ')' also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to group(), start(), end(), and span(). Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so match object methods all have group 0 as their default argument. Later we'll see how to express groups that don't capture the span of text that they match.

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

Subgroups are numbered from left to right, from 1 upward. Groups can be nested; to determine the number, just count the opening parenthesis characters, going from left to right.

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

`group()` can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

```
>>> m.groups()
('b', 'abc', 'b')
```

The `groups()` method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```
>>> m.groups()
('abc', 'b')
```

Backreferences in a pattern allow you to specify that the contents of an earlier capturing group must also be found at the current location in the string. For example, will succeed if the exact contents of group 1 can be found at the current position, and fails otherwise. Remember that Python's string literals also use a backslash followed by numbers to allow including arbitrary characters in a string, so be sure to use a raw string when incorporating backreferences in a RE. For example, the following RE detects doubled words in a string.

```
>>> p = re.compile(r'\b(\w+)\s+\1\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

Backreferences like this aren't often useful for just searching through a string — there are few text formats which repeat data in this way — but you'll soon find out that they're very useful when performing string substitutions.

### 2.7.6 Non-capturing and Named Groups

Elaborate REs may use many groups, both to capture substrings of interest, and to group and structure the RE itself. In complex REs, it becomes difficult to keep track of the group numbers. There are two features which help with this problem. Both of them use a common syntax for regular expression extensions, so we'll look at that first.

Perl 5 is well known for its powerful additions to standard regular expressions. For these new features the Perl developers couldn't choose new single-keystroke metacharacters or new special sequences beginning with without making Perl's regular expressions confusingly different from standard REs. If they chose as a new metacharacter, for example, old expressions would be assuming that was a regular character and wouldn't have escaped it by writing \& or [&].

The solution chosen by the Perl developers was to use (...) as the extension syntax. ? immediately after a parenthesis was a syntax error because the ? would have nothing to repeat, so this didn't introduce any compatibility problems. The characters immediately after the ? indicate what extension is being used, so (?=foo) is one thing (a positive lookahead assertion) and (?:foo) is something else (a non-capturing group containing the subexpression foo).

Python supports several of Perl's extensions and adds an extension syntax to Perl's extension syntax. If the first character after the question mark is a P, you know that it's an extension that's specific to Python. Now that we've looked at the general extension syntax, we can return to the features that simplify working with groups in complex REs. Sometimes you'll want to use a group to denote a part of a regular expression, but aren't interested in retrieving the group's contents. You can make this fact explicit by using a non-capturing group: (?...), where you can replace the ... with any other regular expression.

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

Except for the fact that you can't retrieve the contents of what the group matched, a non-capturing group behaves exactly the same as a capturing group; you can put anything inside it, repeat it with a repetition metacharacter such as `*`, and nest it within other groups (capturing or non-capturing). `(?:...)` is particularly useful when modifying an existing pattern, since you can add new groups without changing how all the other groups are numbered. It should be mentioned that there's no performance difference in searching between capturing and non-capturing groups; neither form is any faster than the other. A more significant feature is named groups: instead of referring to them by numbers, groups can be referenced by a name. The syntax for a named group is one of the Python-specific extensions: `(?P<name>...)`. `name` is, obviously, the name of the group. Named groups behave exactly like capturing groups, and additionally associate a name with a group. The match object methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group's name. Named groups are still given numbers, so you can retrieve information about a group in two ways:

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search( '(((( Lots of punctuation ))))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

Additionally, you can retrieve named groups as a dictionary with `groupdict()`:

```
>>> m = re.match(r'(?P<first>\w+) (?P<last>\w+)', 'Jane Doe')
>>> m.groupdict()
{'first': 'Jane', 'last': 'Doe'}
```

### 2.7.7 Modifying Strings

Up to this point, we've simply performed searches against a static string. Regular expressions are also commonly used to modify strings in various ways, using the following pattern methods:

Method/Attribute	Purpose
split()	Split the string into a list, splitting it wherever the RE matches
sub()	Find all substrings where the RE matches, and replace them with a different string
subn()	Does the same thing as sub(), but returns the new string and the number of replacements

Table 2.6:

### 2.7.8 Splitting Strings

The `split()` method of a pattern splits a string apart wherever the RE matches, returning a list of the pieces. It's similar to the `split()` method of strings but provides much more generality in the delimiters that you can split by; string `split()` only supports splitting by whitespace or by a fixed string. As you'd expect, there's a module-level `re.split()` function, too. You can limit the number of splits made, by passing a value for `maxsplit`. When `maxsplit` is nonzero, at most `maxsplit` splits will be made, and the remainder of the string is returned as the final element of the list. In the following example, the delimiter is any sequence of non-alphanumeric characters.

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', ''
                                         ^ split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

Sometimes you're not only interested in what the text between delimiters is, but also need to know what the delimiter was. If capturing parentheses are used in the RE, then their values are also returned as part of the list. Compare the following calls:

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
```

```
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '...', 'is', ' ', 'a', ' ', 'test', '.', '']
```

The module-level function `re.split()` adds the RE to be used as the first argument, but is otherwise the same.

```
>>> re.split(r'[\W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'([\W]+)', 'Words, words, words.')
['Words', ' ', ' ', 'words', ' ', ' ', 'words', '.', '']
>>> re.split(r'[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

## 2.7.9 Search and Replace

Another common task is to find all the matches for a pattern, and replace them with a different string. The `sub()` method takes a replacement value, which can be either a string or a function, and the string to be processed. Here's a simple example of using the `sub()` method. It replaces colour names with the word colour:

```
>>> p = re.compile('(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

The `subn()` method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed:

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

Empty matches are replaced only when they're not adjacent to a previous empty match.

```
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b--d-'
```

If replacement is a string, any backslash escapes in it are processed. That is, \n is converted to a single newline character, \r is converted to a carriage return, and so forth. Unknown escapes such as \& are left alone. Backreferences, such as , are replaced with the substring matched by the corresponding group in the RE. This lets you incorporate portions of the original text in the resulting replacement string. This example matches the word `section` followed by a string enclosed in , , and changes `section` to `subsection`:

```
>>> p = re.compile('section{ ( [^}]*) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}', 'section{First} section{second}')
'subsection{First} subsection{second}'
```

There's also a syntax for referring to named groups as defined by the (?P<name>...) syntax. \g<name> will use the substring matched by the group named name, and \g<number> uses the corresponding group number. \g<2> is therefore equivalent to \2, but isn't ambiguous in a replacement string such as \g<2>0. (\20 would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character '0'.) The following substitutions are all equivalent, but use all three variations of the replacement string.

```
>>> p = re.compile('section{ (?P<name> [^}]*) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}', 'section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}', 'section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}', 'section{First}')
'subsection{First}'
```

replacement can also be a function, which gives you even more control. If replacement is a function, the function is called for every non-overlapping occurrence of pattern. On each call, the function is passed a match object argument for the match and can use this information to compute the desired replacement string and return it. In

the following example, the replacement function translates decimals into hexadecimal:

```
def hexrepl(match):
    "Return the hex string for a decimal number"
    value = int(match.group())
    return hex(value)

>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user
                           ↪ code.')
'Call 0xffffd2 for printing, 0xc000 for user code.'
```

When using the module-level `re.sub()` function, the pattern is passed as the first argument. The pattern may be provided as an object or as a string; if you need to specify regular expression flags, you must either use a pattern object as the first parameter, or use embedded modifiers in the pattern string, e.g. `sub("(?i)b+", "x", "bbbb BBBB")` returns '`x x`'.

## Notes

<sup>33</sup>For variety of *operators* refer to [https://www.tutorialspoint.com/python/python\\_basic\\_operators.htm](https://www.tutorialspoint.com/python/python_basic_operators.htm). There is a module for operators known as *operator*. Visit <https://docs.python.org/2/library/operator.html> for documentation.

<sup>34</sup>Read more about *string* module at <https://docs.python.org/2/library/string.html>

<sup>35</sup>You may visit <https://docs.python.org/3/tutorial/datastructures.html> for more details.

<sup>36</sup>Retrieved from Python Software Foundation. Visit <https://docs.python.org/3/tutorial/datastructures.html> for more information on *dictionary*

<sup>37</sup>This PEP is driven by the desire to have a simpler way to format strings in Python. Read about this PEP at <https://www.python.org/dev/peps/pep-0498/#rationale>

<sup>38</sup>Nørmark, Kurt. Overview of the four main programming paradigms. Aalborg University, 9 May 2011. Retrieved 22 September 2012. Available at [http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms\\_themes-paradigm-overview-section.html](http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html)

<sup>39</sup>Read more from <https://www.programiz.com/python-programming/function>

<sup>40</sup>Read more about *identifiers* at <https://www.programiz.com/python-programming/keywords-identifier#rules>

<sup>41</sup>Retrieved from <https://www.pythontutorial.eu/lambda.php>

<sup>42</sup>Read about *arguments* at <https://docs.python.org/3/tutorial/controlflow.html#positional-only-parameters>

<sup>43</sup>Retrieved from <https://docs.python.org/3/tutorial/datastructures.html#nested-list-comprehensions>

<sup>44</sup>Zero matrix is one of the very influential matrices often used while processing data in data sciences. In statistics, zero matrices are used in regression methods.

<sup>45</sup>Symmetric matrix is different from square matrix. A symmetric matrix is a matrix which satisfied the property  $A = A^T$ , where  $A^T$  is known as *Transpose A* or *A Transpose*

<sup>46</sup>A zero matrix or null matrix is a matrix all of whose entries are zero. There are number of applications in mathematics using zeros matrix. For instance, *undecidable problem* in decision science, *annihilator matrix* in OLS regression are only few to name with.

<sup>47</sup>Read more about identity matrices at [https://en.wikipedia.org/wiki/Invertible\\_matrix](https://en.wikipedia.org/wiki/Invertible_matrix)

<sup>48</sup>

<sup>49</sup>Arthur Cayley FRS was a prolific British mathematician who worked mostly on algebra. He helped found the modern British school of pure mathematics. Cayley's treatise on geometric transformations using matrices is one of the most renowned classics in geometry. Cayley studies and explains properties of matrices using very simple arithmetic calculations such as addition, subtraction, multiplication and division.

<sup>50</sup>Read more about *csr\_data()* at [https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.sparse.csr\\_matrix.html](https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.sparse.csr_matrix.html)

<sup>51</sup>Visit <https://stackoverflow.com/questions/448271/what-is-init-py-for> for more details.

about `__init__.py` is, perhaps, <https://docs.python.org/3.3/tutorial/modules.html>. This is Python official documentation.

<sup>52</sup>The other better place to learn

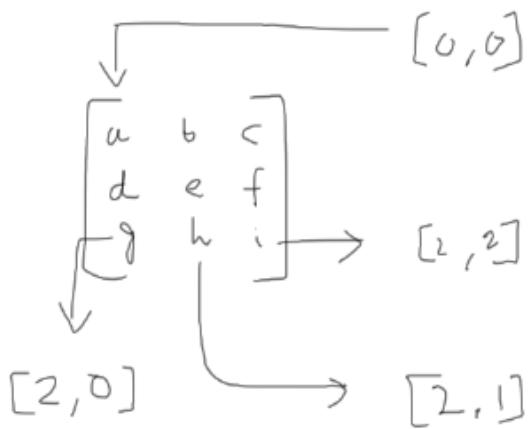


Figure 2.1: Matrix Indexes

## Chapter 3

# Objective Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of “objects”, which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

A feature of objects is that an object’s own procedures can access and often modify the data fields of itself (objects have a notion of this or self). In OOP, computer programs are designed by making them out of objects that interact with one another. OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

Many of the most widely used programming languages (such as C++, Java, Python, etc.) are multi-paradigm and they support object-oriented programming to a greater or lesser degree, typically in combination with imperative, procedural programming. Significant object-oriented languages include: (list order based on TIOBE index) Java, C++, C#, Python, R, PHP, Visual Basic.NET, JavaScript, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Kotlin, Common Lisp, MATLAB, and Smalltalk.

Object-oriented programming uses objects, but not all of the associated techniques and structures are supported directly in languages that claim to support OOP. The features listed below are common among languages considered to be strongly class- and object-oriented (or multi-paradigm with OOP support), with notable exceptions mentioned.

### 3.1 Objects and classes

Languages that support object-oriented programming (OOP) typically use inheritance for code reuse and extensibility in the form of either classes or prototypes. Those that use classes support two main concepts:

- Classes – the definitions for the data format and available procedures for a given type or class of object; may also contain data and procedures (known as class methods) themselves, i.e. classes contain the data members and member functions
- Objects – instances of classes

Objects sometimes correspond to things found in the real world. For example, a graphics program may have objects such as "circle", "square", "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product". Sometimes objects represent more abstract entities, like an object that represents an open file, or an object that provides the service of translating measurements from U.S. customary to metric.

Each object is said to be an instance of a particular class (for example, an object with its name field set to "Mary" might be an instance of class Employee). Procedures in object-oriented programming are known as methods; variables are also known as fields, members, attributes, or properties. This leads to the following terms:

- Class variables – belong to the class as a whole; there is only one copy of each one
- Instance variables or attributes – data that belongs to individual objects; every object has its own copy of each one

- Member variables – refers to both the class and instance variables that are defined by a particular class
- Class methods – belong to the class as a whole and have access only to class variables and inputs from the procedure call
- Instance methods – belong to individual objects, and have access to instance variables for the specific object they are called on, inputs, and class variables

Objects are accessed somewhat like variables with complex internal structure, and in many languages are effectively pointers, serving as actual references to a single instance of said object in memory within a heap or stack. They provide a layer of abstraction which can be used to separate internal from external code. External code can use an object by calling a specific instance method with a certain set of input parameters, read an instance variable, or write to an instance variable. Objects are created by calling a special type of method in the class known as a constructor. A program may create many instances of the same class as it runs, which operate independently. This is an easy way for the same procedures to be used on different sets of data. Object-oriented programming that uses classes is sometimes called class-based programming, while prototype-based programming does not typically use classes. As a result, significantly different yet analogous terminology is used to define the concepts of object and instance.

An object can be a variable, a data structure, a function, or a method, and as such, is a value in memory referenced by an identifier. In the object-oriented programming paradigm object can be a combination of variables, functions, and data structures; in particular in class-based flavour of the paradigm it refers to a particular instance of a class. In the relational model of database management, an object can be a table or column, or an association between data and a database entity (such as relating a person's age to a specific person). Python is a multi-paradigm programming language. It supports different programming approaches. One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP). An object has two characteristics:

- attributes
- behavior

In Python, the concept of OOP follows some basic principles:

### Class

A class is a blueprint for the object. We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object. The example for class of parrot can be:

```
class Parrot:  
    pass
```

Here, we use the `class` keyword to define an empty class `Parrot`. From class, we construct instances. An instance is a specific object created from a particular class. Like function definitions begin with the `def` keyword in Python, class definitions begin with a `class` keyword. The first string inside the class is called docstring and has a brief description about the class. Although not mandatory, this is highly recommended. A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions. There are also special attributes in it that begins with double underscores `__`. For example, `__doc__` gives us the docstring of that class. As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

### Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated. The example for object of parrot class can be:

```
obj = Parrot()
```

Here, `obj` is an object of class `Parrot`. Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots. It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

### 3.1.1 Creating Class and Object in Python

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

#### Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

In the above program, we created a class with the name `Parrot`. Then, we define attributes. The attributes are a characteristic of an object. These attributes are defined inside the `__init__` method of the class. It is the initializer method that is first run as soon as the object is created. Then, we create instances of the `Parrot` class. Here, `blu` and `woo` are references (value) to our new objects. We can access the class

attribute using `__class__.species`. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using `blu.name` and `blu.age`. However, instance attributes are different for every instance of a class.

## Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object. Following example shows as how to create methods in Python.

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

## Output

```
Blu sings 'Happy'
Blu is now dancing
```

In the above program, we define two methods i.e `sing()` and `dance()`. These are called instance methods because they are called on an instance object i.e `blu`.

## 3.2 Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

### 3.2.1 Use of Inheritance in Python

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Output

```
Bird is ready
Penguin is ready
Penguin
```

```
Swim faster
Run faster
```

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from the `swim()` method. Again, the child class modified the behavior of the parent class. We can see this from the `whoisThis()` method. Furthermore, we extend the functions of the parent class, by creating a new `run()` method. Additionally, we use the `super()` function inside the `__init__()` method. This allows us to run the `__init__()` method of the parent class inside the child class.

### Python Multiple Inheritance

A class can be derived from more than one base class in Python, similar to C++. This is called multiple inheritance. In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

We can also inherit from a derived class. This is called multilevel inheritance. It can be of any depth in Python. In multilevel inheritance, features of the base class and the derived class are inherited into the new derived class.

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```

## Method Resolution Order in Python

Every class in Python is derived from the object class. It is the most base type in Python. So technically, all other classes, either built-in or user-defined, are derived classes and all objects are instances of the object class.

```
# Output: True
print(issubclass(list,object))

# Output: True
print(isinstance(5.5,object))

# Output: True
print(isinstance("Hello",object))
```

In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching the same class twice. So, in the above example of MultiDerived class the search order is [MultiDerived, Base1, Base2, object]. This order is also called linearization of MultiDerived class and the set of rules used to find this order is called Method Resolution Order (MRO). MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents. In case of multiple parents, the order is the same as tuples of base classes. MRO of a class can be viewed as the `__mro__` attribute or the `mro()` method. The former returns a tuple while the latter returns a list.

```
>>> MultiDerived.__mro__
(<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>)

>>> MultiDerived.mro()
[<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>]
```

### 3.3 Python Operator Overloading

Python operators work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings. This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading. So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

p1 = Point(1, 2)
p2 = Point(2, 3)
print(p1+p2)
```

Output

```
Traceback (most recent call last):
  File "<string>", line 9, in <module>
    print(p1+p2)
TypeError: unsupported operand type(s) for +: 'Point' and '
                           ^ Point'
```

Here, we can see that a `TypeError` was raised, since Python didn't know how to add two `Point` objects together. However, we can achieve this task in Python through operator overloading. But first, let's get a notion about special functions.

#### 3.3.1 Python Special Functions

Class functions that begin with double underscore `__` are called special functions in Python. These functions are not the typical functions that we define for a class. The `__init__()` function we defined above is one of them. It gets called every time we create a new object of that class. There are numerous other special functions in Python. Visit

Python Special Functions to learn more about them. Using special functions, we can make our class compatible with built-in functions.

```
>>> p1 = Point(2,3)
>>> print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

Suppose we want the `print()` function to print the coordinates of the `Point` object instead of what we got. We can define a `__str__()` method in our class that controls how the object gets printed. Let's look at how we can achieve this:

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "{},{}".format(self.x, self.y)
```

Now let's try the `print()` function again.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "{}, {}".format(self.x, self.y)

p1 = Point(2, 3)
print(p1)
```

Output

```
(2, 3)
```

That's better. Turns out, that this same method is invoked when we use the built-in function `str()` or `format()`.

```
>>> str(p1)
'(2,3)'
```

```
>>> format(p1)
'(2,3)'
```

So, when you use str(p1) or format(p1), Python internally calls the p1.\_\_str\_\_() method. Hence the name, special functions.

### 3.3.2 Overloading Operators

To overload the + operator, we will need to implement \_\_add\_\_() function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is more sensible to return a Point object of the coordinate sum.

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)

Now let's try the addition operation again:

class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)

p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1+p2)
```

## Output

```
(3,5)
```

What actually happens is that, when you use  $p1 + p2$ , Python calls `p1.__add__(p2)` which in turn is `Point.__add__(p1,p2)`. After this, the addition operation is carried out the way we specified. Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	<code>p1 + p2</code>	<code>p1.__add__(p2)</code>
Subtraction	<code>p1 - p2</code>	<code>p1.__sub__(p2)</code>
Multiplication	<code>p1 * p2</code>	<code>p1.__mul__(p2)</code>
Power	<code>p1 ** p2</code>	<code>p1.__pow__(p2)</code>
Division	<code>p1 / p2</code>	<code>p1.__truediv__(p2)</code>
Floor Division	<code>p1 // p2</code>	<code>p1.__floordiv__(p2)</code>
Remainder (modulo)	<code>p1 % p2</code>	<code>p1.__mod__(p2)</code>
Bitwise Left Shift	<code>p1 p2</code>	<code>p1.__lshift__(p2)</code>
Bitwise Right Shift	<code>p1 » p2</code>	<code>p1.__rshift__(p2)</code>
Bitwise AND	<code>p1</code>	<code>p2</code>
	<code>p1.__and__(p2)</code>	
Bitwise OR	<code>p1    p2</code>	<code>p1.__or__(p2)</code>
Bitwise XOR	<code>p1 ^ p2</code>	<code>p1.__xor__(p2)</code>
Bitwise NOT	<code>~ p1</code>	<code>p1.__invert__()</code>

Table 3.1: Overloading operator

### 3.3.3 Overloading Comparison Operators

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well. Suppose we wanted to implement the less than symbol `<` in our Point class. Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

```
# overloading the less than operator
class Point:
    def __init__(self, x=0, y=0):
```

```

        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)

# use less than
print(p1<p2)
print(p2<p3)
print(p1<p3)

```

### Output

```

True
False
False

```

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Operator	Expression	Internally
Less than	p1 p2	p1.__lt__(p2)
Less than or equal to	p1 = p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 = p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Table 3.2: Less than operator

## 3.4 Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single `_` or double `__`.

### 3.4.1 Data Encapsulation in Python

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

#### Output

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

In the above program, we defined a Computer class. We used `__init__()` method to store the maximum selling price of Computer. We tried to modify the price. However, we can't change it because Python treats the `__maxprice` as private attributes. As shown, to change the value, we have to use a setter function i.e `setMaxPrice()` which takes price as a parameter.

## 3.5 Polymorphism

In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function. Polymorphism gives the ability (in OOP) to use a common interface for multiple forms (data types) of same method. Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism. In Python, polymorphism can happen at different levels, such as

1. Polymorphism with class methods (This is widely used approach)
2. Polymorphism with Inheritance (This is widely used approach)
3. Inbuilt polymorphic functions (Ex. `len()`)
4. User-defined polymorphic functions (Ex. Same function with different number of arguments)
5. Polymorphism with a function and objects (Ex. using interface function)
6. Using inheritance and method overriding  
(Using errors like `NotImplementedError()`)

### 3.5.1 Polymorphism with class methods with interface

```
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
```

```
    print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()

# instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

### Output

```
Parrot can fly
Penguin can't fly
```

In the above program, we defined two classes Parrot and Penguin. Each of them have a common `fly()` method. However, their functions are different. To use polymorphism, we created a common interface i.e `flying_test()` function that takes any object and calls the object's `fly()` method. Thus, when we passed the blu and peggy objects in the `flying_test()` function, it ran effectively.

### 3.5.2 Polymorphism with Inheritance

In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class does not quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as Method Overriding.

```
class Bird:
    def intro(self):
        print("There are many types of birds.")
```

```
def flight(self):
    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

The output will be as follows

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

# Chapter 4

## Numpy

NumPy (Numerical Python) is an open source Python library that's used in almost every field of science and engineering. It's the universal standard for working with numerical data in Python, and it's at the core of the scientific Python and PyData ecosystems. NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development. The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages.

The NumPy library contains multidimensional array and matrix data structures (you'll find more information about this in later sections). It provides ndarray, a homogeneous n-dimensional array object, with methods to efficiently operate on it. NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices.

NumPy, short for Numerical Python, is the fundamental package required for high performance scientific computing and data analysis. It is the foundation on which nearly all of the higher-level tools in this

book are built. Here are some of the things it provides:

1. ndarray, a fast and space-efficient multidimensional array providing vectorized arithmetic operations and sophisticated broadcasting capabilities
2. Standard mathematical functions for fast operations on entire arrays of data without having to write loops
3. Tools for reading / writing array data to disk and working with memory-mapped files
4. Linear algebra, random number generation, and Fourier transform capabilities
5. Tools for integrating code written in C, C++, and Fortran

## 4.1 Installing NumPy

To install NumPy, we strongly recommend using a scientific Python distribution. If you’re looking for the full instructions for installing NumPy on your operating system, see [Installing NumPy](#).

If you already have Python, you can install NumPy with:

```
pip install numpy
```

If you don’t have Python yet, you might want to consider using Anaconda. It’s the easiest way to get started. The good thing about getting this distribution is the fact that you don’t need to worry too much about separately installing NumPy or any of the major packages that you’ll be using for your data analyses, like pandas, Scikit-Learn, etc.

### 4.1.1 How to import NumPy

To access NumPy and its functions import it in your Python code like this:

```
import numpy as np
```

We shorten the imported name to `np` for better readability of code using NumPy. This is a widely adopted convention that you should follow so that anyone working with your code can easily understand it.

### 4.1.2 Reading the example code

If you aren't already comfortable with reading tutorials that contain a lot of code, you might not know how to interpret a code block that looks like this:

```
a = np.arange(6)
a2 = a[np.newaxis, :]
a2.shape
(1, 6)
```

If you aren't familiar with this style, it's very easy to understand. If you see `>>`, you're looking at input, or the code that you would enter. Everything that doesn't have `>>` in front of it is output, or the results of running your code. This is the style you see when you run `python` on the command line, but if you're using IPython, you might see a different style. Note that it is not part of the code and will cause an error if typed or pasted into the Python shell. It can be safely typed or pasted into the IPython shell; the `>>` is ignored.

## 4.2 NumPy Arrays

NumPy gives you an enormous range of fast and efficient ways of creating arrays and manipulating numerical data inside them. While a Python list can contain different data types within a single list, all of the elements in a NumPy array should be homogeneous. The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren't homogeneous.

NumPy arrays are faster and more compact than Python lists. An array consumes less memory and is convenient to use. NumPy uses much less memory to store data and it provides a mechanism of specifying the data types. This allows the code to be optimized even further.

### 4.2.1 Array creation functions

Table 4.1 shows the list of functions available in Numpy library to handle arrays.

Function	Description
<code>array</code>	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.
<code>asarray</code>	Convert input to ndarray, but do not copy if the input is already an ndarray
<code>arange</code>	Like the built-in range but returns an ndarray instead of a list.
<code>ones, ones_like</code>	Produce an array of all 1's with the given shape and dtype. <code>ones_like</code> takes another array and produces a ones array of the same shape and dtype.
<code>zeros, zeros_like</code>	Like <code>ones</code> and <code>ones_like</code> but producing arrays of 0's instead
<code>empty, empty_like</code>	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
<code>eye,</code>	identity Create a square $N \times N$ identity matrix (1's on the diagonal and 0's elsewhere)

Table 4.1: Array creation functions

### 4.2.2 What is an array?

An array is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in various ways. The elements are all of the same type, referred to as the array dtype.

An array can be indexed by a tuple of nonnegative integers, by booleans, by another array, or by integers. The rank of the array is the number of dimensions. The shape of the array is a tuple of integers giving the size of the array along each dimension.

One way we can initialize NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

For example:

```
a = np.array([1, 2, 3, 4, 5, 6])
```

or:

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

We can access the elements in the array using square brackets. When you’re accessing elements, remember that indexing in NumPy starts at 0. That means that if you want to access the first element in your array, you’ll be accessing element “0”.

```
print(a[0])
[1 2 3 4]
```

### 4.2.3 Array dimensionality

You might occasionally hear an array referred to as a “ndarray,” which is shorthand for “N-dimensional array.” An N-dimensional array is simply an array with any number of dimensions. You might also hear 1-D, or one-dimensional array, 2-D, or two-dimensional array, and so on. The NumPy ndarray class is used to represent both matrices and vectors. A vector is an array with a single dimension (there’s no difference between row and column vectors), while a matrix refers to an array with two dimensions. For 3-D or higher dimensional arrays, the term tensor is also commonly used.

What are the attributes of an array?

An array is usually a fixed-size container of items of the same type and size. The number of dimensions and items in an array is defined by its shape. The shape of an array is a tuple of non-negative integers that specify the sizes of each dimension.

In NumPy, dimensions are called axes. This means that if you have a 2D array that looks like this:

```
[[0., 0., 0.],
 [1., 1., 1.]]
```

Your array has 2 axes. The first axis has a length of 2 and the second axis has a length of 3.

Just like in other Python container objects, the contents of an array can be accessed and modified by indexing or slicing the array. Unlike the typical container objects, different arrays can share the same data, so changes made on one array might be visible in another.

Array attributes reflect information intrinsic to the array itself. If you need to get, or even set, properties of an array without creating a new array, you can often access an array through its attributes.

#### 4.2.4 How to create a basic array

To create a NumPy array, you can use the function `np.array()`.

All you need to do to create a simple array is pass a list to it. If you choose to, you can also specify the type of data in your list. You can find more information about data types here.

```
import numpy as np  
a = np.array([1, 2, 3])
```

Besides creating an array from a sequence of elements, you can easily create an array filled with 0's:

```
np.zeros(2)  
array([0., 0.])
```

Or an array filled with 1's:

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

Or even an empty array! The function `empty` creates an array whose initial content is random and depends on the state of the memory. The reason to use `empty` over `zeros` (or something similar) is speed - just make sure to fill every element afterwards!

```
# Create an empty array with 2 elements  
np.empty(2)  
array([3.14, 42.]) # may vary
```

You can create an array with a range of elements:

```
np.arange(4)  
array([0, 1, 2, 3])
```

And even an array that contains a range of evenly spaced intervals. To do this, you will specify the first number, last number, and the step size.

```
np.arange(2, 9, 2)
array([2, 4, 6, 8])
```

You can also use `np.linspace()` to create an array with values that are spaced linearly in a specified interval:

```
np.linspace(0, 10, num=5)
array([ 0.,  2.5,  5.,  7.5, 10.])
```

### 4.2.5 Specifying your data type

While the default data type is floating point (`np.float64`), you can explicitly specify which data type you want using the `dtype` keyword. The data type or `dtype` is a special object containing the information the ndarray needs to interpret a chunk of memory as a particular type of data:

```
>>> arr1 = np.array([1, 2, 3], dtype=np.float64)
>>> arr2 = np.array([1, 2, 3], dtype=np.int32)
>>> arr1.dtype
dtype('float64')
```

Dtypes are part of what make NumPy so powerful and flexible. In most cases they map directly onto an underlying machine representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like *float* or *int*, followed by a number indicating the number of bits per element. A standard double-precision floating point value (what's used under the hood in Python's `float` object) takes up *8 bytes* or *64 bits*. Thus, this type is known in NumPy as *float64*. See Table 4.2 for a full listing of NumPy's supported data types.

You can explicitly convert or cast an array from one `dtype` to another using ndarray's `astype` method:

```
>>> arr = np.array([1, 2, 3, 4, 5])
>>> arr.dtype
dtype('int32')
>>> arr.astype(np.int64)
```

Type	Type Code	Description
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 32-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point. Compatible with C float
float64	f8 or d	Standard double-precision floating point. Compatible with C double and Python float object
float128	f16 or g	Extended-precision floating point
complex64,		
complex128,		
complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool		Boolean type storing True and False values
object	O	Python object type
string_	S	Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use 'S10'.
unicode_	U	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as string (e.g., 'U10').

Table 4.2: NumPy data types

```

array([1, 2, 3, 4, 5], dtype=int64)
>>> arr.dtype
dtype('int32')
>>> arr = arr.astype(np.int64)
>>> arr.dtype
dtype('int64')

```

The object `arr` need to be saved as another object to change the type of the data. An array of strings representing numbers can be changed using `astype` to convert them to numeric form:

```

numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.
                           ↪ string_)
numeric_strings.astype(float)
array([ 1.25, -9.6, 42. ])

```

It is also possible to use another array's `dtype` attribute

```

>>> int_array = np.arange(10)
>>> calibers = np.array([.22, .270, .357, .380, .44, .50],
                           ↪ dtype=np.float64)
>>> calibers.dtype
dtype('float64')
>>> int_array.astype(calibers.dtype)
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])

```

There are shorthand type code strings, refer to table 4.2, you can also use to refer to a `dtype`:

```
>>> empty_uint32 = np.empty(8, dtype='u4')
>>> empty_uint32
array([          0, 1074003968,           0, 1075052544,
       1075707904,           0, 1076101120], dtype=uint32)
```

#### 4.2.6 Multidimensional arrays

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
>>> arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> arr2d[2]
array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent. Refer to figure 4.1:

```
>>> arr2d[0][2]
3
>>> arr2d[0, 2]
3
```

In multidimensional arrays, if you omit later indices, the returned object will be a lower-dimensional `ndarray` consisting of all the data along the higher dimensions. So in the  $2 \times 2 \times 3$  array `arr3d`

```
>>> arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10,
   ↪ 11, 12]]])
>>> arr3d
array([[[ 1,  2,  3],
       [ 4,  5,  6]],
      [[ 7,  8,  9],
       [10, 11, 12]])
```

`arr3d[0]` is a  $2 \times 3$  array:

		axis 1			
		0	1	2	
axis 0		0	0, 0	0, 1	0, 2
		1	1, 0	1, 1	1, 2
		2	2, 0	2, 1	2, 2

Figure 4.1: Indexing arrays in Numpy

```
>>> arr3d[0]
array([[1, 2, 3],
       [4, 5, 6]])
```

#### 4.2.7 Adding, removing, and sorting elements

Sorting an element is simple with `np.sort()`. You can specify the axis, kind, and order when you call the function.

If you start with this array:

```
arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])
```

You can quickly sort the numbers in ascending order with:

```
np.sort(arr)
array([1, 2, 3, 4, 5, 6, 7, 8])
```

In addition to `sort`, which returns a sorted copy of an array, you can use:

- `argsort`, which is an indirect sort along a specified axis,
- `lexsort`, which is an indirect stable sort on multiple keys,
- `searchsorted`, which will find elements in a sorted array, and
- `partition`, which is a partial sort.

If you start with these arrays:

```
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
```

You can concatenate them with `np.concatenate()`.

```
np.concatenate((a, b))
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Or, if you start with these arrays:

```
x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6]])
```

You can concatenate them with:

```
np.concatenate((x, y), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In order to remove elements from an array, it's simple to use indexing to select the elements that you want to keep.

#### 4.2.8 shape and size of an array

1. `ndarray.ndim` will tell you the number of axes, or dimensions, of the array.
2. `ndarray.size` will tell you the total number of elements of the array. This is the product of the elements of the array's shape.
3. `ndarray.shape` will display a tuple of integers that indicate the number of elements stored along each dimension of the array. If, for example, you have a 2-D array with 2 rows and 3 columns, the shape of your array is (2, 3).

For example, if you create this array:

```
array_example = np.array([[0, 1, 2, 3],
                         [4, 5, 6, 7]],
                        [[0, 1, 2, 3],
                         [4, 5, 6, 7]],
                        [[0, 1, 2, 3],
                         [4, 5, 6, 7]]])
```

To find the number of dimensions of the array, run:

```
array_example.ndim  
3
```

To find the total number of elements in the array, run:

```
array_example.size  
24
```

And to find the shape of your array, run:

```
array_example.shape  
(3, 2, 4)
```

#### 4.2.9 Reshaping arrays

Using `arr.reshape()` will give a new shape to an array without changing the data. Just remember that when you use the reshape method,

the array you want to produce needs to have the same number of elements as the original array. If you start with an array with 12 elements, you'll need to make sure that your new array also has a total of 12 elements.

If you start with this array:

```
a = np.arange(6)
print(a)
[0 1 2 3 4 5]
```

You can use `reshape()` to reshape your array. For example, you can reshape this array to an array with three rows and two columns:

```
b = a.reshape(3, 2)
print(b)
[[0 1]
 [2 3]
 [4 5]]
```

With `np.reshape`, you can specify a few optional parameters:

```
np.reshape(a, newshape=(1, 6), order='C')
array([0, 1, 2, 3, 4, 5])
```

`a` is the array to be reshaped. `newshape` is the new shape you want. You can specify an integer or a tuple of integers. If you specify an integer, the result will be an array of that length. The shape should be compatible with the original shape. `order`: `C` means to read/write the elements using C-like index order, `F` means to read/write the elements using Fortran-like index order, `A` means to read/write the elements in Fortran-like index order if `a` is Fortran contiguous in memory, C-like order otherwise. (This is an optional parameter and doesn't need to be specified.)

### 4.2.10 Indexing and slicing

You can index and slice *NumPy* arrays in the same ways you can slice Python lists.

```
>>> data = np.array([1, 2, 3])
```

```
>>> data[1]
2
>>> data[0:2]
array([1, 2])
>>> data[1:]
array([2, 3])
>>> data[-2:]
array([2, 3])
```

You may want to take a section of your array or specific array elements to use in further analysis or additional operations. To do that, you'll need to subset, slice, and/or index your arrays. If you want to select values from your array that fulfill certain conditions, it's straightforward with NumPy. For example, if you start with this array:

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

You can easily print all of the values in the array that are less than 5.

```
print(a[a < 5])
[1 2 3 4]
```

You can also select, for example, numbers that are equal to or greater than 5, and use that condition to index an array.

```
five_up = (a >= 5)
print(a[five_up])
[ 5  6  7  8  9 10 11 12]
```

You can select elements that are divisible by 2:

```
divisible_by_2 = a[a%2==0]
print(divisible_by_2)
[ 2  4  6  8 10 12]
```

Or you can select elements that satisfy two conditions using the & and | operators:

```
c = a[(a > 2) & (a < 11)]
print(c)
[ 3  4  5  6  7  8  9 10]
```

You can also make use of the logical operators `and` | in order to return boolean values that specify whether or not the values in an array fulfill a certain condition. This can be useful with arrays that contain names or other categorical values.

```
five_up = (a > 5) | (a == 5)
print(five_up)
[[False False False False]
 [ True  True  True  True]
 [ True  True  True  True]]
```

You can also use `np.nonzero()` to select elements or indices from an array.

Starting with this array:

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

You can use `np.nonzero()` to print the indices of elements that are, for example, less than 5:

```
b = np.nonzero(a < 5)
print(b)
(array([0, 0, 0, 0]), array([0, 1, 2, 3]))
```

In this example, a tuple of arrays was returned: one for each dimension. The first array represents the row indices where these values are found, and the second array represents the column indices where the values are found.

If you want to generate a list of coordinates where the elements exist, you can zip the arrays, iterate over the list of coordinates, and print them. For example:

```
list_of_coordinates= list(zip(b[0], b[1]))
for coord in list_of_coordinates:
    print(coord)
(0, 0)
(0, 1)
(0, 2)
(0, 3)
```

You can also use `np.nonzero()` to print the elements in an array that are less than 5 with:

```
print(a[b])
[1 2 3 4]
```

If the element you're looking for doesn't exist in the array, then the returned array of indices will be empty. For example:

```
not_there = np.nonzero(a == 42)
print(not_there)
(array([], dtype=int64), array([], dtype=int64))
```

## 4.3 Vectorized operations

### 4.3.1 Along axis

It is possible to apply a function to 1-D slices along the given axis.

```
>>> def my_func(a):
...     """Average first and last element of a 1-D array"""
...     return (a[0] + a[-1]) * 0.5
>>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np.apply_along_axis(my_func, 0, b)
array([4., 5., 6.])
>>> np.apply_along_axis(my_func, 1, b)
array([2., 5., 8.])
```

For a function that returns a 1D array, the number of dimensions in `outarr` is the same as `arr`.

```
>>> b = np.array([[8,1,7], [4,3,9], [5,2,6]])
>>> b
array([[8, 1, 7],
       [4, 3, 9],
       [5, 2, 6]])
>>> np.apply_along_axis(sorted, 0, b)
array([[4, 1, 6],
       [5, 2, 7],
       [8, 3, 9]])
>>> np.apply_along_axis(sorted, 1, b)
array([[1, 7, 8],
       [3, 4, 9],
       [2, 5, 6]])
```

For a function that returns a higher dimensional array, those dimensions are inserted in place of the axis dimension.

```
>>> np.apply_along_axis(np.diag, -1, b)
array([[ [8, 0, 0],
          [0, 1, 0],
          [0, 0, 7]],

         [[4, 0, 0],
          [0, 3, 0],
          [0, 0, 9]],

         [[5, 0, 0],
          [0, 2, 0],
```

### 4.3.2 Over axis

It is possible to apply a function repeatedly over multiple axes.

```
>>> a = np.arange(24).reshape(2,3,4)
>>> a
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])]
>>> np.apply_over_axes(np.sum, a, [0,2])
array([[[ 60],
        [ 92],
        [124]]])
>>> np.sum(a, axis=(0,2), keepdims=True)
array([[[ 60],
        [ 92],
        [124]]])
```

The method `apply_over_axis` is equal to `np.sum()` together with arguments `axis` and `keepdims=True`.

### 4.3.3 Vectorization

The `vectorize` function is provided primarily for convenience, not for performance. The implementation is essentially a for loop.

```
>>> def myfunc(a, b):
...     "Return a-b if a>b, otherwise return a+b"
...     if a > b:
...         return a - b
...     else:
...         return a + b
...
...
...
>>> vfunc = np.vectorize(myfunc)
>>> vfunc(range(1, 10), 5)
array([ 6,  7,  8,  9, 10,  1,  2,  3,  4])
```

The output type (*otype*) is determined by evaluating the first element of the input, unless it is specified:

```
>>> out = vfunc([1, 2, 3, 4], 2)
>>> type(out[0])
<class 'numpy.int32'>
>>> vfunc = np.vectorize(myfunc, otypes=[float])
>>> out = vfunc([1, 2, 3, 4], 2)
>>> type(out)
<class 'numpy.ndarray'>
>>> type(out[0])
<class 'numpy.float64'>
```

## 4.4 Statistical analysis

### 4.4.1 Quantile

Given a vector  $V$  of length  $n$ , the  $q$ -th quantile of  $V$  is the value  $q$  of the way from the minimum to the maximum in a sorted copy of  $V$ . The values and distances of the two nearest neighbors as well as the method parameter will determine the quantile if the normalized ranking does not match the location of  $q$  exactly. This function is the same as the median if  $q=0.5$ , the same as the minimum if  $q=0.0$  and the same as the maximum if  $q=1.0$ .

```
>>> np.quantile(a, 0.5, axis=0); np.quantile(a, 0.5, axis=1)
...
array([6.5, 4.5, 2.5])
array([7., 2.])
```

### 4.4.2 Percentile

Given a vector  $V$  of length  $n$ , the  $q$ -th percentile of  $V$  is the value  $q/100$  of the way from the minimum to the maximum in a sorted copy of  $V$ . The values and distances of the two nearest neighbors as well as the method parameter will determine the percentile if the normalized ranking does not match the location of  $q$  exactly. This function is the same as the median if  $q=50$ , the same as the minimum if  $q=0$  and the same as the maximum if  $q=100$ . Compute the  $q$ -th percentile of the data along the specified axis. Returns the  $q$ -th percentile(s) of the array elements.

```
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
...
>>> a
...
array([[10, 7, 4],
       [3, 2, 1]])
>>> np.percentile(a, 50, axis=0)
...
array([6.5, 4.5, 2.5])
>>> np.percentile(a, 50, axis=1)
...
array([7., 2.])
```

### 4.4.3 Averages and variances

The following code demonstrates as to how average can be computed for a one-dimensional array or vector.

```
>>> data = np.arange(1, 5)
>>> data
array([1, 2, 3, 4])
>>> np.average(data)
2.5
>>> np.average(np.arange(1, 11), weights=np.arange(10, 0, -1))
4.0
```

```
>>> data = np.arange(6).reshape((3, 2))
>>> np.average(data, axis=1, weights=[1./4, 3./4])
array([0.75, 2.75, 4.75])
```

## Arithmetic mean

The arithmetic mean is the sum of the elements along the axis divided by the number of elements. Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-precision accumulator using the dtype keyword can alleviate this issue. By default, float16 results are computed using float32 intermediates for extra precision.

```
>>> a = np.array([[5, 9, 13], [14, 10, 12], [11, 15, 19]])
...
>>> a.sum()/a.size
...
12.0
>>> a.mean()
...
12.0
```

The other example

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])
```

other statistics

```
>>> a
...
array([[ 5,  9, 13],
       [14, 10, 12],
       [11, 15, 19]])
>>> a.std()
...
3.7416573867739413
>>> np.std(a, axis=0)
...
array([3.74165739, 2.62466929, 3.09120617])
>>> np.std(a, axis=1)
...
```

```
array([3.26598632, 1.63299316, 3.26598632])
>>> a.var()
...
14.0
>>> np.var(a, axis=0)
...
array([14.          , 6.88888889, 9.55555556])
>>> np.var(a, axis=1)
...
array([10.66666667, 2.66666667, 10.66666667])
>>> np.median(a, axis=0)
...
array([11., 10., 13.])
>>> np.median(a, axis=1)
...
array([ 9., 12., 15.])
```

#### 4.4.4 Correlating

In this example we generate two random arrays, *xarr* and *yarr*, and compute the row-wise and column-wise Pearson correlation coefficients, R. Since *rowvar* is true by default, we first find the row-wise Pearson correlation coefficients between the variables of *xarr*.

```
>>> import numpy as np
>>> rng = np.random.default_rng(seed=42)
>>> rng
Generator(PCG64) at 0x24D740552A0
>>> xarr = rng.random((3, 3))
>>> xarr
array([[0.77395605, 0.43887844, 0.85859792],
       [0.69736803, 0.09417735, 0.97562235],
       [0.7611397 , 0.78606431, 0.12811363]])
>>> R1 = np.corrcoef(xarr)
>>> R1
array([[ 1.          , 0.99256089, -0.68080986],
       [ 0.99256089, 1.          , -0.76492172],
       [-0.68080986, -0.76492172, 1.        ]])
```

If we add another set of variables and observations *yarr*, we can compute the row-wise Pearson correlation coefficients between the variables in *xarr* and *yarr*.

```
>>> yarr = rng.random((3, 3))
```

```
>>> R2 = np.corrcoef(xarr, yarr)
>>> R2
array([[ 1.          ,  0.99256089, -0.68080986,  0.75008178, -0.
         ↪ 934284   ,
        -0.99004057],
       [ 0.99256089,  1.          , -0.76492172,  0.82502011, -0.
         ↪ 97074098,
        -0.99981569],
      [-0.68080986, -0.76492172,  1.          , -0.99507202,  0.
         ↪ 89721355,
        0.77714685],
      [ 0.75008178,  0.82502011, -0.99507202,  1.          , -0.
         ↪ 93657855,
        -0.83571711],
      [-0.934284   , -0.97074098,  0.89721355, -0.93657855,  1.
         ↪ ,
        0.97517215],
      [-0.99004057, -0.99981569,  0.77714685, -0.83571711,  0.
         ↪ 97517215,
        1.          ]])
```

Finally if we use the option `rowvar=False`, the columns are now being treated as the variables and we will find the column-wise Pearson correlation coefficients between variables in `xarr` and `yarr`.

```
>>> R3 = np.corrcoef(xarr, yarr, rowvar=False)
>>> R3
array([[ 1.          ,  0.77598074, -0.47458546, -0.75078643, -0.
         ↪ 9665554   ,
        0.22423734],
       [ 0.77598074,  1.          , -0.92346708, -0.99923895, -0.
         ↪ 58826587,
        -0.44069024],
      [-0.47458546, -0.92346708,  1.          ,  0.93773029,  0.
         ↪ 23297648,
        0.75137473],
      [-0.75078643, -0.99923895,  0.93773029,  1.          ,  0.
         ↪ 55627469,
        0.47536961],
      [-0.9665554   , -0.58826587,  0.23297648,  0.55627469,  1.
         ↪ ,
        -0.46666491],
      [ 0.22423734, -0.44069024,  0.75137473,  0.47536961, -0.
         ↪ 46666491,
        1.          ]])
```

#### 4.4.5 Covariance

```
>>> x = np.array([[0, 2], [1, 1], [2, 0]]).T
>>> x
array([[0, 1, 2],
       [2, 1, 0]])
>>> np.cov(x)
array([[ 1., -1.],
       [-1.,  1.]])
>>> x = [-2.1, -1, 4.3]
>>> y = [3, 1.1, 0.12]
>>> np.cov(x, y)
array([[11.71      , -4.286      ],
       [-4.286      ,  2.14413333]])
>>> X = np.stack((x, y), axis=0)
>>> X
array([[-2.1 , -1. ,  4.3 ],
       [ 3. ,  1.1 ,  0.12]])
>>> np.cov(X)
array([[11.71      , -4.286      ],
       [-4.286      ,  2.14413333]])
```

#### 4.4.6 Curve fitting

Curve fitting is the process of constructing a curve, or mathematical function, that has the best fit to a series of data points, possibly subject to constraints. Curve fitting can involve either interpolation, where an exact fit to the data is required, or smoothing, in which a “smooth” function is constructed that approximately fits the data. A related topic is regression analysis, which focuses more on questions of statistical inference such as how much uncertainty is present in a curve that is fit to data observed with random errors. Fitted curves can be used as an aid for data visualization, to infer values of a function where no data are available, and to summarize the relationships among two or more variables.

Construct the polynomial  $x^2 + 2x + 3$ :

```
>>> p = np.poly1d([1, 2, 3])
>>> p
poly1d([1, 2, 3])
>>> print(p)
      2
1 x + 2 x + 3
```

Evaluate the polynomial at  $x = 0.5$ :

```
>>> p(0.5)
4.25
```

Find the roots

```
>>> p.r
array([-1.+1.41421356j, -1.-1.41421356j])
```

Show the coefficients:

```
>>> p.c
array([1, 2, 3])
```

Display the order (the leading zero-coefficients are removed):

### Polynomial interpolation

In numerical analysis, polynomial interpolation is the interpolation of a given data set by the polynomial of lowest possible degree that passes through the points of the dataset.

```
>>> x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
>>> z
array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254])
```

Above code demonstrates as to how curves  $x, y$  fits to a plane called  $0.08703704x^3 - 0.81349206x^2 + 1.69312169x - 0.03968254$

```
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349231
```

0.6143849206349231 is the value for  $x = 0.5$  for above polynomial.

## Notes

<sup>33</sup>For variety of *operators* refer to [https://www.tutorialspoint.com/python/python\\_basic\\_operators.htm](https://www.tutorialspoint.com/python/python_basic_operators.htm). There is a module for operators known as *operator*. Visit <https://docs.python.org/2/library/operator.html> for documentation.

<sup>34</sup>Read more about *string* module at <https://docs.python.org/2/library/string.html>

<sup>35</sup>You may visit <https://docs.python.org/3/tutorial/datastructures.html> for more details.

<sup>36</sup>Retrieved from Python Software Foundation. Visit <https://docs.python.org/3/tutorial/datastructures.html> for more information on *dictionary*

<sup>37</sup>This PEP is driven by the desire to have a simpler way to format strings in Python. Read about this PEP at <https://www.python.org/dev/peps/pep-0498/#rationale>

<sup>38</sup>Nørmark, Kurt. Overview of the four main programming paradigms. Aalborg University, 9 May 2011. Retrieved 22 September 2012. Available at [http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms\\_the\\_mes-paradigm-overview-section.html](http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_the_mes-paradigm-overview-section.html)

<sup>39</sup>Read more from <https://www.programiz.com/python-programming/function>

<sup>40</sup>Read more about *identifiers* at <https://www.programiz.com/python-programming/keywords-identifier#rules>

<sup>41</sup>Retrieved from <https://www.pythont-course.eu/lambda.php>

<sup>42</sup>Read about *arguments* at <https://docs.python.org/3/tutorial/controlflow.html#positional-only-parameters>

<sup>43</sup>Retrieved from <https://docs.python.org/3/tutorial/datastructures.html#nested-list-comprehensions>

<sup>44</sup>Zero matrix is one of the very influential matrices often used while processing data in data sciences. In statistics, zero matrices are used in regression methods.

<sup>45</sup>Symmetric matrix is different from square matrix. A symmetric matrix is a matrix which satisfied the property  $A = A^T$ , where  $A^T$  is known as *Transpose A* or *A Transpose*

<sup>46</sup>A zero matrix or null matrix is a matrix all of whose entries are zero. There are number of applications in mathematics using zeros matrix. For instance, *undecidable problem* in decision science, *annihilator matrix* in OLS regression are only few to name with.

<sup>47</sup>Read more about identity matrices at [https://en.wikipedia.org/wiki/Invertible\\_matrix](https://en.wikipedia.org/wiki/Invertible_matrix)

<sup>48</sup>

<sup>49</sup>Arthur Cayley FRS was a prolific British mathematician who worked mostly on algebra. He helped found the modern British school of pure mathematics. Cayley's treatise on geometric transformations using matrices is one of the most renowned classics in geometry. Cayley studies and explains properties of matrices using very simple arithmetic calculations such as addition, subtraction, multiplication and division.

<sup>50</sup>Read more about *csr\_data()* at [https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.sparse.csr\\_matrix.html](https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.sparse.csr_matrix.html)

<sup>51</sup>Visit <https://stackoverflow.com/questions/448271/what-is-init-py-for> for more details.

<sup>52</sup>The other better place to learn

about `__init__.py` is, perhaps, <https://docs.python.org/3.3/tutorial/modules.html>. This is Python official documentation.

## Tasks

1. Visit [https://numpy.org/doc/1.24/user/absolute\\_beginners.html](https://numpy.org/doc/1.24/user/absolute_beginners.html). Read the material and answer the following.
    - (a) How to convert a 1D array into a 2D array (how to add a new axis to an array)?
    - (b) How to create an array from existing data ?
    - (c) What are basic array operations?
    - (d) What is broadcasting? Demonstrate.
    - (e) What are more useful array operations?
    - (f) How do you create matrices using NumPy?
    - (g) How do you generate random numbers?
    - (h) How to get unique items and counts?
    - (i) Write about Transposing and reshaping a matrix.
    - (j) How do you reverse an array?
  - (k) How do you reshape and flatten multidimensional arrays?
  - (l) What are docstrings? How to access the docstring for more information?
  - (m) How do you work with mathematical formulas?
  - (n) How to save and load NumPy objects?
  - (o) Explain Importing and exporting a CSV.
  - (p) How do you plot arrays with Matplotlib?
2. Visit <https://www.oreilly.com/library/view/python-for-data/9781449323592/ch04.html>
    1. Read the documentation related to (1) correlation matrix, and (2) random number generation random walks.
  3. Visit <https://numpy.org/doc/stable/reference/routines.statistics.html>. Read the documentation related various statistical techniques.



# Chapter 5

## SciPy

SciPy is a collection of mathematical algorithms and convenience functions built on the NumPy extension of Python. It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data. With SciPy, an interactive Python session becomes a data-processing and system-prototyping environment rivaling systems, such as MATLAB, IDL, Octave, R-Lab, and SciLab.

The additional benefit of basing SciPy on Python is that this also makes a powerful programming language available for use in developing sophisticated programs and specialized applications. Scientific applications using SciPy benefit from the development of additional modules in numerous niches of the software landscape by developers across the world. Everything from parallel programming to web and data-base subroutines and classes have been made available to the Python programmer. All of this power is available in addition to the mathematical libraries in SciPy.

This tutorial will acquaint the first-time user of SciPy with some of its most important features. It assumes that the user has already installed the SciPy package. Some general Python facility is also assumed, such as could be acquired by working through the Python distribution's

Tutorial. For further introductory help the user is directed to the NumPy documentation.

## 5.1 SciPy Organization

SciPy is organized into subpackages covering different scientific computing domains. These are summarized in the following table:

Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions

Table 5.1: SciPy organization

SciPy sub-packages need to be imported separately, for example:

```
from scipy import linalg, optimize
```

Because of their ubiquitousness, some of the functions in these sub-packages are also made available in the `scipy` namespace to ease their use in interactive sessions and programs. In addition, many basic array functions from `numpy` are also available at the top-level of the `scipy` package. Before looking at the sub-packages individually, we will first look at some of these common functions.

## 5.2 Random variables

There are two general distribution classes that have been implemented for encapsulating continuous random variables and discrete random variables. Over 80 continuous random variables (RVs) and 10 discrete random variables have been implemented using these classes. Besides this, new routines and distributions can be easily added by the end user. (If you create one, please contribute it.)

All of the statistics functions are located in the sub-package `scipy.stats` and a fairly complete listing of these functions can be obtained using `info(stats)`. The list of the random variables available can also be obtained from the docstring for the `stats` sub-package.

In the discussion below, we mostly focus on continuous RVs. Nearly everything also applies to discrete variables, but we point out some differences here: Specific points for discrete distributions.

In the code samples below, we assume that the `scipy.stats` package is imported as

```
from scipy import stats
```

and in some cases we assume that individual objects are imported as

```
from scipy.stats import norm
```

### 5.2.1 Common methods

The main public methods for continuous RVs are:

- `rvs`: Random Variates
- `pdf`: Probability Density Function
- `cdf`: Cumulative Distribution Function
- `sf`: Survival Function (1-CDF)
- `ppf`: Percent Point Function (Inverse of CDF)
- `isf`: Inverse Survival Function (Inverse of SF)
- `stats`: Return mean, variance, (Fisher's) skew, or (Fisher's) kurtosis
- `moment`: non-central moments of the distribution

Let's take a normal RV as an example.

```
norm.cdf(0)  
0.5
```

To compute the cdf at a number of points, we can pass a list or a numpy array.

```
norm.cdf([-1., 0, 1])
array([ 0.15865525,  0.5,   0.84134475])
import numpy as np
norm.cdf(np.array([-1., 0, 1]))
array([ 0.15865525,  0.5,   0.84134475])
```

Thus, the basic methods, such as pdf, cdf, and so on, are vectorized. Other generally useful methods are supported too:

```
norm.mean(), norm.std(), norm.var()
(0.0, 1.0, 1.0)
norm.stats(moments="mv")
(array(0.0), array(1.0))
```

To find the median of a distribution, we can use the percent point function ppf, which is the inverse of the cdf:

```
norm.ppf(0.5)
0.0
```

To generate a sequence of random variates, use the size keyword argument:

```
norm.rvs(size=3)
array([-0.35687759,  1.34347647, -0.11710531])  # random
```

Don't think that `norm.rvs(5)` generates 5 variates:

```
norm.rvs(5)
5.471435163732493  # random
```

Here, 5 with no keyword is being interpreted as the first possible keyword argument, loc, which is the first of a pair of keyword arguments taken by all continuous distributions. This brings us to the topic of the next subsection.

### 5.2.2 Random number generation

Drawing random numbers relies on generators from `numpy.random` package. In the examples above, the specific stream of random numbers is not reproducible across runs. To achieve reproducibility, you can explicitly seed a random number generator. In NumPy, a generator is an instance of `numpy.random.Generator`. Here is the canonical way to create a generator:

```
from numpy.random import default_rng
rng = default_rng()
```

And fixing the seed can be done like this:

```
# do NOT copy this value
rng = default_rng(301439351238479871608357552876690613766)
Warning
```

Do not use this number or common values such as 0. Using just a small set of seeds to instantiate larger state spaces means that there are some initial states that are impossible to reach. This creates some biases if everyone uses such values. A good way to get a seed is to use a `numpy.random.SeedSequence`:

```
from numpy.random import SeedSequence
print(SeedSequence().entropy)
301439351238479871608357552876690613766 # random
```

The `random_state` parameter in distributions accepts an instance of `numpy.random.Generator` class, or an integer, which is then used to seed an internal Generator object:

```
norm.rvs(size=5, random_state=rng)
array([ 0.47143516, -1.19097569,  1.43270697, -0.3126519 , -0.
       ↪ 72058873]) # random
```

For further info, see NumPy's documentation.

To learn more about the random number samplers implemented in SciPy, see non-uniform random number sampling tutorial and quasi monte carlo tutorial

## Shifting and scaling

All continuous distributions take loc and scale as keyword parameters to adjust the location and scale of the distribution, e.g., for the standard normal distribution, the location is the mean and the scale is the standard deviation.

```
norm.stats(loc=3, scale=4, moments="mv")
(array(3.0), array(16.0))
```

In many cases, the standardized distribution for a random variable X is obtained through the transformation  $(X - \text{loc})/\text{scale}$ . The default values are  $\text{loc} = 0$  and  $\text{scale} = 1$ .

Smart use of loc and scale can help modify the standard distributions in many ways. To illustrate the scaling further, the cdf of an exponentially distributed RV with mean is given by

By applying the scaling rule above, it can be seen that by taking `scale = 1./lambda` we get the proper scale.

```
from scipy.stats import expon
expon.mean(scale=3.)
3.0
```

Distributions that take shape parameters may require more than simple application of loc and/or scale to achieve the desired form. For example, the distribution of 2-D vector lengths given a constant vector of length perturbed by independent  $N(0, 1)$  deviations in each component is `rice(, scale= )`. The first argument is a shape parameter that needs to be scaled along with .

The uniform distribution is also interesting:

```
from scipy.stats import uniform
uniform.cdf([0, 1, 2, 3, 4, 5], loc=1, scale=4)
array([ 0. ,  0. ,  0.25,  0.5 ,  0.75,  1. ])
```

Finally, recall from the previous paragraph that we are left with the problem of the meaning of `norm.rvs(5)`. As it turns out, calling a distribution like this, the first argument, i.e., the 5, gets passed to set the loc parameter. Let's see:

```
np.mean(norm.rvs(5, size=500))
5.0098355106969992 # random
```

Thus, to explain the output of the example of the last section: `norm.rvs(5)` generates a single normally distributed random variate with mean `loc=5`, because of the default `size=1`.

We recommend that you set `loc` and `scale` parameters explicitly, by passing the values as keywords rather than as arguments. Repetition can be minimized when calling more than one method of a given RV by using the technique of Freezing a Distribution, as explained below.

### 5.2.3 Shape parameters

While a general continuous random variable can be shifted and scaled with the `loc` and `scale` parameters, some distributions require additional shape parameters. For instance, the gamma distribution with density requires the shape parameter `a`. Observe that setting `a` can be obtained by setting the `scale` keyword to 1. Let's check the number and name of the shape parameters of the gamma distribution. (We know from the above that this should be 1.)

```
from scipy.stats import gamma
gamma.numargs
1
gamma.shapes
'a'
```

Now, we set the value of the shape variable to 1 to obtain the exponential distribution, so that we compare easily whether we get the results we expect.

```
gamma(1, scale=2.).stats(moments="mv")
(array(2.0), array(4.0))
```

Notice that we can also specify shape parameters as keywords:

```
gamma(a=1, scale=2.).stats(moments="mv")
(array(2.0), array(4.0))
```

### 5.2.4 Freezing a distribution

Passing the loc and scale keywords time and again can become quite bothersome. The concept of freezing a RV is used to solve such problems.

```
rv = gamma(1, scale=2.)
```

By using `rv` we no longer have to include the scale or the shape parameters anymore. Thus, distributions can be used in one of two ways, either by passing all distribution parameters to each method call (such as we did earlier) or by freezing the parameters for the instance of the distribution. Let us check this:

```
rv.mean(), rv.std()
(2.0, 2.0)
```

This is, indeed, what we should get.

### 5.2.5 Broadcasting

The basic methods `pdf`, and so on, satisfy the usual numpy broadcasting rules. For example, we can calculate the critical values for the upper tail of the t distribution for different probabilities and degrees of freedom.

```
stats.t.isf([0.1, 0.05, 0.01], [[10], [11]])
array([[ 1.37218364,  1.81246112,  2.76376946],
       [ 1.36343032,  1.79588482,  2.71807918]])
```

Here, the first row contains the critical values for 10 degrees of freedom and the second row for 11 degrees of freedom (d.o.f.). Thus, the broadcasting rules give the same result of calling `isf` twice:

```
stats.t.isf([0.1, 0.05, 0.01], 10)
array([ 1.37218364,  1.81246112,  2.76376946])
stats.t.isf([0.1, 0.05, 0.01], 11)
array([ 1.36343032,  1.79588482,  2.71807918])
```

If the array with probabilities, i.e.,  $[0.1, 0.05, 0.01]$  and the array of degrees of freedom i.e.,  $[10, 11, 12]$ , have the same array shape, then

element-wise matching is used. As an example, we can obtain the 10% tail for 10 d.o.f., the 5% tail for 11 d.o.f. and the 1% tail for 12 d.o.f. by calling

```
stats.t.isf([0.1, 0.05, 0.01], [10, 11, 12])
array([ 1.37218364,  1.79588482,  2.68099799])
```

### 5.2.6 Specific points for discrete distributions

Discrete distributions have mostly the same basic methods as the continuous distributions. However pdf is replaced by the probability mass function pmf, no estimation methods, such as fit, are available, and scale is not a valid keyword parameter. The location parameter, keyword loc, can still be used to shift the distribution.

The computation of the cdf requires some extra attention. In the case of continuous distribution, the cumulative distribution function is, in most standard cases, strictly monotonic increasing in the bounds (a,b) and has, therefore, a unique inverse. The cdf of a discrete distribution, however, is a step function, hence the inverse cdf, i.e., the percent point function, requires a different definition:

```
ppf(q) = min{x : cdf(x) >= q, x integer}
```

We can look at the hypergeometric distribution as an example

```
from scipy.stats import hypergeom
[M, n, N] = [20, 7, 12]
```

If we use the cdf at some integer points and then evaluate the ppf at those cdf values, we get the initial integers back, for example

```
x = np.arange(4) * 2
x
array([0, 2, 4, 6])
prb = hypergeom.cdf(x, M, n, N)
prb
array([ 1.03199174e-04,   5.21155831e-02,   6.08359133e-01,
       9.89783282e-01])
hypergeom.ppf(prb, M, n, N)
array([ 0.,  2.,  4.,  6.])
```

If we use values that are not at the kinks of the cdf step function, we get the next higher integer back:

```
hypergeom.ppf(prb + 1e-8, M, n, N)
array([ 1.,  3.,  5.,  7.])
hypergeom.ppf(prb - 1e-8, M, n, N)
array([ 0.,  2.,  4.,  6.])
```

## 5.3 Building specific distributions

The next examples shows how to build your own distributions. Further examples show the usage of the distributions and some statistical tests.

### 5.3.1 Making a continuous distribution, i.e., sub-classing *rv\_continuous*

Making continuous distributions is fairly simple.

```
from scipy import stats
class deterministic_gen(stats.rv_continuous):
    def _cdf(self, x):
        return np.where(x < 0, 0., 1.)
    def _stats(self):
        return 0., 0., 0., 0.
deterministic = deterministic_gen(name="deterministic")
deterministic.cdf(np.arange(-3, 3, 0.5))
array([ 0.,  0.,  0.,  0.,  0.,  1.,  1.,  1.,  1.,
       ↗ 1.])
```

Interestingly, the pdf is now computed automatically:

```
deterministic.pdf(np.arange(-3, 3, 0.5))
array([ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
       0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
       5.83333333e+04,   4.16333634e-12,   4.16333634e-12,
      4.16333634e-12,   4.16333634e-12,   4.16333634e-12])
```

Be aware of the performance issues mentioned in Performance issues and cautionary remarks. The computation of unspecified common methods can become very slow, since only general methods are called, which, by their very nature, cannot use any specific information about the distribution. Thus, as a cautionary example:

```
from scipy.integrate import quad
quad(deterministic.pdf, -1e-1, 1e-1)
(4.163336342344337e-13, 0.0)
```

But this is not correct: the integral over this pdf should be 1. Let's make the integration interval smaller:

```
quad(deterministic.pdf, -1e-3, 1e-3) # warning removed
(1.000076872229173, 0.0010625571718182458)
```

This looks better. However, the problem originated from the fact that the pdf is not specified in the class definition of the deterministic distribution.

### 5.3.2 Subclassing rv\_discrete

In the following, we use `stats.rv_discrete` to generate a discrete distribution that has the probabilities of the truncated normal for the intervals centered around the integers.

#### General info

From the docstring of `rv_discrete`, `help(stats.rv_discrete)`,

“You can construct an arbitrary discrete rv where  $PX=x_k = p_k$  by passing to the `rv_discrete` initialization method (through the `values=` keyword) a tuple of sequences ( $x_k$ ,  $p_k$ ) which describes only those values of  $X$  ( $x_k$ ) that occur with nonzero probability ( $p_k$ ).”

Next to this, there are some further requirements for this approach to work:

- The keyword name is required.
- The support points of the distribution  $x_k$  have to be integers.
- The number of significant digits (decimals) needs to be specified.

In fact, if the last two requirements are not satisfied, an exception may be raised or the resulting numbers may be incorrect.

## An example

Let's do the work. First:

```
npoints = 20    # number of integer support points of the
                 ↪ distribution minus 1
npointsh = npoints // 2
npointsf = float(npoints)
nbound = 4      # bounds for the truncated normal
normbound = (1+1/npointsf) * nbound   # actual bounds of
                                         ↪ truncated normal
grid = np.arange(-npointsh, npointsh+2, 1)  # integer grid
gridlimitsnorm = (grid-0.5) / npointsh * nbound  # bin limits
                                         ↪ for the truncnorm
gridlimits = grid - 0.5    # used later in the analysis
grid = grid[:-1]
probs = np.diff(stats.truncnorm.cdf(gridlimitsnorm, -normbound,
                                         ↪ normbound))
gridint = grid
```

And, finally, we can subclass `rv_discrete`:

```
normdiscrete = stats.rv_discrete(values=(gridint,
                                         np.round(probs, decimals=7)), name='normdiscrete')
```

Now that we have defined the distribution, we have access to all common methods of discrete distributions.

```
print('mean = %6.4f, variance = %6.4f, skew = %6.4f, kurtosis = '
      ↪ '%6.4f' %
      normdiscrete.stats(moments='mvsk'))
mean = -0.0000, variance = 6.3302, skew = 0.0000, kurtosis = -0
                                         ↪ .0076
nd_std = np.sqrt(normdiscrete.stats(moments='v'))
```

## Testing the implementation

Let's generate a random sample and compare observed frequencies with the probabilities.

```
n_sample = 500
rvs = normdiscrete.rvs(size=n_sample)
f, l = np.histogram(rvs, bins=gridlimits)
sfreq = np.vstack([gridint, f, probs*n_sample]).T
print(sfreq)
```

```
[[-1.0000000e+01  0.0000000e+00  2.95019349e-02] # random
 [-9.0000000e+00  0.0000000e+00  1.32294142e-01]
 [-8.0000000e+00  0.0000000e+00  5.06497902e-01]
 [-7.0000000e+00  2.0000000e+00  1.65568919e+00]
 [-6.0000000e+00  1.0000000e+00  4.62125309e+00]
 [-5.0000000e+00  9.0000000e+00  1.10137298e+01]
 [-4.0000000e+00  2.6000000e+01  2.24137683e+01]
 [-3.0000000e+00  3.7000000e+01  3.89503370e+01]
 [-2.0000000e+00  5.1000000e+01  5.78004747e+01]
 [-1.0000000e+00  7.1000000e+01  7.32455414e+01]
 [ 0.0000000e+00  7.4000000e+01  7.92618251e+01]
 [ 1.0000000e+00  8.9000000e+01  7.32455414e+01]
 [ 2.0000000e+00  5.5000000e+01  5.78004747e+01]
 [ 3.0000000e+00  5.0000000e+01  3.89503370e+01]
 [ 4.0000000e+00  1.7000000e+01  2.24137683e+01]
 [ 5.0000000e+00  1.1000000e+01  1.10137298e+01]
 [ 6.0000000e+00  4.0000000e+00  4.62125309e+00]
 [ 7.0000000e+00  3.0000000e+00  1.65568919e+00]
 [ 8.0000000e+00  0.0000000e+00  5.06497902e-01]
 [ 9.0000000e+00  0.0000000e+00  1.32294142e-01]
 [ 1.0000000e+01  0.0000000e+00  2.95019349e-02]]
```

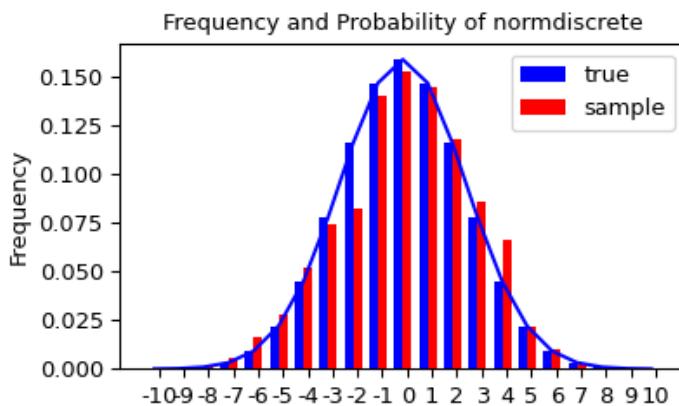


Figure 5.1: Normal distribution plot 1

Next, we can test whether our sample was generated by our norm-

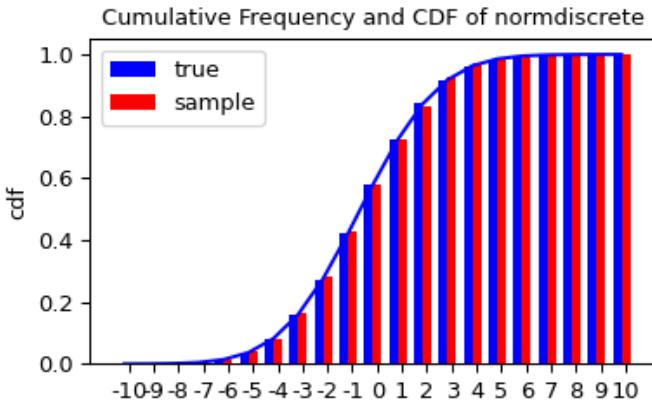


Figure 5.2: Normal distribution plot 2

discrete distribution. This also verifies whether the random numbers were generated correctly.

The chisquare test requires that there are a minimum number of observations in each bin. We combine the tail bins into larger bins so that they contain enough observations.

```
f2 = np.hstack([f[:5].sum(), f[5:-5], f[-5:].sum()])
p2 = np.hstack([probs[:5].sum(), probs[5:-5], probs[-5:].sum()])
ch2, pval = stats.chisquare(f2, p2*n_sample)
print('chisquare for normdiscrete: chi2 = %6.3f pvalue = %6.4f'
      % (ch2, pval))
chisquare for normdiscrete: chi2 = 12.466 pvalue = 0.4090 #random
```

The pvalue in this case is high, so we can be quite confident that our random sample was actually generated by the distribution.

## 5.4 Analysing one sample

First, we create some random variables. We set a seed so that in each run we get identical results to look at. As an example we take a sample from the Student t distribution:

```
x = stats.t.rvs(10, size=1000)
```

Here, we set the required shape parameter of the t distribution, which in statistics corresponds to the degrees of freedom, to 10. Using size=1000 means that our sample consists of 1000 independently drawn (pseudo) random numbers. Since we did not specify the keyword arguments loc and scale, those are set to their default values zero and one.

### 5.4.1 Descriptive statistics

*x* is a numpy array, and we have direct access to all array methods, e.g.,

```
print(x.min())      # equivalent to np.min(x)
-3.78975572422   # random
print(x.max())      # equivalent to np.max(x)
5.26327732981    # random
print(x.mean())     # equivalent to np.mean(x)
0.0140610663985   # random
print(x.var())      # equivalent to np.var(x))
1.28899386208    # random
```

How do the sample properties compare to their theoretical counterparts?

```
m, v, s, k = stats.t.stats(10, moments='mvsk')
n, (smin, smax), sm, sv, ss, sk = stats.describe(x)
sstr = '%-14s mean = %6.4f, variance = %6.4f, skew = %6.4f,
        ↪ kurtosis = %6.4f'
print(sstr % ('distribution:', m, v, s ,k))
distribution: mean = 0.0000, variance = 1.2500, skew = 0.0000,
               ↪ kurtosis = 1.0000 # random
print(sstr % ('sample:', sm, sv, ss, sk))
sample:       mean = 0.0141, variance = 1.2903, skew = 0.2165,
               ↪ kurtosis = 1.0556 # random
```

`stats.describe` uses the unbiased estimator for the variance, while `np.var` is the biased estimator. For our sample the sample statistics differ a bit by a small amount from their theoretical counterparts.

### 5.4.2 T-test and KS-test

We can use the t-test to test whether the mean of our sample differs in a statistically significant way from the theoretical expectation.

```
print('t-statistic = %6.3f pvalue = %6.4f' % stats.ttest_1samp
      ↪ (x, m))
t-statistic = 0.391 pvalue = 0.6955 # random
```

The pvalue is 0.7, this means that with an alpha error of, for example, 10%, we cannot reject the hypothesis that the sample mean is equal to zero, the expectation of the standard t-distribution. As an exercise, we can calculate our ttest also directly without using the provided function, which should give us the same answer, and so it does:

```
tt = (sm-m)/np.sqrt(sv/float(n)) # t-statistic for mean
pval = stats.t.sf(np.abs(tt), n-1)*2 # two-sided pvalue = Prob
      ↪ (abs(t)>tt)
print('t-statistic = %6.3f pvalue = %6.4f' % (tt, pval))
t-statistic = 0.391 pvalue = 0.6955 # random
```

The Kolmogorov-Smirnov test can be used to test the hypothesis that the sample comes from the standard t-distribution

```
print('KS-statistic D = %6.3f pvalue = %6.4f' % stats.kstest(x,
      ↪ 't', (10,)))
KS-statistic D = 0.016 pvalue = 0.9571 # random
```

Again, the p-value is high enough that we cannot reject the hypothesis that the random sample really is distributed according to the t-distribution. In real applications, we don't know what the underlying distribution is. If we perform the Kolmogorov-Smirnov test of our sample against the standard normal distribution, then we also cannot reject the hypothesis that our sample was generated by the normal distribution given that, in this example, the p-value is almost 40

```
print('KS-statistic D = %6.3f pvalue = %6.4f' % stats.kstest(x,
      ↪ 'norm'))
```

```
KS-statistic D = 0.028 pvalue = 0.3918 # random
```

However, the standard normal distribution has a variance of 1, while our sample has a variance of 1.29. If we standardize our sample and test it against the normal distribution, then the p-value is again large enough that we cannot reject the hypothesis that the sample came from the normal distribution.

```
d, pval = stats.kstest((x-x.mean())/x.std(), 'norm')
print('KS-statistic D = %6.3f pvalue = %6.4f' % (d, pval))
KS-statistic D = 0.032 pvalue = 0.2397 # random
```

The Kolmogorov-Smirnov test assumes that we test against a distribution with given parameters, since, in the last case, we estimated mean and variance, this assumption is violated and the distribution of the test statistic, on which the p-value is based, is not correct.

### 5.4.3 Tails of the distribution

Finally, we can check the upper tail of the distribution. We can use the percent point function ppf, which is the inverse of the cdf function, to obtain the critical values, or, more directly, we can use the inverse of the survival function

```
crit01, crit05, crit10 = stats.t.ppf([1-0.01, 1-0.05, 1-0.10],
                                     ↪ 10)
print('critical values from ppf at 1%, 5% and 10%' %8.4f %8.
      ↪ 4f %8.4f' % (crit01, crit05,
      ↪ crit10))
critical values from ppf at 1%, 5% and 10%    2.7638    1.8125
      ↪ 1.3722
print('critical values from isf at 1%, 5% and 10%' %8.4f %8.
      ↪ 4f %8.4f' % tuple(stats.t.
      ↪ isf([0.01,0.05,0.10],10)))
critical values from isf at 1%, 5% and 10%    2.7638    1.8125
      ↪ 1.3722
freq01 = np.sum(x>crit01) / float(n) * 100
freq05 = np.sum(x>crit05) / float(n) * 100
freq10 = np.sum(x>crit10) / float(n) * 100
print('sample %-frequency at 1%, 5% and 10%' tail %8.4f %8.
      ↪ 4f %8.4f' % (freq01, freq05,
      ↪ freq10))
```

```
sample %-frequency at 1%, 5% and 10% tail    1.4000   5.8000  10
      ↪ .5000 # random
```

In all three cases, our sample has more weight in the top tail than the underlying distribution. We can briefly check a larger sample to see if we get a closer match. In this case, the empirical frequency is quite close to the theoretical probability, but if we repeat this several times, the fluctuations are still pretty large.

```
freq051 = np.sum(stats.t.rvs(10, size=10000) > crit05) / 10000.
          ↪ 0 * 100
print('larger sample %-frequency at 5% tail %8.4f' % freq051)
larger sample %-frequency at 5% tail    4.8000 # random
```

We can also compare it with the tail of the normal distribution, which has less weight in the tails:

```
print('tail prob. of normal at 1%%, 5%% and 10%% %8.4f %8.4f %8
      ↪ .4f' %
      tuple(stats.norm.sf([crit01, crit05, crit10])*100))
tail prob. of normal at 1%, 5% and 10%    0.2857   3.4957   8.
      ↪ 5003
```

The chisquare test can be used to test whether for a finite number of bins, the observed frequencies differ significantly from the probabilities of the hypothesized distribution.

```
quantiles = [0.0, 0.01, 0.05, 0.1, 1-0.10, 1-0.05, 1-0.01, 1.0]
crit = stats.t.ppf(quantiles, 10)
crit
array([-inf, -2.76376946, -1.81246112, -1.37218364,  1.
       ↪ 37218364,
       1.81246112,  2.76376946,         inf])
n_sample = x.size
freqcount = np.histogram(x, bins=crit)[0]
tprob = np.diff(quantiles)
nprob = np.diff(stats.norm.cdf(crit))
tch, tpval = stats.chisquare(freqcount, tprob*n_sample)
nch, npval = stats.chisquare(freqcount, nprob*n_sample)
print('chisquare for t:      chi2 = %6.2f pvalue = %6.4f' % (
      ↪ tch, tpval))
chisquare for t:      chi2 =  2.30 pvalue = 0.8901 # random
print('chisquare for normal: chi2 = %6.2f pvalue = %6.4f' % (
      ↪ nch, npval))
```

```
chisquare for normal: chi2 = 64.60 pvalue = 0.0000 # random
```

We see that the standard normal distribution is clearly rejected, while the standard t-distribution cannot be rejected. Since the variance of our sample differs from both standard distributions, we can again redo the test taking the estimate for scale and location into account.

The fit method of the distributions can be used to estimate the parameters of the distribution, and the test is repeated using probabilities of the estimated distribution.

```
tddf, tloc, tscale = stats.t.fit(x)
nloc, nscale = stats.norm.fit(x)
tprob = np.diff(stats.t.cdf(crit, tddf, loc=tloc, scale=tscale))
nprob = np.diff(stats.norm.cdf(crit, loc=nloc, scale=nscale))
tch, tpval = stats.chisquare(freqcount, tprob*n_sample)
nch, npval = stats.chisquare(freqcount, nprob*n_sample)
print('chisquare for t:      chi2 = %6.2f pvalue = %6.4f' %
      ↪ tch, tpval)
chisquare for t:      chi2 = 1.58 pvalue = 0.9542 # random
print('chisquare for normal: chi2 = %6.2f pvalue = %6.4f' %
      ↪ nch, npval))
chisquare for normal: chi2 = 11.08 pvalue = 0.0858 # random
```

Taking account of the estimated parameters, we can still reject the hypothesis that our sample came from a normal distribution (at the 5

#### 5.4.4 Special tests for normal distributions

Since the normal distribution is the most common distribution in statistics, there are several additional functions available to test whether a sample could have been drawn from a normal distribution.

First, we can test if skew and kurtosis of our sample differ significantly from those of a normal distribution:

```
print('normal skewtest teststat = %6.3f pvalue = %6.4f' % stats
      ↪ .skewtest(x))
normal skewtest teststat = 2.785 pvalue = 0.0054 # random
print('normal kurtosistest teststat = %6.3f pvalue = %6.4f' %
      ↪ stats.kurtosistest(x))
normal kurtosistest teststat = 4.757 pvalue = 0.0000 # random
```

These two tests are combined in the normality test

```
print('normaltest teststat = %6.3f pvalue = %6.4f' % stats.
      ↪ normaltest(x))
normaltest teststat = 30.379 pvalue = 0.0000 # random
```

In all three tests, the p-values are very low and we can reject the hypothesis that the our sample has skew and kurtosis of the normal distribution. Since skew and kurtosis of our sample are based on central moments, we get exactly the same results if we test the standardized sample:

```
print('normaltest teststat = %6.3f pvalue = %6.4f' %
      stats.normaltest((x-x.mean())/x.std()))
normaltest teststat = 30.379 pvalue = 0.0000 # random
```

Because normality is rejected so strongly, we can check whether the normaltest gives reasonable results for other cases:

```
print('normaltest teststat = %6.3f pvalue = %6.4f' %
      stats.normaltest(stats.t.rvs(10, size=100)))
normaltest teststat = 4.698 pvalue = 0.0955 # random
print('normaltest teststat = %6.3f pvalue = %6.4f' %
      stats.normaltest(stats.norm.rvs(size=1000)))
normaltest teststat = 0.613 pvalue = 0.7361 # random
```

When testing for normality of a small sample of t-distributed observations and a large sample of normal-distributed observations, then in neither case can we reject the null hypothesis that the sample comes from a normal distribution. In the first case, this is because the test is not powerful enough to distinguish a t and a normally distributed random variable in a small sample.

## 5.5 Comparing two samples

In the following, we are given two samples, which can come either from the same or from different distribution, and we want to test whether these samples have the same statistical properties.

### 5.5.1 Comparing means

Test with sample with identical means:

```
rvs1 = stats.norm.rvs(loc=5, scale=10, size=500)
rvs2 = stats.norm.rvs(loc=5, scale=10, size=500)
stats.ttest_ind(rvs1, rvs2)
Ttest_indResult(statistic=-0.5489036175088705, pvalue=0.
                 ↪ 5831943748663959) # random
```

Test with sample with different means:

```
rvs3 = stats.norm.rvs(loc=8, scale=10, size=500)
stats.ttest_ind(rvs1, rvs3)
Ttest_indResult(statistic=-4.533414290175026, pvalue=6.
                 ↪ 507128186389019e-06) # random
```

### 5.5.2 Kolmogorov-Smirnov test for two samples

For the example, where both samples are drawn from the same distribution, we cannot reject the null hypothesis, since the pvalue is high

```
stats.ks_2samp(rvs1, rvs2)
KstestResult(statistic=0.026, pvalue=0.9959527565364388) #
                 ↪ random
```

In the second example, with different location, i.e., means, we can reject the null hypothesis, since the pvalue is below 1

```
stats.ks_2samp(rvs1, rvs3)
KstestResult(statistic=0.114, pvalue=0.00299005061044668) #
                 ↪ random
```

## Tasks

1. Visit <https://docs.scipy.org/doc/scipy/>. Read the material and answer the following.  
(a)

# Chapter 6

## Data analysis

Data analysis is a process of inspecting, cleansing, transforming, and modeling data with the goal of discovering useful information, informing conclusions, and supporting decision-making.<sup>53</sup> Data analysis has multiple facets and approaches, encompassing diverse techniques under a variety of names, and is used in different business, science, and social science domains.<sup>54</sup> In today's business world, data analysis plays a role in making decisions more scientific and helping businesses operate more effectively.<sup>55</sup>

Data mining is a particular data analysis technique that focuses on statistical modelling and knowledge discovery for predictive rather than purely descriptive purposes, while business intelligence covers data analysis that relies heavily on aggregation, focusing mainly on business information. In statistical applications, data analysis can be divided into descriptive statistics, exploratory data analysis (EDA), and confirmatory data analysis (CDA).<sup>56</sup> EDA focuses on discovering new features in the data while CDA focuses on confirming or falsifying existing hypotheses. Predictive analytics focuses on the application of statistical models for predictive forecasting or classification, while text analytics applies statistical, linguistic, and structural techniques to extract and classify information from textual sources, a species of unstructured data. All of the above are varieties of data analysis.<sup>57</sup>

## 6.1 Univariate analysis

Univariate is a term commonly used in statistics to describe a type of data which consists of observations on only a single characteristic or attribute. A simple example of univariate data would be the salaries of workers in industry.<sup>58</sup> Like all the other data, univariate data can be visualized using graphs, images or other analysis tools after the data is measured, collected, reported, and analyzed.<sup>59</sup>

### 6.1.1 Measures of central tendency

Central tendency is one of the most common numerical descriptive measures. It's used to estimate the central location of the univariate data by the calculation of mean, median and mode.[9] Each of these calculations has its own advantages and limitations. The mean has the advantage that its calculation includes each value of the data set, but it is particularly susceptible to the influence of outliers. The median is a better measure when the data set contains outliers. The mode is simple to locate. The important thing is that it's not restricted to using only one of these measure of central tendency. If the data being analyzed is categorical, then the only measure of central tendency that can be used is the mode. However, if the data is numerical in nature (ordinal or interval/ratio) then the mode, median, or mean can all be used to describe the data. Using more than one of these measures provides a more accurate descriptive summary of central tendency for the univariate.

### Python practice

Following code shows as how to simulate data using a module called `numpy`. This library has very nice ways to simulate data sets using few probability distributions.

```
>>> import numpy as np
>>> x = np.random.normal(0, 1, 10)
>>> x
array([ 1.03366928,  1.52520786, -1.17460073, -1.66790491,  0.
       ↪ 15278477,
       -1.99390991,  0.73550033, -1.07696366,  1.06895286, -0.
       ↪ 38274239])
```

```
>>> y = [1, 2, 1, 1, 2, 1, 3, 4, 5]
```

The object “ $x$ ” has 10 data points that were simulated by using a mean value of 0 and a standard deviation 1. The object “ $y$ ” is being created using few integers. Now let us explore the central tendency measures using the same library (*numpy*).

```
>>> x.mean()
-0.1780006495604616
>>> np.median(x)
-0.11497881050239521
```

The mean is not exactly 0, but close. The other measure i.e., *mode* can't be calculated directly using *numpy*. See the below snippet of code that shows the procedure to calculate *mode* using *numpy*.

```
>>> np.unique(y)
array([1, 2, 3, 4, 5])
>>> np.unique(y, return_counts=True)
(array([1, 2, 3, 4, 5]), array([4, 2, 1, 1, 1], dtype=int64))
```

There is certain library called *statistics*.<sup>60</sup> This package has got functions for most of the measures of central tendency can be calculated as following.

```
>>> st.mean(x)
-0.17800064956046163
>>> st.median(x)
-0.11497881050239521
>>> st.mode(y)
1
```

The other two very essential measures of central tendency are (1) geometric mean and (2) harmonic mean. These measures can be calculated using following procedure.

```
x = np.random.uniform(0, 10, 10)
st.geometric_mean(x)
3.502846158314328
st.mean(x)
4.527877004355158
st.geometric_mean(x)
3.502846158314328
```

```
st.harmonic_mean(x)
2.541531096522052
```

### 6.1.2 Measures of variability

A measure of variability or dispersion (deviation from the mean) of a univariate data set can reveal the shape of a univariate data distribution more sufficiently. It will provide some information about the variation among data values. The measures of variability together with the measures of central tendency give a better picture of the data than the measures of central tendency alone. The three most frequently used measures of variability are range, variance and standard deviation. The appropriateness of each measure would depend on the type of data, the shape of the distribution of data and which measure of central tendency are being used. If the data is categorical, then there is no measure of variability to report. For data that is numerical, all three measures are possible. If the distribution of data is symmetrical, then the measures of variability are usually the variance and standard deviation. However, if the data are skewed, then the measure of variability that would be appropriate for that data set is the range.

#### Python practice

There are roughly a dozen methods to measure variability of a numerical distribution. The very first yet best measure of dispersion or variability is variance and followed by standard deviation. The following snippet shows as how to calculate both variance and standard deviation for simulated data sets.

```
>>> x.var()
7.8683941632286345
>>> x.std()
1.1928313325533353
>>> st.variance(x)
8.742660181365151
>>> st.stdev(x)
1.257354625094097
```

In statistics there is a concept called *quantiles* these measures describe central values for various parts of distribution say 25%, 50% and 75%. A quartile is a type of quantile which divides the number of data points into four parts, or quarters, of more-or-less equal size. The data must be ordered from smallest to largest to compute quartiles; as such, quartiles are a form of order statistic. The three main quartiles are as follows:

1. The first quartile (Q1) is defined as the middle number between the smallest number (minimum) and the median of the data set. It is also known as the lower or 25th empirical quartile, as 25% of the data is below this point.
2. The second quartile (Q2) is the median of a data set; thus 50% of the data lies below this point. The third quartile (Q3) is the middle value between the median and the highest value (maximum) of the data set. It is known as the upper or 75th empirical quartile, as 75% of the data lies below this point.

Along with the minimum and maximum of the data (which are also quartiles), the three quartiles described above provide a five-number summary of the data. This summary is important in statistics because it provides information about both the center and the spread of the data. Knowing the lower and upper quartile provides information on how big the spread is and if the dataset is skewed toward one side. Since quartiles divide the number of data points evenly, the range is not the same between quartiles (i.e.,  $Q3 - Q2 \neq Q2 - Q1$ ) and is instead known as the interquartile range (IQR). While the maximum and minimum also show the spread of the data, the upper and lower quartiles can provide more detailed information on the location of specific data points, the presence of outliers in the data, and the difference in spread between the middle 50% of the data and the outer data points. The package *numpy* can help us greatly while calculating these quantiles.

```
np.quantile(x, 0.25)
2.054117999713459
np.quantile(x, 0.5)
4.294556898345667
np.quantile(x, 0.75)
6.872170277127191
```

```
np.quantile(x, 1)
9.233681015174586
x.max()
9.233681015174586
```

The interquartile range (IQR) can be calculated as shown below.

```
q75, q25 = np.percentile(x, [75, 25])
iqr = q75 - q25
iqr
4.818052277413733
```

or it can be done using python's unique *asterisk (\*)* operator.

```
iqr = np.subtract(*np.percentile(x, [75, 25]))
iqr
4.818052277413733
```

### 6.1.3 Measures of shape

In statistics, measures of shape refers to skewness and kurtosis.

#### Skewness

Skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean. The skewness value can be positive, zero, negative, or undefined.

For a unimodal distribution, negative skew commonly indicates that the tail is on the left side of the distribution, and positive skew indicates that the tail is on the right. In cases where one tail is long but the other tail is fat, skewness does not obey a simple rule. For example, a zero value means that the tails on both sides of the mean balance out overall; this is the case for a symmetric distribution, but can also be true for an asymmetric distribution where one tail is long and thin, and the other is short but fat.

Consider the two distributions in the figure just below. Within each graph, the values on the right side of the distribution taper differently from the values on the left side. These tapering sides are called tails,

and they provide a visual means to determine which of the two kinds of skewness a distribution has:

1. negative skew: The left tail is longer; the mass of the distribution is concentrated on the right of the figure. The distribution is said to be left-skewed, left-tailed, or skewed to the left, despite the fact that the curve itself appears to be skewed or leaning to the right; left instead refers to the left tail being drawn out and, often, the mean being skewed to the left of a typical center of the data. A left-skewed distribution usually appears as a right-leaning curve.
2. positive skew: The right tail is longer; the mass of the distribution is concentrated on the left of the figure. The distribution is said to be right-skewed, right-tailed, or skewed to the right, despite the fact that the curve itself appears to be skewed or leaning to the left; right instead refers to the right tail being drawn out and, often, the mean being skewed to the right of a typical center of the data. A right-skewed distribution usually appears as a left-leaning curve.

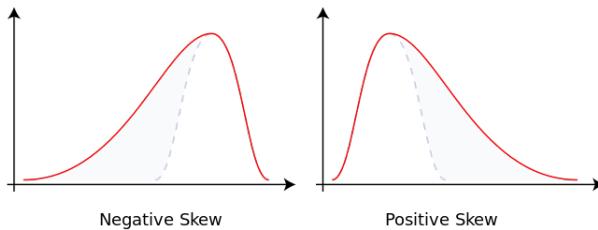


Figure 6.1: Negative and positive skewness

### Python practice

For normally distributed data, the skewness should be about zero. For unimodal continuous distributions, a skewness value greater than zero means that there is more weight in the right tail of the distribution. The function `skewtest` can be used to determine if the skewness value is close enough to zero, statistically speaking.

The sample skewness is computed as the Fisher-Pearson coefficient of skewness, i.e.

$$g_1 = \frac{m_3}{m_2^{3/2}}$$

where

$$m_i = \frac{1}{N} \sum_{n=1}^N \frac{x_n^i}{\bar{x}}$$

is the biased sample  $i^{th}$  central moment, and  $\bar{x}$  is the sample mean. If *bias* is False, the calculations are corrected for bias and the value computed is the adjusted Fisher-Pearson standardized moment coefficient, i.e.

$$G_1 = \frac{k_3}{k_2^{3/2}} = \frac{\sqrt{N(N-1)}}{N-2} \frac{m_3}{m_2^{3/2}}$$

```
from scipy.stats import skew
skew([1, 2, 3, 4, 5])
0.0
skew([2, 8, 0, 4, 1, 9, 9, 0])
0.2650554122698573
```

## Kurtosis

kurtosis is a measure of the “tailedness” of the probability distribution of a real-valued random variable. Like skewness, kurtosis describes the shape of a probability distribution and there are different ways of quantifying it for a theoretical distribution and corresponding ways of estimating it from a sample from a population. Different measures of kurtosis may have different interpretations.

The standard measure of a distribution’s kurtosis, originating with Karl Pearson, is a scaled version of the fourth moment of the distribution. This number is related to the tails of the distribution, not its peak; hence, the sometimes-seen characterization of kurtosis as “peakedness” is incorrect. For this measure, higher kurtosis corresponds to greater extremity of deviations (or outliers), and not the configuration of data near the mean.

### Python practice

Kurtosis is the fourth central moment divided by the square of the variance. If Fisher's definition is used, then 3.0 is subtracted from the result to give 0.0 for a normal distribution. If bias is False then the kurtosis is calculated using k statistics to eliminate bias coming from biased moment estimators. In Fisher's definition, the kurtosis of the normal distribution is zero. In the following example, the kurtosis is close to zero, because it was calculated from the dataset, not from the continuous distribution.

```
from scipy.stats import norm, kurtosis
data = norm.rvs(size=1000, random_state=3)
kurtosis(data)
-0.06928694200380558
```

The distribution with a higher kurtosis has a heavier tail. The zero valued kurtosis of the normal distribution in Fisher's definition can serve as a reference point.

```
import matplotlib.pyplot as plt
import scipy.stats as stats
from scipy.stats import kurtosis

x = np.linspace(-5, 5, 100)
ax = plt.subplot()

distnames = ['laplace', 'norm', 'uniform']

for distname in distnames:
    if distname == 'uniform':
        dist = getattr(stats, distname)(loc=-2, scale=4)
    else:
        dist = getattr(stats, distname)
    data = dist.rvs(size=1000)
    kur = kurtosis(data, fisher=True)
    y = dist.pdf(x)
    ax.plot(x, y, label="{}: {}".format(distname, round(kur, 3)))
ax.legend()
```

The Laplace distribution has a heavier tail than the normal distribution. The uniform distribution (which has negative kurtosis) has the thinnest tail.

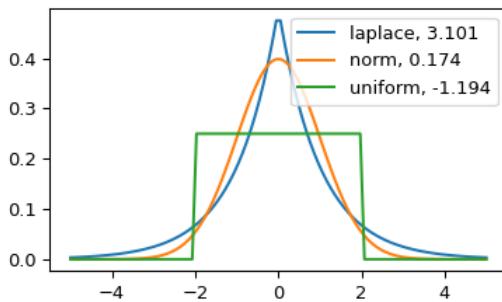


Figure 6.2: Kurtosis for three different distributions

#### 6.1.4 Student's t-test

The t-test is any statistical hypothesis test in which the test statistic follows a Student's t-distribution under the null hypothesis. A t-test is the most commonly applied when the test statistic would follow a normal distribution if the value of a scaling term in the test statistic were known. When the scaling term is unknown and is replaced by an estimate based on the data, the test statistics (under certain conditions) follow a Student's t distribution. The t-test can be used, for example, to determine if the means of two sets of data are significantly different from each other. Among the most frequently used t-tests are:

1. A one-sample location test of whether the mean of a population has a value specified in a null hypothesis.
2. A two-sample location test of the null hypothesis such that the means of two populations are equal. All such tests are usually called Student's t-tests, though strictly speaking that name should only be used if the variances of the two populations are also assumed to be equal; the form of the test used when this assumption is dropped is sometimes called Welch's t-test. These tests are often referred to as unpaired or independent samples t-tests, as they are typically applied when the statistical units

underlying the two samples being compared are non-overlapping.

Most test statistics have the form  $t = Z/s$ , where  $Z$  and  $s$  are functions of the data.  $Z$  may be sensitive to the alternative hypothesis (i.e., its magnitude tends to be larger when the alternative hypothesis is true), whereas  $s$  is a scaling parameter that allows the distribution of  $t$  to be determined.

As an example, in the one-sample t-test

$$t = \frac{Z}{s} = \frac{\bar{X} - \mu}{\hat{\sigma}/\sqrt{n}}$$

where  $X$  is the sample mean from a sample  $X_1, X_2, \dots, X_n$ , of size  $n$ ,  $s$  is the standard error of the mean,  $\hat{\sigma}$  is the estimate of the standard deviation of the population, and  $\mu$  is the population mean. The assumptions underlying a t-test in the simplest form above are that:

1. follows a normal distribution with mean and variance  $\sigma^2/n$
2.  $s^2(n)/\sigma^2$  follows a  $\chi^2$  distribution with  $n-1$  degrees of freedom.  
This assumption is met when the observations used for estimating  $s^2$  come from a normal distribution (and i.i.d for each group).
3.  $Z$  and  $s$  are independent.

### Python practice

One sample  $t$  test for the null hypothesis that the expected value (mean) of a sample of independent observations  $a$  is equal to the given population *mean*, *popmean*.

```
from scipy import stats
rng = np.random.default_rng()
rvs = stats.norm.rvs(loc=5, scale=10, size=(50, 2),
                     random_state=rng)
```

Test if mean of random sample is equal to true mean, and different mean. We reject the null hypothesis in the second case and don't reject it in the first case.

```
stats.ttest_1samp(rvs, 5.0)
Ttest_1sampResult(statistic=array([-2.09794637, -1.75977004]),
                   pvalue=array([0.04108952, 0.08468867]))
```

```
stats.ttest_1samp(rvs, 0.0)
Ttest_1sampResult(statistic=array([1.64495065, 1.62095307]),
                   pvalue=array([0.10638103, 0.
                   ↪ 11144602]))
```

## 6.2 Bivariate analysis

### 6.2.1 Two-sample t-test

#### Equal sample sizes and variance

Given two groups (1, 2), this test is only applicable when:

1. the two sample sizes (that is, the number  $n$  of participants of each group) are equal;
2. it can be assumed that the two distributions have the same variance;

Violations of these assumptions are discussed below.

The t statistic to test whether the means are different can be calculated as follows:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{2}{n} s_p^2}} \quad (6.1)$$

where

$$s_p = \sqrt{\frac{s_{X_1}^2 + s_{X_2}^2}{2}}.$$

Here  $s_p$  is the pooled standard deviation for  $n = n_1 = n_2$  and  $s_{X_1}^2$  and  $s_{X_2}^2$  are the unbiased estimators of the variances of the two samples. The denominator of t is the standard error of the difference between two means. For significance testing, the degrees of freedom for this test is  $2n - 2$  where  $n$  is the number of participants in each group.

### Equal or unequal sample sizes, similar variances

This test is used only when it can be assumed that the two distributions have the same variance. (When this assumption is violated, see below.) The previous formulae are a special case of the formulae below, one recovers them when both samples are equal in size:  $n = n_1 = n_2$ .

The t statistic to test whether the means are different can be calculated as follows:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{s_p \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

where

$$s_p = \sqrt{\frac{(n_1-1)s_{X_1}^2 + (n_2-1)s_{X_2}^2}{n_1+n_2-2}}$$

is an estimator of the pooled standard deviation of the two samples: it is defined in this way so that its square is an unbiased estimator of the common variance whether or not the population means are the same. In these formulae,  $n_1$  is the number of degrees of freedom for each group, and the total sample size minus two (that is,  $n_1 + n_2 - 2$ ) is the total number of degrees of freedom, which is used in significance testing.

### Equal or unequal sample sizes, unequal variances

This test, also known as Welch's t-test, is used only when the two population variances are not assumed to be equal (the two sample sizes may or may not be equal) and hence must be estimated separately. The t statistic to test whether the population means are different is calculated as:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{s_{\bar{\Delta}}}$$

where

$$s_{\bar{\Delta}} = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}.$$

Here  $s_i^2$  is the unbiased estimator of the variance of each of the two samples with  $n_i$  = number of participants in group  $i$  ( $i = 1$  or  $2$ ). In this case  $(s_{\bar{\Delta}})^2$  is not a pooled variance. For use in significance testing,

the distribution of the test statistic is approximated as an ordinary Student's t-distribution with the degrees of freedom calculated using

$$\text{d.f.} = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{(s_1^2/n_1)^2}{n_1-1} + \frac{(s_2^2/n_2)^2}{n_2-1}}.$$

This is known as the Welch–Satterthwaite equation. The true distribution of the test statistic actually depends (slightly) on the two unknown population variances.

Two sample *t test* for the null hypothesis that 2 independent samples have identical average (expected) values. This test assumes that the populations have identical variances by default. Suppose we observe two independent samples, e.g. flower petal lengths, and we are considering whether the two samples were drawn from the same population (e.g. the same species of flower or two species with similar petal characteristics) or two different populations.

The t-test quantifies the difference between the arithmetic means of the two samples. The p-value quantifies the probability of observing as or more extreme values assuming the null hypothesis, that the samples are drawn from populations with the same population means, is true. A p-value larger than a chosen threshold (e.g. 5% or 1%) indicates that our observation is not so unlikely to have occurred by chance. Therefore, we do not reject the null hypothesis of equal population means. If the p-value is smaller than our threshold, then we have evidence against the null hypothesis of equal population means.

By default, the p-value is determined by comparing the t-statistic of the observed data against a theoretical t-distribution. When  $1 < \text{permutations} < \text{binom}(n, k)$ , where

1.  $k$  is the number of observations in  $a$ ,
2.  $n$  is the total number of observations in  $a$  and  $b$ , and
3.  $\text{binom}(n, k)$  is the binomial coefficient ( $n$  choose  $k$ ),

The data are pooled (concatenated), randomly assigned to either group a or b, and the t-statistic is calculated. This process is performed repeatedly (permutation times), generating a distribution of the t-

statistic under the null hypothesis, and the t-statistic of the observed data is compared to this distribution to determine the p-value. When permutations  $\geq \text{binom}(n, k)$ , an exact test is performed: the data are partitioned between the groups in each distinct way exactly once.

The permutation test can be computationally expensive and not necessarily more accurate than the analytical test, but it does not make strong assumptions about the shape of the underlying distribution.

Use of trimming is commonly referred to as the trimmed t-test. At times called Yuen's t-test, this is an extension of Welch's t-test, with the difference being the use of winsorized means in calculation of the variance and the trimmed sample size in calculation of the statistic. Trimming is recommended if the underlying distribution is long-tailed or contaminated with outliers.

```
from scipy import stats
rng = np.random.default_rng()
```

Test with sample with identical means:

```
rvs1 = stats.norm.rvs(loc=5, scale=10, size=500, random_state=
                      ↪ rng)
rvs2 = stats.norm.rvs(loc=5, scale=10, size=500, random_state=
                      ↪ rng)
stats.ttest_ind(rvs1, rvs2)
Ttest_indResult(statistic=-0.4390847099199348, pvalue=0.
                ↪ 6606952038870015)
stats.ttest_ind(rvs1, rvs2, equal_var=False)
Ttest_indResult(statistic=-0.4390847099199348, pvalue=0.
                ↪ 6606952553131064)
```

*ttest\_ind* underestimates  $p$  for unequal variances:

```
rvs3 = stats.norm.rvs(loc=5, scale=20, size=500, random_state=
                      ↪ rng)
stats.ttest_ind(rvs1, rvs3)
Ttest_indResult(statistic=-1.6370984482905417, pvalue=0.
                ↪ 1019251574705033)
stats.ttest_ind(rvs1, rvs3, equal_var=False)
Ttest_indResult(statistic=-1.637098448290542, pvalue=0.
                ↪ 10202110497954867)
```

When  $n1! = n2$ , the equal variance t-statistic is no longer equal to the unequal variance t-statistic:

```
rvs4 = stats.norm.rvs(loc=5, scale=20, size=100, random_state=
                      ↪ rng)
stats.ttest_ind(rvs1, rvs4)
Ttest_indResult(statistic=-1.9481646859513422, pvalue=0.
                  ↪ 05186270935842703)
stats.ttest_ind(rvs1, rvs4, equal_var=False)
Ttest_indResult(statistic=-1.3146566100751664, pvalue=0.
                  ↪ 1913495266513811)
T-test with different means, variance, and n:
rvs5 = stats.norm.rvs(loc=8, scale=20, size=100, random_state=
                      ↪ rng)
stats.ttest_ind(rvs1, rvs5)
Ttest_indResult(statistic=-2.8415950600298774, pvalue=0.
                  ↪ 0046418707568707885)
stats.ttest_ind(rvs1, rvs5, equal_var=False)
Ttest_indResult(statistic=-1.8686598649188084, pvalue=0.
                  ↪ 06434714193919686)
```

When performing a permutation test, more permutations typically yields more accurate results. Use a `np.random.Generator` to ensure reproducibility:

```
stats.ttest_ind(rvs1, rvs5, permutations=10000,
                random_state=rng)
Ttest_indResult(statistic=-2.8415950600298774, pvalue=0.0052)
```

Take these two samples, one of which has an extreme tail.

```
a = (56, 128.6, 12, 123.8, 64.34, 78, 763.3)
b = (1.1, 2.9, 4.2)
```

Use the `trim` keyword to perform a trimmed (Yuen) t-test. For example, using 20% trimming, `trim=.2`, the test will reduce the impact of one (`np.floor(trim*len(a))`) element from each tail of sample *a*. It will have no effect on sample *b* because `np.floor(trim*len(b))` is 0.

```
stats.ttest_ind(a, b, trim=.2)
Ttest_indResult(statistic=3.4463884028073513,
                pvalue=0.01369338726499547)
```

### 6.2.2 Chi-squared test

A chi-squared test (also chi-square or  $\chi^2$  test) is a statistical hypothesis test that is valid to perform when the test statistic is chi-squared distributed under the null hypothesis, specifically Pearson's chi-squared test and variants thereof. Pearson's chi-squared test is used to determine whether there is a statistically significant difference between the expected frequencies and the observed frequencies in one or more categories of a contingency table.

In the standard applications of this test, the observations are classified into mutually exclusive classes. If the null hypothesis that there are no differences between the classes in the population is true, the test statistic computed from the observations follows a  $\chi^2$  frequency distribution. The purpose of the test is to evaluate how likely the observed frequencies would be assuming the null hypothesis is true. Test statistics that follow a  $\chi^2$  distribution occur when the observations are independent. There are also  $\chi^2$  tests for testing the null hypothesis of independence of a pair of random variables based on observations of the pairs.

Chi-squared tests often refers to tests for which the distribution of the test statistic approaches the  $\chi^2$  distribution asymptotically, meaning that the sampling distribution (if the null hypothesis is true) of the test statistic approximates a chi-squared distribution more and more closely as sample sizes increase.

Suppose that  $n$  observations in a random sample from a population are classified into  $k$  mutually exclusive classes with respective observed numbers  $x_i$  (for  $i = 1, 2, \dots, k$ ), and a null hypothesis gives the probability  $p_i$  that an observation falls into the  $i^{th}$  class. So we have the expected numbers  $m_i = np_i$  for all  $i$ , where

$$\sum_{i=1}^k p_i = 1$$

$$\sum_{i=1}^k m_i = n \sum_{i=1}^k p_i = n = \sum_{i=1}^k x_i$$

Pearson proposed that, under the circumstance of the null hypothesis being correct, as  $n \rightarrow \infty$  the limiting distribution of the quantity given below is the  $\chi^2$  distribution.

$$X^2 = \sum_{i=1}^k \frac{(x_i - m_i)^2}{m_i} = \sum_{i=1}^k \frac{x_i^2}{m_i} - n$$

### Python practice

The chi-square test tests the null hypothesis that the categorical data has the given frequencies. This test is invalid when the observed or expected frequencies in each category are too small. A typical rule is that all of the observed and expected frequencies should be at least 5. According to , the total number of samples is recommended to be greater than 13, otherwise exact tests (such as Barnard's Exact test) should be used because they do not overreject.

Also, the sum of the observed and expected frequencies must be the same for the test to be valid; `chisquare` raises an error if the sums do not agree within a relative tolerance of  $1e-8$ .

The default degrees of freedom,  $k - 1$ , are for the case when no parameters of the distribution are estimated. If  $p$  parameters are estimated by efficient maximum likelihood then the correct degrees of freedom are  $k - 1 - p$ . If the parameters are estimated in a different way, then the `dof` can be between  $k - 1 - p$  and  $k - 1$ . However, it is also possible that the asymptotic distribution is not chi-square, in which case this test is not appropriate.

When just `f_obs` is given, it is assumed that the expected frequencies are uniform and given by the mean of the observed frequencies.

```
from scipy.stats import chisquare
chisquare([16, 18, 16, 14, 12, 12])
(2.0, 0.84914503608460956)
```

With `f_exp` the expected frequencies can be given.

```
chisquare([16, 18, 16, 14, 12, 12], f_exp=[16, 16, 16, 16,
                                             ↪ 8])
(3.5, 0.62338762774958223)
```

When `f_obs` is  $2 - D$ , by default the test is applied to each column.

```

obs = np.array([[16, 18, 16, 14, 12, 12], [32, 24, 16, 28, 20,
                                         ↪ 24]]).T
obs.shape
(6, 2)
chisquare(obs)
(array([ 2.           ,  6.66666667]), array([ 0.84914504,  0.
                                         ↪ 24663415]))

```

By setting  $axis = None$ , the test is applied to all data in the array, which is equivalent to applying the test to the flattened array.

```

chisquare(obs, axis=None)
(23.31034482758621, 0.015975692534127565)
chisquare(obs.ravel())
(23.31034482758621, 0.015975692534127565)

```

`ddof` is the change to make to the default degrees of freedom.

```

chisquare([16, 18, 16, 14, 12, 12], ddof=1)
(2.0, 0.73575888234288467)

```

The calculation of the p-values is done by broadcasting the chi-squared statistic with `ddof`.

```

chisquare([16, 18, 16, 14, 12, 12], ddof=[0,1,2])
(2.0, array([ 0.84914504,  0.73575888,  0.5724067 ]))

```

$f_{obs}$  and  $f_{exp}$  are also broadcast. In the following,  $f_{obs}$  has shape (6,) and  $f_{exp}$  has shape (2, 6), so the result of broadcasting  $f_{obs}$  and  $f_{exp}$  has shape (2, 6). To compute the desired chi-squared statistics, we use  $axis=1$ :

```

chisquare([16, 18, 16, 14, 12, 12],
          f_exp=[[16, 16, 16, 16, 16, 8], [8, 20, 20, 16, 12,
                                         ↪ 12]], 
          axis=1)
(array([ 3.5 ,  9.25]), array([ 0.62338763,  0.09949846]))

```

### 6.2.3 Cross tabulation

A contingency table is a type of table in a matrix format that displays the (multivariate) frequency distribution of the variables. They are

heavily used in survey research, business intelligence, engineering, and scientific research. They provide a basic picture of the interrelation between two variables and can help find interactions between them. The term contingency table was first used by Karl Pearson in “On the Theory of Contingency and Its Relation to Association and Normal Correlation”, part of the Drapers’ Company Research Memoirs Biometric Series I published in 1904.

A crucial problem of multivariate statistics is finding the dependence structure underlying the variables contained in high-dimensional contingency tables. If some of the conditional interdependence are revealed, then even the storage of the data can be done in a smarter way. In order to do this one can use information theory concepts, which gain the information only from the distribution of probability, which can be expressed easily from the contingency table by the relative frequencies. A pivot table is a way to create contingency tables using spreadsheet software.

### **Example**

Suppose there are two variables, sex (male or female) and handedness (right- or left-handed). Further suppose that 100 individuals are randomly sampled from a very large population as part of a study of sex differences in handedness. A contingency table can be created to display the numbers of individuals who are male right-handed and left-handed, female right-handed and left-handed. Such a contingency table is shown below.

Type Sex \	Right-handedness	Left-handedness	Total
Male	43	9	52
Female	44	4	48
Total	87	13	100

Table 6.1: Cross table

The numbers of the males, females, and right- and left-handed individuals are called marginal totals. The grand total (the total number of individuals represented in the contingency table) is the number in

the bottom right corner.

The table allows users to see at a glance that the proportion of men who are right-handed is about the same as the proportion of women who are right-handed although the proportions are not identical. The strength of the association can be measured by the odds ratio, and the population odds ratio estimated by the sample odds ratio. The significance of the difference between the two proportions can be assessed with a variety of statistical tests including Pearson's chi-squared test, the G-test, Fisher's exact test, Boschloo's test, and Barnard's test, provided the entries in the table represent individuals randomly sampled from the population about which conclusions are to be drawn. If the proportions of individuals in the different columns vary significantly between rows (or vice versa), it is said that there is a contingency between the two variables. In other words, the two variables are not independent. If there is no contingency, it is said that the two variables are independent.

The example above is the simplest kind of contingency table, a table in which each variable has only two levels; this is called a  $2 \times 2$  contingency table. In principle, any number of rows and columns may be used. There may also be more than two variables, but higher order contingency tables are difficult to represent visually. The relation between ordinal variables, or between ordinal and categorical variables, may also be represented in contingency tables, although such a practice is rare. For more on the use of a contingency table for the relation between two ordinal variables, see Goodman and Kruskal's gamma.

### Python practice

`scipy.stats.contingency.crosstab` class return table of counts for each possible unique combination in `*args`. When `len(args) > 1`, the array computed by this function is often referred to as a contingency table. The arguments must be sequences with the same length. The second return value, `count`, is an integer array with `len(args)` dimensions. If `levels` is `None`, the shape of `count` is  $(n_0, n_1, \dots)$ , where  $n_k$  is the number of unique elements in `args[k]`.

```
>>> from scipy.stats.contingency import crosstab
>>> a = ['A', 'B', 'A', 'A', 'B', 'B', 'A', 'A', 'B', 'B']
```

```
>>> x = ['X', 'X', 'X', 'Y', 'Z', 'Z', 'Y', 'Y', 'Z', 'Z']
>>> (avals, xvals), count = crosstab(a, x)
>>> avals
array(['A', 'B'], dtype='<U1')
>>> xvals
array(['X', 'Y', 'Z'], dtype='<U1')
>>> count
array([[2, 3, 0],
       [1, 0, 4]])
```

The above (`counts`) array should be visualized as in the below given form

	X	Y	Z
A	2	3	0
B	1	0	4

Table 6.2: Cross table

### 6.2.4 Measures of association

The degree of association between the two variables can be assessed by a number of coefficients. The following subsections describe a few of them. For a more complete discussion of their uses, see the main articles linked under each subsection heading.

#### Odds ratio

Odds ratio is a bivariate analysis. It is two-way crosstab. Odds ratio is the simplest measure of association for a 2 contingency table is the odds ratio. Given two events, A and B, the odds ratio is defined as the ratio of the odds of A in the presence of B and the odds of A in the absence of B, or equivalently (due to symmetry), the ratio of the odds of B in the presence of A and the odds of B in the absence of A. Two events are independent if and only if the odds ratio is 1; if the odds ratio is greater than 1, the events are positively associated; if the odds ratio is less than 1, the events are negatively associated. The odds ratio has a simple expression in terms of probabilities; given the joint probability distribution:

	B=1	B=0
A=1	$p_{11}$	$p_{10}$
A=0	$p_{01}$	$p_{00}$

Table 6.3: Cross table

the odds ratio is:

$$OR = \frac{p_{11}p_{00}}{p_{10}p_{01}}.$$

### Python practice

```
>>> x = ['X', 'X', 'X', 'Y', 'X', 'Y', 'Y', 'Y', 'X', 'Y']
>>> crosstab(a, x)
((array(['A', 'B']), dtype='<U1'), array(['X', 'Y', 'Z']), dtype=
   → '<U1')), array([[2, 3, 0],
      [1, 0, 4]))
>>> (_ , _), table = crosstab(a, x)
>>> table
array([[2, 3, 0],
      [1, 0, 4]])
>>> (_ , _), table = crosstab(a, x)
>>> oddsr, p = fisher_exact(table, alternative='greater')
>>> oddsr
0.4444444444444444
```

Both variables are negatively associated.

### Phi coefficient

A simple measure, applicable only to the case of  $2 \times 2$  contingency tables, is the phi coefficient ( $\phi$ ) defined by

$$\phi = \pm \sqrt{\frac{\chi^2}{N}},$$

where  $\chi^2$  is computed as in Pearson's chi-squared test, and  $N$  is the grand total of observations.  $\phi$  varies from 0 (corresponding to no association between the variables) to 1 or  $-1$  (complete association or complete inverse association), provided it is based on frequency data represented in  $2 \times 2$  tables. Then its sign equals the sign of the

product of the main diagonal elements of the table minus the product of the off-diagonal elements.  $\phi$  takes on the minimum value  $-1.0$  or the maximum value of  $+1.0$  if and only if every marginal proportion is equal to  $0.5$  (and two diagonal cells are empty).

Two alternatives or equivalent measures for  $\phi$  are the contingency coefficient C, and Cramér's V.

The formulae for the C and V coefficients are:

$$C = \sqrt{\frac{\chi^2}{N + \chi^2}} \text{ and } V = \sqrt{\frac{\chi^2}{N(k-1)}},$$

$k$  being the number of rows or the number of columns, whichever is less.

$C$  suffers from the disadvantage that it does not reach a maximum of  $1.0$ , notably the highest it can reach in a  $2 \times 2$  table is  $0.707$ . It can reach values closer to  $1.0$  in contingency tables with more categories; for example, it can reach a maximum of  $0.870$  in a  $4 \times 4$  table. It should, therefore, not be used to compare associations in different tables if they have different numbers of categories.

$C$  can be adjusted so it reaches a maximum of  $1.0$  when there is complete association in a table of any number of rows and columns by dividing  $C$  by  $\sqrt{\frac{k-1}{k}}$  where  $k$  is the number of rows or columns, when

the table is square, or by  $\sqrt[4]{\frac{r-1}{r} \times \frac{c-1}{c}}$  where  $r$  is the number of rows and  $c$  is the number of columns.

## Python practice

The class `scipy.stats.contingency.association` has the methods to compute  $\phi$  coefficient. The function provides the option for computing one of three measures of association between two nominal variables from the data given in a 2d contingency table: Tschuprow's T, Pearson's Contingency Coefficient and Cramer's V. Both the Cramer's V and Tschuprow's T are extensions of the phi coefficient. Moreover, due to the close relationship between the Cramer's V and Tschuprow's T the returned values can often be similar or even equivalent. They are likely to diverge more as the array shape diverges from a  $2 \times 2$ .

Cramer's V, Tschuprow's T and Pearson's Contingency Coefficient, all measure the degree to which two nominal or ordinal variables are related, or the level of their association. This differs from correlation, although many often mistakenly consider them equivalent. Correlation measures in what way two variables are related, whereas, association measures how related the variables are. As such, association does not subsume independent variables, and is rather a test of independence. A value of 1.0 indicates perfect association, and 0.0 means the variables have no association.

An example with a  $4 \times 2$  contingency table:

```
>>> from scipy.stats.contingency import association  
>>> obs4x2 = np.array([[100, 150], [203, 322], [420, 700], [320  
    ↪ , 210]])
```

Pearson's contingency coefficient can be found

```
>>> association(obs4x2, method="pearson")  
0.18303298140595667
```

### Cramer's V

```
>>> association(obs4x2, method="cramer")  
0.18617813077483678
```

### Tschuprow's T

```
>>> association(obs4x2, method="tschuprow")  
0.14146478765062995
```

### Matthews correlation coefficient

One last way to compute  $\phi$  coefficient is through the *Matthews correlation coefficient (MCC)*. The Matthews correlation coefficient is used in machine learning as a measure of the quality of binary and multi-class classifications. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes. The MCC is in essence a correlation coefficient value between -1 and +1. A coefficient of +1 represents a perfect prediction, 0 an average random

prediction and -1 an inverse prediction. Binary and multiclass labels are supported. Only in the binary case does this relate to information about true and false positives and negatives. We need *sklearn* library to compute MCC.

```
>>> from sklearn.metrics import matthews_corrcoef
>>> y_true = [+1, +1, +1, -1]
>>> y_pred = [+1, -1, +1, +1]
>>> matthews_corrcoef(y_true, y_pred)
-0.33...
```

The coefficient show that the relationship between *y\_true* and *y\_pred* are negatively associated.

### 6.2.5 Exact tests

An exact (significance) test is a test such that if the null hypothesis is true, then all assumptions made during the derivation of the distribution of the test statistic are met. Using an exact test provides a significance test that maintains the type I error rate of the test ( $\alpha$ ) at the desired significance level of the test. For example, an exact test at a significance level of  $\alpha = 5\%$ , when repeated over many samples where the null hypothesis is true, will reject at most 5% of the time. This is in contrast to an approximate test in which the desired type I error rate is only approximately maintained (i.e.: the test might reject  $> 5\%$  of the time), while this approximation may be made as close to  $\alpha$  as desired by making the sample size sufficiently large.

Exact tests that are based on discrete test statistics may be conservative, indicating that the actual rejection rate lies below the nominal significance level  $\alpha$ . As an example, this is the case for Fisher's exact test and its more powerful alternative, Boschloo's test. If the test statistic is continuous, it will reach the significance level exactly.

Parametric tests, such as those used in exact statistics, are exact tests when the parametric assumptions are fully met, but in practice, the use of the term exact (significance) test is reserved for non-parametric tests, i.e., tests that do not rest on parametric assumptions. However, in practice, most implementations of non-parametric test software use

asymptotical algorithms to obtain the significance value, which renders the test non-exact.

Hence, when a result of statistical analysis is termed an “exact test” or specifies an “exact p-value”, this implies that the test is defined without parametric assumptions and is evaluated without making use of approximate algorithms. In principle, however, this could also signify that a parametric test has been employed in a situation where all parametric assumptions are fully met, but it is in most cases impossible to prove this completely in a real-world situation. Exceptions in which it is certain that parametric tests are exact include tests based on the binomial or Poisson distributions. The term permutation test is sometimes used as a synonym for exact test, but it should be kept in mind that all permutation tests are exact tests, but not all exact tests are permutation tests.

The basic equation underlying exact tests is

$$\Pr(\text{exact}) = \sum_{\mathbf{y}: T(\mathbf{y}) \geq T(\mathbf{x})} \Pr(\mathbf{y})$$

where:

$x$  is the actual observed outcome,  $\Pr(y)$  is the probability under the null hypothesis of a potentially observed outcome  $y$ ,  $T(y)$  is the value of the test statistic for an outcome  $y$ , with larger values of  $T$  representing cases which notionally represent greater departures from the null hypothesis, and where the sum ranges over all outcomes  $y$  (including the observed one) that have the same value of the test statistic obtained for the observed sample  $x$ , or a larger one.

### Python practice

There a few exact tests such as *fisher*, *barnard* and *Boschloo*'s. Usually, these tests are required to check or verify consistency of  $\chi^2$  test. There are quite a few conditions or rules to do so. We shall see how fisher's exact test is done as a matter of verifying the consistency of chi-squre test.

The method `scipy.stats.fisher_exact` is useful to perform fisher's exact test. The null hypothesis is that the input table is from the hypergeometric distribution with parameters (as used in `hypergeom`)

$M = a + b + c + d, n = a + b$  and  $N = a + c$ , where the input table is  $[[a, b], [c, d]]$ . This distribution has support  $\max(0, N + n - M) \leq x \leq \min(N, n)$ , or, in terms of the values in the input table,  $\min(0, a - d) \leq x \leq a + \min(b, c)$ .  $x$  can be interpreted as the upper-left element of a  $2 \times 2$  table, so the tables in the distribution have form:

$$\begin{bmatrix} x & n - x \\ N - x & M - (n + N) + x \end{bmatrix}$$

Let us perform fishers exact test for the below cross table.

$$\begin{bmatrix} 6 & 2 \\ 1 & 4 \end{bmatrix}$$

It does not matter as what is this table.

```
>>> from scipy.stats import hypergeom
>>> table = np.array([[6, 2], [1, 4]])
>>> from scipy.stats import fisher_exact
>>> odds, p = fisher_exact(table, alternative='two-sided')
>>> odds
12.0
>>> p
0.10256410256410257
```

The odds ratio is  $(6 \times 4)/2 \times 1 = 12.0$ . This is, in fact, the total amount of asymmetry/imbalance in the input matrix. P value is useful to assess if this imbalance is statistically significant or not. Since the P value is greater than 0.05 (5%), so we conclude that the imbalance in the input matrix is statistically insignificant. This means there exist some perceivable amount of balance in the data.<sup>61</sup>

### 6.2.6 Bivariate correlation

Correlation or dependence is any statistical relationship, whether causal or not, between two random variables or bivariate data. In the broadest sense correlation is any statistical association, though it actually refers to the degree to which a pair of variables are linearly related. Familiar examples of dependent phenomena include the correlation between the height of parents and their offspring, and the correlation

between the price of a good and the quantity the consumers are willing to purchase, as it is depicted in the so-called demand curve.

Correlations are useful because they can indicate a predictive relationship that can be exploited in practice. For example, an electrical utility may produce less power on a mild day based on the correlation between electricity demand and weather. In this example, there is a causal relationship, because extreme weather causes people to use more electricity for heating or cooling. However, in general, the presence of a correlation is not sufficient to infer the presence of a causal relationship (i.e., correlation does not imply causation).

### Sample correlation coefficient

Given a series of  $n$  measurements of the pair  $(X_i, Y_i)$  indexed by  $i = 1, \dots, n$ , the sample correlation coefficient can be used to estimate the population Pearson correlation  $\rho_{X,Y}$  between  $X$  and  $Y$ . The sample correlation coefficient is defined as

$$r_{xy} \stackrel{\text{def}}{=} \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{(n-1)s_x s_y} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}},$$

where  $\bar{x}$  and  $\bar{y}$  are the sample means of  $X$  and  $Y$ , and  $s_x$  and  $s_y$  are the corrected sample standard deviations of  $X$  and  $Y$ .

Equivalent expressions for  $r_{xy}$  are

$$\begin{aligned} r_{xy} &= \frac{\sum x_i y_i - n \bar{x} \bar{y}}{n s'_x s'_y} \\ &= \frac{n \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \sqrt{n \sum y_i^2 - (\sum y_i)^2}}. \end{aligned}$$

where  $s'_x$  and  $s'_y$  are the uncorrected sample standard deviations of  $X$  and  $Y$ .

If  $x$  and  $y$  are results of measurements that contain measurement error, the realistic limits on the correlation coefficient are not  $-1$  to  $+1$  but a smaller range. For the case of a linear model with a single independent

variable, the coefficient of determination ( $R$  squared) is the square of  $r_{xy}$ , Pearson's product-moment coefficient.

### Python practice

*NumPy* has many statistics routines, including `numpy.corrcoef()`, that return a matrix of Pearson correlation coefficients. You can start by importing *NumPy* and defining two *NumPy* arrays. These are instances of the class `ndarray`. The function `numpy.correlate()` performs *cross-correlation* of two 1-dimensional sequences. This function computes the correlation as generally defined in signal processing texts:

$$c_{av}[k] = \sum_n a[n + k] * \text{conj}(v[n])$$

with  $a$  and  $v$  sequences being zero-padded where necessary and  $\text{conj}$  being the conjugate.

```
np.correlate([1, 2, 3], [0, 1, 0.5])
array([3.5])
np.correlate([1, 2, 3], [0, 1, 0.5], "same")
array([2., 3.5, 3.])
np.correlate([1, 2, 3], [0, 1, 0.5], "full")
array([0.5, 2., 3.5, 3., 0.])
```

Using complex sequences:

```
np.correlate([1+1j, 2, 3-1j], [0, 1, 0.5j], 'full')
array([ 0.5-0.5j, 1.0+0.j, 1.5-1.5j, 3.0-1.j, 0.0+0.j])
```

Note that you get the time reversed, complex conjugated result when the two input sequences change places, i.e.,  $c_{va}[k] = c_{av}^*[-k]$ :

```
np.correlate([0, 1, 0.5j], [1+1j, 2, 3-1j], 'full')
array([ 0.0+0.j, 3.0+1.j, 1.5+1.5j, 1.0+0.j, 0.5+0.5j])
```

The function `numpy.corrcoef()` return Pearson product-moment correlation coefficients. Please refer to the documentation for `cov` for more detail. The relationship between the correlation coefficient matrix,  $R$ , and the covariance matrix,  $C$ , is

$$R_{ij} = C_{ij} / \sqrt{C_{ii} * C_{jj}}$$

The values of  $R$  are between -1 and 1, inclusive.

```

x = np.random.normal(0, 1, 10)
y = np.random.normal(0, 1, 10)
x
array([ 1.94965253e+00, -3.46288974e-01, -2.52211486e-01, -1.
       ↪ 02141916e+00,
       6.86401694e-02, -1.34748000e-01,  1.44670729e+00, -1.
       ↪ 54224064e-03,
       -3.62078904e-01, -8.27241737e-01])
y
array([-0.98795662,  1.14789565,  0.42134813,  0.8055516 ,  0.
       ↪ 54898627,
       -1.71166348, -0.23432952, -1.52286644, -0.88531972, -0.
       ↪ 6025864 ])
np.corrcoef(x, y)
array([[ 1.          , -0.27817599],
       [-0.27817599,  1.        ]])

```

### 6.2.7 Regression

Simple linear regression is a linear regression model with a single explanatory variable. That is, it concerns two-dimensional sample points with one independent variable and one dependent variable (conventionally, the  $x$  and  $y$  coordinates in a Cartesian coordinate system) and finds a linear function (a non-vertical straight line) that, as accurately as possible, predicts the dependent variable values as a function of the independent variable. The adjective simple refers to the fact that the outcome variable is related to a single predictor.

It is common to make the additional stipulation that the ordinary least squares (OLS) method should be used: the accuracy of each predicted value is measured by its squared residual (vertical distance between the point of the data set and the fitted line), and the goal is to make the sum of these squared deviations as small as possible. Other regression methods that can be used in place of ordinary least squares include least absolute deviations (minimizing the sum of absolute values of residuals) and the Theil–Sen estimator (which chooses a line whose slope is the median of the slopes determined by pairs of sample points). Deming regression (total least squares) also finds a line that fits a set of two-dimensional sample points, but (unlike ordinary least squares, least absolute deviations, and median slope regression) it is not really

an instance of simple linear regression, because it does not separate the coordinates into one dependent and one independent variable and could potentially return a vertical line as its fit.

## Model

Consider the model function

$$y = \alpha + \beta x,$$

which describes a line with slope  $\beta$  and  $y$ -intercept  $\alpha$ . In general such a relationship may not hold exactly for the largely unobserved population of values of the independent and dependent variables; we call the unobserved deviations from the above equation the errors. Suppose we observe  $n$  data pairs and call them  $(x_i, y_i), i = 1, \dots, n$ . We can describe the underlying relationship between  $y_i$  and  $x_i$  involving this error term  $\epsilon_i$  by

$$y_i = \alpha + \beta x_i + \epsilon_i.$$

This relationship between the true (but unobserved) underlying parameters  $\alpha$  and  $\beta$  and the data points is called a linear regression model.

The goal is to find estimated values  $\hat{\alpha}$  and  $\hat{\beta}$  for the parameters  $\alpha$  and  $\beta$  which would provide the “best” fit in some sense for the data points. As mentioned in the introduction, in this article the “best” fit will be understood as in the least-squares approach: a line that minimizes the sum of squared residuals  $\hat{\epsilon}_i$  (differences between actual and predicted values of the dependent variable  $y$ ), each of which is given by, for any candidate parameter values  $\alpha$  and  $\beta$ ,

$$\hat{\epsilon}_i = y_i - \alpha - \beta x_i.$$

In other words,  $\hat{\alpha}$  and  $\hat{\beta}$  solve the following minimization problem:

$$\text{Find } \min_{\alpha, \beta} Q(\alpha, \beta), \quad \text{for } Q(\alpha, \beta) = \sum_{i=1}^n \hat{\epsilon}_i^2 = \sum_{i=1}^n (y_i - \alpha - \beta x_i)^2.$$

By expanding to get a quadratic expression in  $\alpha$  and  $\beta$ , we can derive values of  $\alpha$  and  $\beta$  that minimize the objective function  $Q$  (these minimizing values are denoted  $\hat{\alpha}$  and  $\hat{\beta}$ ).

$$\begin{aligned}
\hat{\alpha} &= \bar{y} - (\hat{\beta} \bar{x}), \\
\hat{\beta} &= \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} \\
&= \frac{s_{x,y}}{s_x^2} \\
&= r_{xy} \frac{s_y}{s_x}.
\end{aligned}$$

Here we have introduced

1.  $\bar{x}$  and  $\bar{y}$  as the average of the  $x_i$  and  $y_i$ , respectively
2.  $r_{xy}$  as the sample correlation coefficient between  $x$  and  $y$ .
3.  $s_x$  and  $s_y$  as the uncorrected sample standard deviations of  $x$  and  $y$ .
4.  $s_x^2$  and  $s_{x,y}$  as the sample variance and sample covariance, respectively

Substituting the above expressions for  $\hat{\alpha}$  and  $\hat{\beta}$  into

$$f = \hat{\alpha} + \hat{\beta}x,$$

yields

$$\frac{f - \bar{y}}{s_y} = r_{xy} \frac{x - \bar{x}}{s_x}.$$

This shows that  $r_{xy}$  is the slope of the regression line of the standardized data points (and that this line passes through the origin). Since  $-1 \leq r_{xy} \leq 1$  then we get that if  $x$  is some measurement and  $y$  is a followup measurement from the same item, then we expect that  $y$  (on average) will be closer to the mean measurement than it was to the original value of  $x$ . This phenomenon is known as regressions toward the mean.

Generalizing the  $\bar{x}$  notation, we can write a horizontal bar over an expression to indicate the average value of that expression over the set of samples. For example:

$$\bar{xy} = \frac{1}{n} \sum_{i=1}^n x_i y_i.$$

This notation allows us a concise formula for  $r_{xy}$ :

$$r_{xy} = \frac{\bar{xy} - \bar{x}\bar{y}}{\sqrt{(\bar{x^2} - \bar{x}^2)(\bar{y^2} - \bar{y}^2)}}.$$

The coefficient of determination (“R squared”) is equal to  $r_{xy}^2$  when the model is linear with a single independent variable. See sample correlation coefficient for additional details.

### Estimation

Under the first assumption above, that of the normality of the error terms, the estimator of the slope coefficient will itself be normally distributed with mean  $\beta$  and variance  $\sigma^2 / \sum(x_i - \bar{x})^2$ , where  $\sigma^2$  is the variance of the error terms . At the same time the sum of squared residuals  $Q$  is distributed proportionally to  $\chi^2$  with  $n - 2$  degrees of freedom, and independently from  $\hat{\beta}$ . This allows us to construct a t-value

$$t = \frac{\hat{\beta} - \beta}{s_{\hat{\beta}}} \sim t_{n-2},$$

where

$$s_{\hat{\beta}} = \sqrt{\frac{\frac{1}{n-2} \sum_{i=1}^n \hat{\varepsilon}_i^2}{\sum_{i=1}^n (x_i - \bar{x})^2}}$$

is the standard error of the estimator  $\hat{\beta}$ .

#### 6.2.8 Curve fitting

The function `numpy.polyfit()` fit a polynomial  $p(x) = p[0]*x**deg + ... + p[deg]$  of degree  $deg$  to points  $(x, y)$ . Returns a vector of coefficients  $p$  that minimises the squared error in the order  $deg, deg - 1, \dots, 0$ . The `Polynomial.fit` class method is recommended for new code as it is more stable numerically. See the documentation of the method for more information.

The solution minimizes the squared error  
in the equations:

$$\begin{aligned} x[0]**n * p[0] + \dots + x[0] * p[n-1] + p[n] &= y[0] \\ x[1]**n * p[0] + \dots + x[1] * p[n-1] + p[n] &= y[1] \\ &\vdots \\ x[k]**n * p[0] + \dots + x[k] * p[n-1] + p[n] &= y[k] \end{aligned}$$

The coefficient matrix of the coefficients  $p$  is a Vandermonde matrix. `polyfit` issues a *RankWarning* when the least-squares fit is badly conditioned. This implies that the best fit is not well-defined due to numerical error. The results may be improved by lowering the polynomial degree or by replacing  $x$  by  $x - x.mean()$ . The `rcond` parameter can also be set to a value smaller than its default, but the resulting fit may be spurious: including contributions from the small singular values can add numerical noise to the result.

Note that fitting polynomial coefficients is inherently badly conditioned when the degree of the polynomial is large or the interval of sample points is badly centered. The quality of the fit should always be checked in these cases. When polynomial fits are not satisfactory, splines may be a good alternative.

```
import warnings
x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
z = np.polyfit(x, y, 3)
array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254]) # 
    ↪ may vary
```

It is convenient to use `poly1d` objects for dealing with polynomials:

```
p = np.poly1d(z)
p(0.5)
0.6143849206349179 # may vary
p(3.5)
-0.34732142857143039 # may vary
p(10)
22.579365079365115 # may vary
```

High-order polynomials may oscillate wildly:

```
with warnings.catch_warnings():
    warnings.simplefilter('ignore', np.RankWarning)
p30 = np.poly1d(np.polyfit(x, y, 30))

p30(4)
```

```
-0.800000000000000204 # may vary
p30(5)
-0.9999999999999945 # may vary
p30(4.5)
-0.10547061179440398 # may vary
```

Illustration:

```
import matplotlib.pyplot as plt
xp = np.linspace(-2, 6, 100)
_ = plt.plot(x, y, '.', xp, p(xp), '-',
             xp, p30(xp), '--')
plt.ylim(-2,2)
(-2, 2)
plt.show()
```

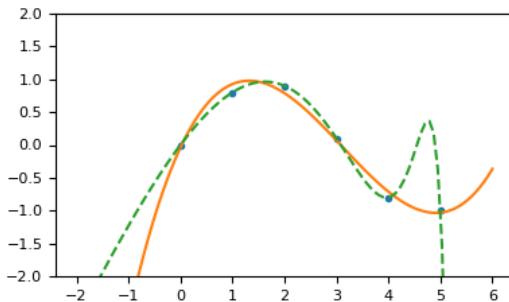


Figure 6.3: Polyfit using Numpy

### 6.2.9 Simple linear regression

The method or function `scipy.stats.linregress()` in `scipy` package can fit Calculate a regression line. This computes a least-squares regression for two sets of measurements.

```
>>> from scipy import stats
>>> import numpy as np
>>> x = np.random.random(10)
>>> y = np.random.random(10)
```

```
>>> slope, intercept, r_value, p_value, std_err = stats.  
      ↪ linregress(x,y)
```

To get coefficient of determination ( $r_squared$ )

```
>>> print "r-squared:", r_value**2  
r-squared: 0.15286643777
```

## 6.3 Multivariate analysis

Multivariate statistics is a subdivision of statistics encompassing the simultaneous observation and analysis of more than one outcome variable. Multivariate statistics concerns understanding the different aims and background of each of the different forms of multivariate analysis, and how they relate to each other. The practical application of multivariate statistics to a particular problem may involve several types of univariate and multivariate analyses in order to understand the relationships between variables and their relevance to the problem being studied. In addition, multivariate statistics is concerned with multivariate probability distributions, in terms of both

1. how these can be used to represent the distributions of observed data;
2. how they can be used as part of statistical inference, particularly where several different quantities are of interest to the same analysis.

Certain types of problems involving multivariate data, for example simple linear regression and multiple regression, are not usually considered to be special cases of multivariate statistics because the analysis is dealt with by considering the (univariate) conditional distribution of a single outcome variable given the other variables.

### 6.3.1 Multivariate analysis

Multivariate analysis (MVA) is based on the principles of multivariate statistics. Typically, MVA is used to address the situations where multiple measurements are made on each experimental unit and the

relations among these measurements and their structures are important. A modern, overlapping categorization of MVA includes:

1. Normal and general multivariate models and distribution theory
2. The study and measurement of relationships
3. Probability computations of multidimensional regions
4. The exploration of data structures and patterns

Multivariate analysis can be complicated by the desire to include physics-based analysis to calculate the effects of variables for a hierarchical “system-of-systems”. Often, studies that wish to use multivariate analysis are stalled by the dimensionality of the problem. These concerns are often eased through the use of surrogate models, highly accurate approximations of the physics-based code. Since surrogate models take the form of an equation, they can be evaluated very quickly. This becomes an enabler for large-scale MVA studies: while a Monte Carlo simulation across the design space is difficult with physics-based codes, it becomes trivial when evaluating surrogate models, which often take the form of response-surface equations.

### 6.3.2 Types of analysis

There are many different models, each with its own type of analysis:

1. Multivariate analysis of variance (MANOVA) extends the analysis of variance to cover cases where there is more than one dependent variable to be analyzed simultaneously; see also Multivariate analysis of covariance (MANCOVA).
2. Multivariate regression attempts to determine a formula that can describe how elements in a vector of variables respond simultaneously to changes in others. For linear relations, regression analyses here are based on forms of the general linear model. Some suggest that multivariate regression is distinct from multi-variable regression, however, that is debated and not consistently true across scientific fields.
3. Principal components analysis (PCA) creates a new set of orthogonal variables that contain the same information as the

original set. It rotates the axes of variation to give a new set of orthogonal axes, ordered so that they summarize decreasing proportions of the variation.

4. Factor analysis is similar to PCA but allows the user to extract a specified number of synthetic variables, fewer than the original set, leaving the remaining unexplained variation as error. The extracted variables are known as latent variables or factors; each one may be supposed to account for covariation in a group of observed variables.
5. Canonical correlation analysis finds linear relationships among two sets of variables; it is the generalised (i.e. canonical) version of bivariate correlation.
6. Redundancy analysis (RDA) is similar to canonical correlation analysis but allows the user to derive a specified number of synthetic variables from one set of (independent) variables that explain as much variance as possible in another (independent) set. It is a multivariate analogue of regression.
7. Correspondence analysis (CA), or reciprocal averaging, finds (like PCA) a set of synthetic variables that summarise the original set. The underlying model assumes chi-squared dissimilarities among records (cases).
8. Canonical (or “constrained”) correspondence analysis (CCA) for summarising the joint variation in two sets of variables (like redundancy analysis); combination of correspondence analysis and multivariate regression analysis. The underlying model assumes chi-squared dissimilarities among records (cases).
9. Multidimensional scaling comprises various algorithms to determine a set of synthetic variables that best represent the pairwise distances between records. The original method is principal coordinates analysis (PCoA; based on PCA).
10. Discriminant analysis, or canonical variate analysis, attempts to establish whether a set of variables can be used to distinguish between two or more groups of cases.
11. Linear discriminant analysis (LDA) computes a linear predictor

from two sets of normally distributed data to allow for classification of new observations.

12. Clustering systems assign objects into groups (called clusters) so that objects (cases) from the same cluster are more similar to each other than objects from different clusters.

All the above mentioned techniques are not the exhaustive list. I shall discuss very few only for moderate needs of analysis.

### 6.3.3 Covariance

Covariance is a measure of the joint variability of two random variables. If the greater values of one variable mainly correspond with the greater values of the other variable, and the same holds for the lesser values (that is, the variables tend to show similar behavior), the covariance is positive. In the opposite case, when the greater values of one variable mainly correspond to the lesser values of the other, (that is, the variables tend to show opposite behavior), the covariance is negative. The sign of the covariance therefore shows the tendency in the linear relationship between the variables. The magnitude of the covariance is not easy to interpret because it is not normalized and hence depends on the magnitudes of the variables. The normalized version of the covariance, the correlation coefficient, however, shows by its magnitude the strength of the linear relation.

A distinction must be made between (1) the covariance of two random variables, which is a population parameter that can be seen as a property of the joint probability distribution, and (2) the sample covariance, which in addition to serving as a descriptor of the sample, also serves as an estimated value of the population parameter.

For two jointly distributed real-valued random variables  $X$  and  $Y$  with finite second moments, the covariance is defined as the expected value (or mean) of the product of their deviations from their individual expected values:

$$\text{cov}(X, Y) = E[(X - E[X])(Y - E[Y])]$$

where  $E[X]$  is the expected value of  $X$ , also known as the mean of  $X$ . The covariance is also sometimes denoted  $\sigma_{XY}$  or  $\sigma(X, Y)$ , in analogy

to variance. The units of measurement of the covariance  $\text{cov}(X, Y)$  are those of  $X$  times those of  $Y$ . *By contrast, correlation coefficients, which depend on the covariance, are a dimensionless measure of linear dependence. In fact, correlation coefficients can simply be understood as a normalized version of covariance.*

If the (real) random variable pair  $(X, Y)$  can take on the values  $(x_i, y_i)$  for  $i = 1, \dots, n$ , with equal probabilities  $p_i = 1/n$ , then the covariance can be equivalently written in terms of the means  $E[X]$  and  $E[Y]$  as

$$\text{cov}(X, Y) = \frac{1}{n} \sum_{i=1}^n (x_i - E(X))(y_i - E(Y)).$$

It can also be equivalently expressed, without directly referring to the means, as

$$\begin{aligned} \text{cov}(X, Y) &= \frac{1}{n^2} \sum_{i=1}^n \sum_{j=1}^n \frac{1}{2} (x_i - x_j)(y_i - y_j) = \\ &\quad \frac{1}{n^2} \sum_i \sum_{j>i} (x_i - x_j)(y_i - y_j). \end{aligned}$$

More generally, if there are  $n$  possible realizations of  $(X, Y)$ , namely  $(x_i, y_i)$  but with possibly unequal probabilities  $p_i$  for  $i = 1, \dots, n$ , then the covariance is

$$\text{cov}(X, Y) = \sum_{i=1}^n p_i (x_i - E(X))(y_i - E(Y)).$$

### Python practice

The function `numpy.cov()` estimate a covariance matrix, given data and weights. Covariance indicates the level to which two variables vary together. If we examine N-dimensional samples, , then the covariance matrix element is the covariance of and . The element is the variance of .

Assume that the observations are in the columns of the observation array `m` and let  $f = \text{fweights}$  and  $a = \text{aweights}$  for brevity. The steps to compute the weighted covariance are as follows:

```
m = np.arange(10, dtype=np.float64)
f = np.arange(10) * 2
a = np.arange(10) ** 2.
ddof = 1
w = f * a
v1 = np.sum(w)
v2 = np.sum(w * a)
```

```
m == np.sum(m * w, axis=None, keepdims=True) / v1
cov = np.dot(m * w, m.T) * v1 / (v1**2 - ddof * v2)
```

Note that when  $a == 1$ , the normalization factor  $v1/(v1**2 - ddof*v2)$  goes over to  $1/(np.sum(f) - ddof)$  as it should.

Consider two variables, and, which correlate perfectly, but in opposite directions:

```
x = np.array([[0, 2], [1, 1], [2, 0]]).T
x
array([[0, 1, 2],
       [2, 1, 0]])
Note how increases while decreases. The covariance matrix
→ shows this clearly:

np.cov(x)
array([[ 1., -1.],
       [-1.,  1.]])
```

Note that element , which shows the correlation between and, is negative. Further, note how x and y are combined:

```
x = [-2.1, -1, 4.3]
y = [3, 1.1, 0.12]
X = np.stack((x, y), axis=0)
np.cov(X)
array([[11.71      , -4.286      ], # may vary
       [-4.286      ,  2.144133]]) 
np.cov(x, y)
array([[11.71      , -4.286      ], # may vary
       [-4.286      ,  2.144133]]) 
np.cov(x)
array(11.71)
```

### 6.3.4 Correlation

In this example we generate two random arrays,  $xarr$  and  $yarr$ , and compute the row-wise and column-wise Pearson correlation coefficients,  $R$ . Since  $rowvar$  is true by default, we first find the row-wise Pearson correlation coefficients between the variables of  $xarr$ .

```
import numpy as np
```

```

rng = np.random.default_rng(seed=42)
xarr = rng.random((3, 3))
xarr
array([[0.77395605, 0.43887844, 0.85859792],
       [0.69736803, 0.09417735, 0.97562235],
       [0.7611397 , 0.78606431, 0.12811363]])
R1 = np.corrcoef(xarr)
R1
array([[ 1.          ,  0.99256089, -0.68080986],
       [ 0.99256089,  1.          , -0.76492172],
       [-0.68080986, -0.76492172,  1.        ]])

```

If we add another set of variables and observations *yarr*, we can compute the row-wise Pearson correlation coefficients between the variables in *xarr* and *yarr*.

```

yarr = rng.random((3, 3))
yarr
array([[0.45038594, 0.37079802, 0.92676499],
       [0.64386512, 0.82276161, 0.4434142 ],
       [0.22723872, 0.55458479, 0.06381726]])
R2 = np.corrcoef(xarr, yarr)
R2
array([[ 1.          ,  0.99256089, -0.68080986,  0.75008178, -0.
         ↪ 934284   ,
         -0.99004057],
       [ 0.99256089,  1.          , -0.76492172,  0.82502011, -0.
         ↪ 97074098,
         -0.99981569],
       [-0.68080986, -0.76492172,  1.          , -0.99507202,  0.
         ↪ 89721355,
         0.77714685],
       [ 0.75008178,  0.82502011, -0.99507202,  1.          , -0.
         ↪ 93657855,
         -0.83571711],
       [-0.934284   , -0.97074098,  0.89721355, -0.93657855,  1.
         ↪           ,
         0.97517215],
       [-0.99004057, -0.99981569,  0.77714685, -0.83571711,  0.
         ↪ 97517215,
         1.        ]])

```

Finally if we use the option *rowvar=False*, the columns are now being treated as the variables and we will find the column-wise Pearson correlation coefficients between variables in *xarr* and *yarr*.

```
R3 = np.corrcoef(xarr, yarr, rowvar=False)
R3
array([[ 1.          ,  0.77598074, -0.47458546, -0.75078643, -0.
       ↪ 9665554 ,
        0.22423734],
       [ 0.77598074,  1.          , -0.92346708, -0.99923895, -0.
       ↪ 58826587,
        -0.44069024],
       [-0.47458546, -0.92346708,  1.          ,  0.93773029,  0.
       ↪ 23297648,
        0.75137473],
       [-0.75078643, -0.99923895,  0.93773029,  1.          ,
       ↪ 55627469,
        0.47536961],
       [-0.9665554 , -0.58826587,  0.23297648,  0.55627469,  1.
       ↪ ,
        -0.46666491],
       [ 0.22423734, -0.44069024,  0.75137473,  0.47536961, -0.
       ↪ 46666491,
        1.          ]])
```

### 6.3.5 Multiple regression

Multiple linear regression (MLR) is a multivariate statistical technique for examining the linear correlations between two or more independent variables (IVs) and a single dependent variable (DV). Research questions suitable for MLR can be of the form “To what extent do X1, X2, and X3 (IVs) predict Y (DV)?” e.g., “To what extent does people’s age and gender (IVs) predict their levels of blood cholesterol (DV)?” MLR analyses can be visualised as path diagrams and/or venn diagrams

#### Level of measurement

1. DV: A normally distributed interval or ratio variable
2. IVs: Two or more normally distributed interval or ratio variables or dichotomous variables. Note that it may be necessary to recode non-normal interval or ratio IVs or multichotomous categorical or ordinal IVs into dichotomous variables or a series of dummy variables).

### Sample size

Enough data is needed to provide reliable estimates of the correlations. As the number of IVs increases, more inferential tests are being conducted, therefore more data is needed, otherwise the estimates of the regression line are probably unstable and are unlikely to replicate if the study is repeated.

Some rules of thumb:

1. Use at least 50 cases plus at least 10 to 20 as many cases as there are IVs.
2.  $50 + 8(k)$  for testing an overall regression model and
3.  $104 + k$  when testing individual predictors (where  $k$  is the number of IVs)
4. Based on detecting a medium effect size ( $\beta \geq .20$ ), with critical  $\alpha \leq .05$ , with power of 80%.

To be more accurate, study-specific power and sample size calculations should be conducted.

### Normality

1. Check the univariate descriptive statistics (M, SD, skewness and kurtosis). As a general guide, skewness and kurtosis should be between -1 and +1.
2. Check the histograms with a normal curve imposed.
3. Be wary (i.e., avoid!) using inferential tests of normality (e.g., the Shapiro-Wilk test - they are notoriously overly sensitive for the purposes/needs of regression).
4. Normally distributed variables will enhance the MLR solution. Estimates of correlations will be more reliable and stable when the variables are normally distributed, but regression will be reasonably robust to minor to moderate deviations from non-normal data when moderate to large sample sizes are involved. Also examine scatterplots for bivariate outliers because non-normal uni-

variate data may make bivariate and multivariate outliers more likely.

### Linearity

Check scatterplots between the DV (Y) and each of the IVs (Xs) to determine linearity:

1. Are there any bivariate outliers? If so, consider removing the outliers.
2. Are there any non-linear relationships? If so, consider using a more appropriate type of regression.

### Homoscedasticity

Based on the scatterplots between the IVs and the DV:

1. Are the bivariate distributions reasonably evenly spread about the line of best fit?
2. Also can be checked via the normality of the residuals.

### Multicollinearity

IVs should not be overly correlated with one another. Ways to check:

1. Examine bivariate correlations and scatterplots between each of the IVs (i.e., are the predictors overly correlated - above  $\sim .7$ ?).
2. Check the collinearity statistics in the coefficients table:
  - Various recommendations for acceptable levels of VIF and Tolerance have been published.
  - Variance Inflation Factor (VIF) should be low ( $< 3$  to  $10$ ) or
  - Tolerance should be high ( $> .1$  to  $.3$ )
  - Note that VIF and Tolerance have a reciprocal relationship (i.e.,  $TOL=1/VIF$ ), so only one of the indicators needs to be used.

### Python practice

The function `numpy.linalg.lstsq()` solves the equation  $ax = b$  by computing a vector  $x$  that minimizes the norm  $\|b - ax\|$ . If  $b$  is a matrix, then all array results returned as matrices.

The following code fits a line,  $y = mx + c$ , through some noisy data-points:

```
>>> x = np.array([0, 1, 2, 3])
>>> y = np.array([-1, 0.2, 0.9, 2.1])
```

By examining the coefficients, we see that the line should have a gradient of roughly 1 and cuts the y-axis at more-or-less -1. We can rewrite the line equation as  $y = Ap$ , where  $A = [[x_1]]$  and  $p = [[m], [c]]$ . Now use `lstsq` to solve for  $p$ :

```
>>> A = np.vstack([x, np.ones(len(x))]).T
>>> A
array([[ 0.,  1.],
       [ 1.,  1.],
       [ 2.,  1.],
       [ 3.,  1.]])
>>> m, c = np.linalg.lstsq(A, y)[0]
>>> print m, c
1.0 -0.95
```

Plot the data along with the fitted line:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o', label='Original data', markersize=10)
>>> plt.plot(x, m*x + c, 'r', label='Fitted line')
>>> plt.legend()
>>> plt.show()
```

Let us see more intuitive example. The following code snippet shows as to how multiple regression can be done on truly multivariate IV. The variable  $y$  is a single column target variable (DV) and  $X$  is multivariate data set with 9 columns 7 rows.

```
import numpy as np

y = np.array([-6, -5, -10, -5, -8, -3, -6, -8, -8])
X = np.array(
```

```
[ -4.95, -4.55, -10.96, -1.08, -6.52, -0.81, -7.01, -4.
   ↪ 46, -11.54],
[ -5.87, -4.52, -11.64, -3.36, -7.45, -2.36, -7.33, -7.
   ↪ 65, -10.03],
[ -0.76, -0.71, -0.98, 0.75, -0.86, -0.50, -0.33, -0.94,
   ↪ -1.03],
[14.73, 13.74, 15.49, 24.72, 16.59, 22.44, 13.93, 11.40
   ↪ , 18.18],
[4.02, 4.47, 4.18, 4.96, 4.29, 4.81, 4.32, 4.43, 4.28],
[0.20, 0.16, 0.19, 0.16, 0.10, 0.15, 0.21, 0.16, 0.21],
[0.45, 0.50, 0.53, 0.60, 0.48, 0.53, 0.50, 0.49, 0.55],
]
)
X = X.T # transpose so input vectors are along the rows
X = np.c_[X, np.ones(X.shape[0])] # add bias term
beta_hat = np.linalg.lstsq(X, y, rcond=None)[0]
print(beta_hat)
```

Result:

```
[ -0.49104607  0.83271938  0.0860167   0.1326091   6.
   ↪ 85681762  22.98163883 -41.
   ↪ 08437805 -19.08085066]
```

You can see the estimated output with:

```
print(np.dot(X,beta_hat))
```

Result:

```
[ -5.97751163, -5.06465759, -10.16873217, -4.96959788, -7.
   ↪ 96356915, -3.06176313, -6.
   ↪ 01818435, -7.90878145, -7.
   ↪ 86720264]
```

## Multiple regression using `scipy` package

The function `scipy.optimize.curve_fit()` can be used for non-linear least squares to fit a function,  $f$ , to data. With `method='lm'`, the algorithm uses the *Levenberg-Marquardt* algorithm through `leastsq`. Note that this algorithm can only deal with unconstrained problems. Box constraints can be handled by methods “`trf`” and “`dogbox`”.

```
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
def func(x, a, b, c):
    return a * np.exp(-b * x) + c
```

Define the data to be fit with some noise:

```
xdata = np.linspace(0, 4, 50)
y = func(xdata, 2.5, 1.3, 0.5)
rng = np.random.default_rng()
y_noise = 0.2 * rng.normal(size=xdata.size)
ydata = y + y_noise
plt.plot(xdata, ydata, 'b-', label='data')
```

Fit for the parameters a, b, c of the function func:

```
popt, pcov = curve_fit(func, xdata, ydata)
popt
array([2.56274217, 1.37268521, 0.47427475])
plt.plot(xdata, func(xdata, *popt), 'r-',
         label='fit: a=%5.3f, b=%5.3f, c=%5.3f' % tuple(popt))
```

Constrain the optimization to the region of  $0 \leq a \leq 3, 0 \leq b \leq 1$  and  $0 \leq c \leq 0.5$ :

```
popt, pcov = curve_fit(func, xdata, ydata, bounds=(0, [3., 1.,
                                                       ↪ 0.5]))
popt
array([2.43736712, 1.           , 0.34463856])
plt.plot(xdata, func(xdata, *popt), 'g--',
         label='fit: a=%5.3f, b=%5.3f, c=%5.3f' % tuple(popt))
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

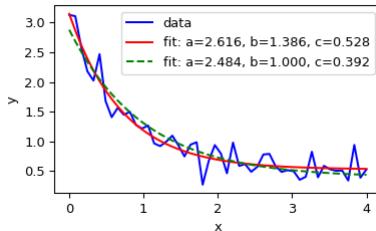


Figure 6.4: Curve fitting using Scipy

## Notes

<sup>53</sup>“Transforming Unstructured Data into Useful Information”, Big Data, Mining, and Analytics, Auerbach Publications, pp. 227–246, 2014-03-12.

<sup>54</sup><sup>62</sup>, Data Analysis Techniques for Physical Scientists, Cambridge University Press, pp. 526–576, 2017.

<sup>55</sup>Xia, B. S., Gong, P. (2015). Review of business intelligence through data analysis. *Benchmarking*, 21(2), 300-311.

<sup>56</sup>“Data Coding and Exploratory Analysis (EDA) Rules for Data Coding Exploratory Data Analysis (EDA) Statistical Assumptions”, SPSS for Intermediate Statistics, Routledge, pp. 42–67, 2004-08-16,

<sup>57</sup>Goodnight, James (2011-01-13).

”The forecast for predictive analytics: hot and getting hotter”. Statistical Analysis and Data Mining: The ASA Data Science Journal. 4 (1): 9–10.

<sup>58</sup>Kachigan, Sam Kash (1986). Statistical analysis : an interdisciplinary introduction to univariate multivariate methods. New York: Radius Press. ISBN 0-942154-99-1.

<sup>59</sup>Lacke, Prem S. Mann ; with the help of Christopher Jay (2010). Introductory statistics (7th ed.). Hoboken, NJ: John Wiley Sons. ISBN 978-0-470-44466-5.

<sup>60</sup>Visit <https://docs.python.org/3/library/statistics.html> for more details.

<sup>61</sup>Visit [https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.fisher\\_exact.html](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.fisher_exact.html) for more information.

## Tasks

1. Visit <https://docs.scipy.org/doc/scipy/>. Read the material and answer the following.
  - (a)



# Chapter 7

## pandas

pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license. The name is derived from the term “panel data”, an econometrics term for data sets that include observations over multiple time periods for the same individuals. Its name is a play on the phrase “Python data analysis” itself. Wes McKinney started building what would become pandas at AQR Capital while he was a researcher there from 2007 to 2010.

Pandas is mainly used for data analysis and associated manipulation of tabular data in DataFrames. Pandas allows importing data from various file formats such as comma-separated values, JSON, Parquet, SQL database tables or queries, and Microsoft Excel. Pandas allows various data manipulation operations such as merging, reshaping, selecting, as well as data cleaning, and data wrangling features. The development of pandas introduced into Python many comparable features of working with DataFrames that were established in the R programming language. The pandas library is built upon another library NumPy, which is oriented to efficiently working with arrays instead of the features of working on DataFrames.

## 7.1 Data import and export

pandas is a widely-used Python library for data science, analysis, and machine learning that offers a flexible and intuitive way to handle data sets of all sizes. One of the most important functionalities of pandas is the tools it provides for reading and writing data. For data available in a tabular format and stored as a CSV file, you can use pandas to read it into memory using the `read_csv()` function, which returns a pandas dataframe. But there are other functionalities too. For example, you can use pandas to perform merging, reshaping, joining, and concatenation operations.

In this article, you will learn about the `read_csv()` function and how you can alter the parameters to customize the output you receive once the function is executed. We will also cover the different methods available to a pandas dataframe object, including how to write pandas dataframe to a CSV file and how to quickly learn more about your data through various methods and attributes.

Before reading a CSV file into a pandas dataframe, you should have some insight into what the data contains. Thus, it's recommended you skim the file before attempting to load it into memory: this will give you more insight into what columns are required and which ones can be discarded. Let's write some code to import a file using `read_csv()`. Then we can talk about what's going on and how we can customize the output we receive while reading the data into memory.

```
import pandas as pd

# Read the CSV file
airbnb_data = pd.read_csv("data/listings_austin.csv")

# View the first 5 rows
airbnb_data.head()
```

All that has gone on in the code above is we have:

1. Imported the pandas library into our environment
2. Passed the filepath to `read_csv()` to read the data into memory as a pandas `dataframe`.  
Pr

But there's a lot more to the `read_csv()` function.

### 7.1.1 Setting a column as the index

The default behavior of pandas is to add an initial index to the dataframe returned from the CSV file it has loaded into memory. However, you can explicitly specify what column to make as the index to the `read_csv` function by setting the `index_col` parameter. Note the value you assign to `index_col` may be given as either a string name, column index or a sequence of str

```
# Setting the id column as the index
airbnb_data = pd.read_csv("data/listings_austin.csv", index_col
                           ↪ ="id")
# airbnb_data = pd.read_csv("data/listings_austing.csv",
                           ↪ index_col=0)

# Preview first 5 rows
airbnb_data.head()
```

### 7.1.2 Selecting specific columns to read into memory

What if you only want to read specific columns into memory because not all of them are important? This is a common scenario that occurs in the real world. Using the `read_csv` function, you can select only the columns you need after like object to the `usecols` parameter of the `read_csv` function.

```
# Defining the columns to read
usecols = ["id", "name", "host_id", "neighbourhood", "room_type
                           ↪ ", "price", "minimum_nights"
                           ↪ ]

# Read data with subset of columns
airbnb_data = pd.read_csv("data/listings_austin.csv", index_col
                           ↪ ="id", usecols=usecols)

# Preview first 5 rows
airbnb_data.head()
```

### 7.1.3 Reading Data from a URL

Once you know how to read a CSV file from local storage into memory, reading data from other sources is a breeze. It's ultimately the same process, except that you're no longer passing a file path. Let's say

there's data you want from a specific webpage; how would you read it into memory? We will use the Iris dataset from the UCI repository as an example:

```
# Webpage URL
url = "https://archive.ics.uci.edu/ml/machine-learning-
      ↪ databases/iris/iris.data"

# Define the column names
col_names = ["sepal_length_in_cm",
             "sepal_width_in_cm",
             "petal_length_in_cm",
             "petal_width_in_cm",
             "class"]

# Read data from URL
iris_data = pd.read_csv(url, names=col_names)

iris_data.head()
```

You may have noticed we assigned a list of strings to the names parameter in the `read_csv` function. This is just so we can rename the column headers while reading the data into memory.

## 7.2 Summary statsitics

The most common object in the pandas library is, by far, the dataframe object. It's a 2-dimensional labeled data structure consisting of rows and columns that may be of different data types (i.e., float, numeric, categorical, etc.). Conceptually, you can think of a pandas dataframe like a spreadsheet, SQL table, or a dictionary of series objects, whichever you're more familiar with. The cool thing about the pandas dataframe is that it comes with many methods that make it easy for you to become acquainted with your data as quickly as possible.

You have already seen one of those methods: `iris_data.head()`, which shows the first  $n$  (the default is 5) rows. The “opposite” method of `head()` is `tail()`, which shows the last  $n$  (5 by default) rows of the dataframe object. For example:

```
iris_data.tail()
```

You can quickly discover the column names by using the columns attribute on your dataframe object:

```
# Discover the column names
iris_data.columns
```

Another important method you can use on your dataframe object is info(). This method prints out a concise summary of the dataframe, including information about the index, data types, columns, non-null values, and memory usage.

```
# Get summary information of the dataframe
iris_data.info()

"""
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype  
 ---  --  
 0   sepal_length_in_cm  150 non-null   float64 
 1   sepal_width_in_cm   150 non-null   float64 
 2   petal_length_in_cm  150 non-null   float64 
 3   petal_width_in_cm   150 non-null   float64 
 4   class              150 non-null   object  
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
"""
```

DataFrame.describe() generates descriptive statistics, including those that summarize the central tendency, dispersion, and shape of the dataset's distribution. If your data has missing values, don't worry; they are not included in the descriptive statistics. Let's call the describe method on the Iris dataset:

```
# Get descriptive statistics
iris_data.describe()
```

### 7.2.1 Exporting the DataFrame to a CSV File

Another method available to pandas dataframe objects is to\_csv(). When you have cleaned and preprocessed your data, the next step may be to export the dataframe to a file - this is pretty straightforward:

```
# Export the file to the current working directory
iris_data.to_csv("cleaned_iris_data.csv")
```

Executing this code will create a CSV in the current working directory called *cleaned\_iris\_data.csv*. But what if you want to use a different delimiter to mark the beginning and end of a unit of data or you wanted to specify how your missing values should be represented? Maybe you don't want the headers to be exported to the file. Well, you can adjust the parameters of the *to\_csv()* method to suit your requirements for the data you want to export. Let's take a look at a few examples of how you can adjust the output of *to\_csv()*.

3. Export data to the current working directory but using a tab delimiter.

```
# Change the delimiter to a tab
iris_data.to_csv("tab_seperated_iris_data.csv", sep="\t")
```

2. Exporting data without the index

```
# Export data without the index
iris_data.to_csv("tab_seperated_iris_data.csv", sep="\t")

# If you get UnicodeEncodeError use this...
# iris_data.to_csv("tab_seperated_iris_data.csv", sep="\t",
#                  index=False,
#                  encoding='utf-8')
```

3. Change the name of missing values

```
# Replace missing values with "Unknown"
iris_data.to_csv("tab_seperated_iris_data.csv", sep="\t",
                 na_rep="Unknown")
```

4. Export dataframe to file without headers (column names)

```
# Do not include headers when exporting the data
iris_data.to_csv("tab_seperated_iris_data.csv", sep="\t",
                 na_rep="Unknown",
                 header=False)
```

## 7.3 Inferential statistics

Inferential statistical analysis, also known as Statistical inference, is the process of using data analysis to infer properties of an underlying distribution of probability.<sup>63</sup> Inferential statistical analysis infers properties of a population, for example by testing hypotheses and deriving estimates. It is assumed that the observed data set is sampled from a larger population.

Inferential statistics can be contrasted with descriptive statistics. Descriptive statistics is solely concerned with properties of the observed data, and it does not rest on the assumption that the data come from a larger population. In machine learning, the term inference is sometimes used instead to mean “make a prediction, by evaluating an already trained model”; in this context inferring properties of the model is referred to as training or learning (rather than inference), and using a model for prediction is referred to as inference (instead of prediction); see also predictive inference.

Statistical inference makes propositions about a population, using data drawn from the population with some form of sampling. Given a hypothesis about a population, for which we wish to draw inferences, statistical inference consists of (first) selecting a statistical model of the process that generates the data and (second) deducing propositions from the model.<sup>64</sup> The conclusion of a statistical inference is a statistical proposition. Some common forms of statistical proposition are the following:

1. A point estimate, i.e. a particular value that best approximates some parameter of interest;
2. An interval estimate, e.g. a confidence interval (or set estimate), i.e. an interval constructed using a dataset drawn from a population so that, under repeated sampling of such datasets, such intervals would contain the true parameter value with the probability at the stated confidence level;
3. A credible interval, i.e. a set of values containing, for example, 95% of posterior belief; rejection of a hypothesis;

### 7.3.1 Models and assumptions

Any statistical inference requires some assumptions. A statistical model is a set of assumptions concerning the generation of the observed data and similar data. Descriptions of statistical models usually emphasize the role of population quantities of interest, about which we wish to draw inference. Descriptive statistics are typically used as a preliminary step before more formal inferences are drawn.<sup>65</sup>

#### Degree of models/assumptions

Statisticians distinguish between three levels of modeling assumptions:

1. *Fully parametric*: The probability distributions describing the data-generation process are assumed to be fully described by a family of probability distributions involving only a finite number of unknown parameters. For example, one may assume that the distribution of population values is truly Normal, with unknown mean and variance, and that datasets are generated by 'simple' random sampling. The family of generalized linear models is a widely used and flexible class of parametric models.
2. *Non-parametric*: The assumptions made about the process generating the data are much less than in parametric statistics and may be minimal.<sup>66</sup> For example, every continuous probability distribution has a median, which may be estimated using the sample median or the Hodges–Lehmann–Sen estimator, which has good properties when the data arise from simple random sampling.
3. *Semi-parametric*: This term typically implies assumptions “in between” fully and non-parametric approaches. For example, one may assume that a population distribution has a finite mean. Furthermore, one may assume that the mean response level in the population depends in a truly linear manner on some covariate (a parametric assumption) but not make any parametric assumption describing the variance around that mean (i.e. about the presence or possible form of any heteroscedasticity). More generally, semi-parametric models can often be separated into “structural” and “random variation” components. One compo-

nent is treated parametrically and the other non-parametrically. The well-known Cox model is a set of semi-parametric assumptions.

### 7.3.2 T Test

The following examples show how to perform three different t-tests using a pandas DataFrame:

1. Independent Two Sample t-Test
2. Welch's Two Sample t-Test
3. Paired Samples t-Test

#### Independent Two Sample t-Test

An independent two sample t-test is used to determine if two population means are equal. For example, suppose a professor wants to know if two different studying methods lead to different mean exam scores. To test this, he recruits 10 students to use method A and 10 students to use method B. The following code shows how to enter the scores of each student in a pandas DataFrame and then use the `ttest_ind()` function from the SciPy library to perform an independent two sample t-test:

```
import pandas as pd
from scipy.stats import ttest_ind

#create pandas DataFrame
df = pd.DataFrame({'method': ['A', 'A', 'A', 'A', 'A', 'A', 'A',
                               'A', 'A', 'A', 'B', 'B', 'B', 'B', 'B', 'B',
                               'B', 'B', 'B', 'B'],
                    'score': [71, 72, 72, 75, 78, 81, 82, 83, 89,
                              91, 80, 81, 81, 84, 88, 88,
                              89, 90, 90, 91]})

#view first five rows of DataFrame
df.head()

   method  score
0       A      71
1       A      72
```

```

2      A    72
3      A    75
4      A    78

#define samples
group1 = df[df['method']=='A']
group2 = df[df['method']=='B']

#perform independent two sample t-test
ttest_ind(group1['score'], group2['score'])

Ttest_indResult(statistic=-2.6034304605397938, pvalue=0.
                ↪ 017969284594810425)

```

From the output we can see: t test statistic: -2.6034; p-value: 0.0179. Since the p-value is less than .05, we reject the null hypothesis of the t-test and conclude that there is sufficient evidence to say that the two methods lead to different mean exam scores.

### Welch's t-Test in Pandas

Welch's t-test is similar to the independent two sample t-test, except it does not assume that the two populations that the samples came from have equal variance. To perform Welch's t-test on the exact same dataset as the previous example, we simply need to specify `equal_var=False` within the `ttest_ind()` function as follows:

```

import pandas as pd
from scipy.stats import ttest_ind

#create pandas DataFrame
df = pd.DataFrame({'method': ['A', 'A', 'A', 'A', 'A', 'A', 'A',
                               ↪ , 'A', 'A', 'A', 'B', 'B', 'B',
                               ↪ B', 'B', 'B', 'B', 'B', 'B',
                               ↪ 'B', 'B'], 'score': [71, 72,
                               ↪ 72, 75, 78, 81, 82, 83, 89,
                               ↪ 91, 80, 81, 81, 84, 88, 88,
                               ↪ 89, 90, 90, 91]})

#define samples
group1 = df[df['method']=='A']
group2 = df[df['method']=='B']

#perform Welch's t-test
ttest_ind(group1['score'], group2['score'], equal_var=False)

```

```
Ttest_indResult(statistic=-2.603430460539794, pvalue=0.  
                           ↪ 02014688617423973)
```

From the output we can see: t test statistic: -2.6034; p-value: 0.0201. Since the p-value is less than .05, we reject the null hypothesis of Welch's t-test and conclude that there is sufficient evidence to say that the two methods lead to different mean exam scores.

### 7.3.3 Paired Samples t-Test in Pandas

A paired samples t-test is used to determine if two population means are equal in which each observation in one sample can be paired with an observation in the other sample. For example, suppose a professor wants to know if two different studying methods lead to different mean exam scores. To test this, he recruits 10 students to use method A and then take a test. Then, he lets the same 10 students used method B to prepare for and take another test of similar difficulty. Since all of the students appear in both samples, we can perform a paired samples t-test in this scenario.

The following code shows how to enter the scores of each student in a pandas DataFrame and then use the `ttest_rel()` function from the SciPy library to perform a paired samples t-test:

```
import pandas as pd  
from scipy.stats import ttest_rel  
  
#create pandas DataFrame  
df = pd.DataFrame({'method': ['A', 'A', 'A', 'A', 'A', 'A', 'A',  
                           ↪ , 'A', 'A', 'A', 'B', 'B', 'B',  
                           ↪ B', 'B', 'B', 'B', 'B', 'B',  
                           ↪ 'B', 'B'], 'score': [71, 72,  
                           ↪ 72, 75, 78, 81, 82, 83, 89,  
                           ↪ 91, 80, 81, 81, 84, 88, 88,  
                           ↪ 89, 90, 90, 91]})  
  
#view first five rows of DataFrame  
df.head()  
  
   method  score  
0        A      71  
1        A      72
```

```

2      A      72
3      A      75
4      A      78

#define samples
group1 = df[df['method']=='A']
group2 = df[df['method']=='B']

#perform independent two sample t-test
ttest_rel(group1['score'], group2['score'])

Ttest_relResult(statistic=-6.16204531967805, pvalue=0.
                ↪ 0001662872100210469)

```

From the output we can see: t test statistic: -6.1620; p-value: 0.0001; Since the p-value is less than .05, we reject the null hypothesis of the paired samples t-test and conclude that there is sufficient evidence to say that the two methods lead to different mean exam scores.

### 7.3.4 One-Way ANOVA

A one-way ANOVA (“analysis of variance”) is used to determine whether or not there is a statistically significant difference between the means of three or more independent groups.

For example, a researcher recruits 30 students to participate in a study. The students are randomly assigned to use one of three studying techniques for the next three weeks to prepare for an exam. At the end of the three weeks, all of the students take the same test. Use the following steps to perform a one-way ANOVA to determine if the average scores are the same across all three groups.

- Step 1: Enter the data. First, we'll enter the exam scores for each group into three separate arrays:

```

#enter exam scores for each group
group1 = [85, 86, 88, 75, 78, 94, 98, 79, 71, 80]
group2 = [91, 92, 93, 85, 87, 84, 82, 88, 95, 96]
group3 = [79, 78, 88, 94, 92, 85, 83, 85, 82, 81]

```

- Step 2: Perform the one-way ANOVA. Next, we'll use the `f_oneway()` function from the SciPy library to perform the one-way ANOVA:

```
from scipy.stats import f_oneway

#perform one-way ANOVA
f_oneway(group1, group2, group3)

(statistic=2.3575, pvalue=0.1138)
```

### 3. Step 3: Interpret the results.

A one-way ANOVA uses the following null and alternative hypotheses:

- $H_0$  (null hypothesis):  $\mu_1 = \mu_2 = \mu_3 = \dots = \mu_k$  (all the population means are equal)
- $H_0$  (null hypothesis): at least one population mean is different from the rest

The F test statistic is 2.3575 and the corresponding p-value is 0.1138. Since the p-value is not less than .05, we fail to reject the null hypothesis. This means we do not have sufficient evidence to say that there is a difference in exam scores among the three studying techniques.

#### 7.3.5 Chi-Square Test

A Chi-Square Test of Independence is used to determine whether or not there is a significant association between two categorical variables. This tutorial explains how to perform a Chi-Square Test of Independence in Python.

For Example, suppose we want to know whether or not gender is associated with political party preference. We take a simple random sample of 500 voters and survey them on their political party preference. The following table shows the results of the survey:

Use the following steps to perform a Chi-Square Test of Independence in Python to determine if gender is associated with political party preference.

	Republican	Democrat	Independent	Total
Male	120	90	40	250
Female	110	95	45	250
Total	230	185	85	500

Table 7.1: Sample data for Chi-Square Test

- Step 1: Create the data. First, we will create a table to hold our data:

```
data = [[120, 90, 40],
        [110, 95, 45]]
```

- Step 2: Perform the Chi-Square Test of Independence. Next, we can perform the Chi-Square Test of Independence using the `chi2_contingency` function from the SciPy library, which uses the syntax: `chi2_contingency(observed)`, where *observed* is a contingency table of observed values. The following code shows how to use this function in our specific example:

```
import scipy.stats as stats

#perform the Chi-Square Test of Independence
stats.chi2_contingency(data)

(0.864,
 0.649,
 2,
 array([[115., 92.5, 42.5],
        [115., 92.5, 42.5]]))
```

The way to interpret the output is as follows:

- Chi-Square Test Statistic: 0.864
- p-value: 0.649
- Degrees of freedom: 2 (calculated as rows-1 \* columns-1)
- Array: The last array displays the expected values for each cell in the contingency table.

Recall that the Chi-Square Test of Independence uses the following null and alternative hypotheses:

- $H_0$ : (null hypothesis) The two variables are independent.
- $H_1$ : (alternative hypothesis) The two variables are not independent.

Since the p-value (.649) of the test is not less than 0.05, we fail to reject the null hypothesis. This means we do not have sufficient evidence to say that there is an association between gender and political party preference. In other words, gender and political party preference are independent.

### 7.3.6 Correlations

In statistics, correlation or dependence is any statistical relationship, whether causal or not, between two random variables or bivariate data. Although in the broadest sense, “correlation” may indicate any type of association, in statistics it usually refers to the degree to which a pair of variables are linearly related. In Python; Pearson, Kendall and Spearman correlation are currently computed using pairwise complete observations.

- Pearson correlation coefficient: Pearson correlation coefficient also known as Pearson’s  $r$ , is a measure of linear correlation between two sets of data. It is the ratio between the covariance of two variables and the product of their standard deviations; thus, it is essentially a normalized measurement of the covariance, such that the result always has a value between  $-1$  and  $1$ .
- Kendall rank correlation coefficient: the Kendall rank correlation coefficient, commonly referred to as Kendall’s  $\tau$  coefficient (after the Greek letter  $\tau$ , tau), is a statistic used to measure the ordinal association between two measured quantities. A  $\tau$  test is a non-parametric hypothesis test for statistical dependence based on the  $\tau$  coefficient.
- Spearman’s rank correlation coefficient: Spearman’s rank correlation coefficient or Spearman’s  $\rho$ , named after Charles Spearman and often denoted by the Greek letter  $\rho$  (rho) or as  $r\_s$ , is

a nonparametric measure of rank correlation (statistical dependence between the rankings of two variables). It assesses how well the relationship between two variables can be described using a monotonic function.

### Pearson correlation coefficient

```
>>> from scipy.stats import pearsonr
>>> import numpy as np
>>> x = np.random.random_sample(10)
>>> y = np.random.random_sample(10)
>>> import pandas as pd
>>> df = pd.DataFrame({'x':x, 'y':y})
>>> rho = df.corr()
>>> rho
      x          y
x  1.000000  0.383876
y  0.383876  1.000000
>>> pval = df.corr(method=lambda x, y: pearsonr(x, y)[1]) - np.
               ↪ eye(*rho.shape)
>>> pval
      x          y
x  0.000000  0.273458
y  0.273458  0.000000
```

### Kendall rank correlation coefficient

```
>>> rho = df.corr(method='spearman')
>>> pval = df.corr(method=lambda x, y: spearmanr(x, y)[1]) - np.
               ↪ eye(*rho.shape)
>>> rho
      x          y
x  1.000000  0.418182
y  0.418182  1.000000
>>> pval
      x          y
x  0.000000  0.229113
y  0.229113  0.000000
```

### Spearman's rank correlation coefficient

```
>>> from scipy.stats import kendalltau
>>> rho = df.corr(method='kendall')
```

```
>>> pval = df.corr(method=lambda x, y: kendalltau(x, y)[1]) -
           ↪ np.eye(*rho.shape)
>>> rho
      x          y
x  1.000000  0.333333
y  0.333333  1.000000
>>> pval
      x          y
x  0.000000  0.216373
y  0.216373  0.000000
```

Correlation coefficients and P Values are different for all the three methods for variables  $x, y$ .

### Pandas corwith function

Compute pairwise correlation. Pairwise correlation is computed between rows or columns of DataFrame with rows or columns of Series or DataFrame. DataFrames are first aligned along both axes before computing the correlations.

```
>>> index = ["a", "b", "c", "d", "e"]
>>> columns = ["one", "two", "three", "four"]
>>> df1 = pd.DataFrame(np.arange(20).reshape(5, 4), index=index
           ↪ , columns=columns)
>>> df2 = pd.DataFrame(np.arange(16).reshape(4, 4), index=index
           ↪ [:4], columns=columns)
>>> df1.corrwith(df2)
one      1.0
two      1.0
three    1.0
four     1.0
dtype: float64
>>> df2.corrwith(df1, axis=1)
a      1.0
b      1.0
c      1.0
d      1.0
e      NaN
dtype: float64
```

### 7.3.7 Regression

In this section, we will try to understand visually what Linear Regression is. Then make an example on real data. What is Linear Regression? Linear Regression can be described as (1) given data input (independent variables) can we predict output (dependent variable), (2) it is the mapping from an input point to a continuous value. The goal of Linear Regression is to find the best-fitting line. Hence, some data will be fitted better as it will be closer to the line. The predictions will be on the line. That is when you have fitted your Linear Regression model, it will predict new values to be on the line. While this sounds simple, the model is one of the most used models and creates high value.

Often there is a bit of confusion between Linear Regression and Correlation. But they do different things. Correlation is one number describing a relationship between two variables. While Linear Regression is an equation used to predict values. Correlation is a single measure of the relationship between two variables. Linear Regression is an equation is used for prediction. For instance, it is possible to make a simple scatter plot for some simulated data known as *Weight* and *Height* is as follows.

```
import pandas as pd
data = pd.DataFrame({'Weight': np.abs(np.random.normal(0, 1, 15
    ↪ )), 'Height': np.abs(np.
    ↪ random.normal(0, 1, 15))})
data.head()
    Weight      Height
0   0.788573  0.108496
1   0.469591  0.043912
2   0.716523  0.458724
3   0.795473  1.565468

data.plot.scatter(x='Height', y='Weight')
```

The plot looks like the one shown at 7.1

It is also possible to plot regression line such as below.

```
from sklearn.linear_model import LinearRegression
# Creating a Linear Regression model on our data
```

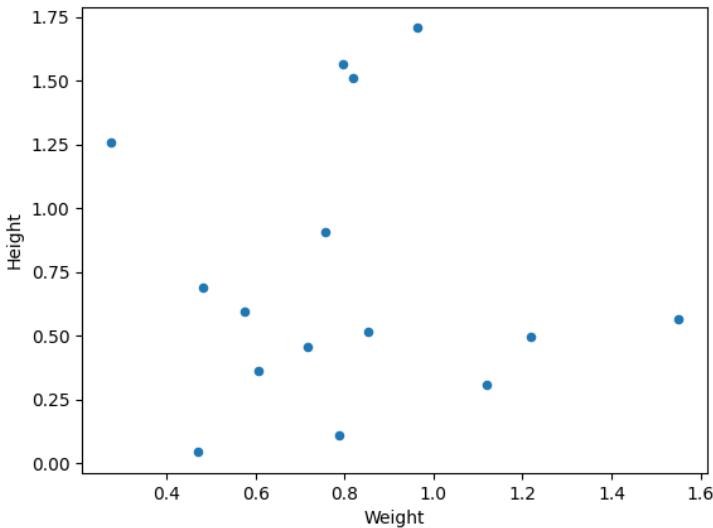


Figure 7.1: Scatter plot for simulated  $x, y$  variables

```

lin = LinearRegression()
lin.fit(data[['Weight']], data['Height'])

# Creating a plot
ax = data.plot.scatter(x='Height', y='Weight', alpha=.1)
ax.plot(data['Weight'], lin.predict(data[['Weight']]), c='r')

```

The plot looks like the one shown at 7.2

Its clear that the relationship is slightly inverse (after all it is simulated data). To measure the accuracy of the prediction the r-squared function is often used, which you can access directly on the model by using the following code.

```

lin.score(data[['Weight']], data['Height'])
0.0022213092860585704

```

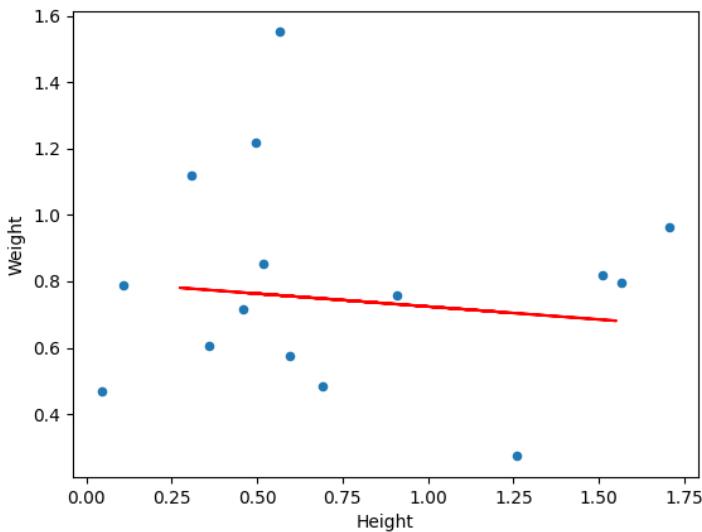


Figure 7.2: Scatter plot for simulated  $x, y$  variables with regression line

Its really very bad score. It is possible to get best possible fit using data simulation techniques. Its fine. However, the concept under discussion is “inferential”, which is all about making inferences. It is possible to try for the P Value through *Scipy*.

```
>>> result = linregress(x, y)
>>> result.intercept, result.slope, result.pvalue, result.
   ↪ rvalue
(0.6188289611363726, 0.29233431161967727, 0.2734578735994691, 0
   ↪ .3838755635595937)
```

The P Value 0.2734578735994691 is for slope but not for intercept. The slope does not seems to be statistically significant. This means the null hypothesis,  $H_0 : \beta_1 = 0$  can not be rejected.

### 7.3.8 Multiple linear regression

Multiple linear regression (MLR) is a multivariate statistical technique for examining the linear correlations between two or more independent variables (IVs) and a single dependent variable (DV). Research questions suitable for MLR can be of the form “To what extent do X<sub>1</sub>, X<sub>2</sub>, and X<sub>3</sub> (IVs) predict Y (DV)?”, or “To what extent does people’s age and gender (IVs) predict their levels of blood cholesterol (DV)?”

#### Level of measurement

1. DV: A normally distributed interval or ratio variable.
2. IVs: Two or more normally distributed interval or ratio variables or dichotomous variables. Note that it may be necessary to recode non-normal interval or ratio IVs or multichotomous categorical or ordinal IVs into dichotomous variables or a series of dummy variables).

#### Sample size

Enough data is needed to provide reliable estimates of the correlations. As the number of IVs increases, more inferential tests are being conducted, therefore more data is needed, otherwise the estimates of the regression line are probably unstable and are unlikely to replicate if the study is repeated.

Some rules of thumb:

1. Use at least 50 cases plus at least 10 to 20 as many cases as there are IVs.
2. Green (1991) and Tabachnick and Fidell (2007):
3.  $50 + 8(k)$  for testing an overall regression model and
4.  $104 + k$  when testing individual predictors (where k is the number of IVs). Based on detecting a medium effect size ( $\beta \geq .20$ ), with critical  $\leq .05$ , with power of 80%.

## Normality

1. Check the univariate descriptive statistics (M, SD, skewness and kurtosis). As a general guide, skewness and kurtosis should be between -1 and +1.
2. Check the histograms with a normal curve imposed.
3. Note: Be wary (i.e., avoid!) using inferential tests of normality (e.g., the Shapiro–Wilk test - they are notoriously overly sensitive for the purposes/needs of regression).
4. Normally distributed variables will enhance the MLR solution. Estimates of correlations will be more reliable and stable when the variables are normally distributed, but regression will be reasonably robust to minor to moderate deviations from non-normal data when moderate to large sample sizes are involved. Also examine scatterplots for bivariate outliers because non-normal univariate data may make bivariate and multivariate outliers more likely.

## Linearity

Check scatterplots between the DV (Y) and each of the IVs (Xs) to determine linearity:

1. Are there any bivariate outliers? If so, consider removing the outliers.
2. Are there any non-linear relationships? If so, consider using a more appropriate type of regression.

## Homoscedasticity

Based on the scatterplots between the IVs and the DV:

1. Are the bivariate distributions reasonably evenly spread about the line of best fit?
2. Also can be checked via the normality of the residuals.

### Multicollinearity

IVs should not be overly correlated with one another. Ways to check:

1. Examine bivariate correlations and scatterplots between each of the IVs (i.e., are the predictors overly correlated - above ~.7?).
2. Check the collinearity statistics in the coefficients table:
  - (a) Various recommendations for acceptable levels of VIF and Tolerance have been published.
  - (b) Variance Inflation Factor (VIF) should be low (< 3 to 10)  
or
  - (c) Tolerance should be high (> .1 to .3)
  - (d) Note that VIF and Tolerance have a reciprocal relationship (i.e.,  $TOL=1/VIF$ ), so only one of the indicators needs to be used.

### Normality of residuals

The residuals should be normally distributed around 0.

1. Residuals are more likely to be normally distributed if each of the variables normally distributed, so check normality first.
2. Histogram: Use histogram for normality check.
3. Normal probability plot

If residuals are not normally distributed, there is probably something wrong with the distribution of one or more variables - re-check.

### Python practice

Now we will use the *scikitlearn* linear regression library to solve the multiple linear regression problem. There are 4 steps to follow to train a machine-learning model to do multiple linear regression. Let's look into each of these steps in detail while applying multiple linear regression on the *50\_startupsdataset*. You can click here to download the dataset.

1. **Step 1:** Reading the Dataset: Most of the datasets are in CSV file format; for reading this file, we use pandas library:

```
>>> import pandas as pd
>>> df = pd.read_csv('50_Startups.csv')
>>> df.head()
   R&D Spend Administration Marketing Spend      State
0    165349.20       136897.80        471784.10  New York
1    162597.70       151377.59        192261.83
2    153441.51       101145.55        443898.53  California
3    144372.41       118671.85        191792.06
4    142107.34       91391.77        407934.54  Florida
                                         ↪ Profit
                                         ↪
                                         ↪
                                         ↪
                                         ↪
                                         ↪
```

Here you can see that there are 5 columns in the dataset where the state stores the categorical data points, and the rest are numerical features. Now, we have to classify independent and dependent features. There are total 5 features in the dataset, of which profit is our dependent feature, and the rest are our independent features.

2. **Step 2:** Handling Categorical Variables

In our dataset, there is one categorical column, State. We must handle the categorical values inside this column as part of data preprocessing. For that, we will use pandas `get_dummies()` function:

```
# handle categorical variable
>>> states=pd.get_dummies(df.State,drop_first=True)
>>> states.head()
   Florida  New York
0         0         1
1         0         0
2         1         0
3         0         1

# dropping extra column
df= df.drop('State',axis=1)
```

```
# concatenation of independent variables and new
#                                     ↪ categorical variable.
>>> df=pd.concat([df,states],axis=1)
>>> df.head()
   R&D Spend Administration Marketing Spend      Profit
0  165349.20     136897.80          471784.10    Florida  New York
   ↪ 0           ↪ 1
1  162597.70     151377.59          443898.53  191792.06
   ↪ 0           ↪ 0
2  153441.51     101145.55          407934.54  191050.39
   ↪ 1           ↪ 0
3  144372.41     118671.85          383199.62  182901.99
   ↪ 0           ↪ 1
4  142107.34     91391.77          366168.42  166187.94
   ↪ 1           ↪ 0
```

### 3. Step 3: Splitting the Data

Now, we have to split the data into training and test sets using the scikit-learn `train_test_split()` function.

```
# importing train_test_split from sklearn
from sklearn.model_selection import train_test_split

# splitting the data
x_train, x_test, y_train, y_test = train_test_split(x, y,
                                                    test_size = 0.2,
                                                    random_state = 42)
```

### 4. Step 4: Applying the Model

Now, we apply the linear regression model to our training data. First of all, we have to import linear regression from the scikit-learn library. Unlike linear regression, there is no other library to implement multiple linear regression.

```
#Preparing data for train-test-split operation
x = df.drop(['Profit'], axis=1)
x.columns
Index(['R&D Spend', 'Administration', 'Marketing Spend', 'Florida',
       'New York'],
      dtype='object')
y = df['Profit']
```

```
y.head()
0    192261.83
1    191792.06
2    191050.39
3    182901.99
4    166187.94
Name: Profit, dtype: float64
```

## 5. Step 4: Applying the Model

Now, we apply the linear regression model to our training data. First of all, we have to import linear regression from the scikit-learn library. Unlike linear regression, there is no other library to implement multiple linear regression.

```
# importing module
from sklearn.linear_model import LinearRegression

# creating an object of LinearRegression class
lm = LinearRegression()

# fitting the training data
lm.fit(x_train,y_train)

# predictions
y_prediction = lm.predict(x_test)

# importing r2_score module
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error

# predicting the accuracy score
score=r2_score(y_test,y_prediction)
print('r2 score is ',score) # 0.8987266414328636
print('mean_sqrd_error is ==',mean_squared_error(y_test,
                                                y_prediction)) #
                                                # 82010363.04430111
print('root_mean_squared error of is ==',np.sqrt(
                                                mean_squared_error(
                                                y_test,y_prediction)))
                                                # 9055.957323458471
```

You can see that the accuracy score is greater than 0.8, which means we can use this model to solve multiple linear regression, and also mean squared error rate is also low.

## 7.4 Exploratory statistics

In statistics, exploratory data analysis (EDA) is an approach of analyzing data sets to summarize their main characteristics, often using statistical graphics and other data visualization methods. A statistical model can be used or not, but primarily EDA is for seeing what the data can tell us beyond the formal modeling and thereby contrasts traditional hypothesis testing. Exploratory data analysis has been promoted by John Tukey since 1970 to encourage statisticians to explore the data, and possibly formulate hypotheses that could lead to new data collection and experiments. EDA is different from initial data analysis (IDA), which focuses more narrowly on checking assumptions required for model fitting and hypothesis testing, and handling missing values and making transformations of variables as needed. EDA encompasses IDA.<sup>6768</sup>

Tukey defined data analysis in 1961 as: “Procedures for analyzing data, techniques for interpreting the results of such procedures, ways of planning the gathering of data to make its analysis easier, more precise or more accurate, and all the machinery and results of (mathematical) statistics which apply to analyzing data.”<sup>69</sup>

John W. Tukey wrote the book *Exploratory Data Analysis* in 1977. Tukey held that too much emphasis in statistics was placed on statistical hypothesis testing (confirmatory data analysis); more emphasis needed to be placed on using data to suggest hypotheses to test. In particular, he held that confusing the two types of analyses and employing them on the same set of data can lead to systematic bias owing to the issues inherent in testing hypotheses suggested by the data.

The objectives of EDA are to:

1. Enable unexpected discoveries in the data
2. Suggest hypotheses about the causes of observed phenomena
3. Assess assumptions on which statistical inference will be based
4. Support the selection of appropriate statistical tools and techniques

5. Provide a basis for further data collection through surveys or experiments

Many EDA techniques have been adopted into data mining. They are also being taught to young students as a way to introduce them to statistical thinking.<sup>70</sup>

### 7.4.1 Techniques and tools

There are a number of tools that are useful for EDA, but EDA is characterized more by the attitude taken than by particular techniques.<sup>71</sup> EDA techniques can be classified under three categories (1) graphical techniques; such as Box plot, Histogram, Pareto chart, etc., (2) quantitative techniques; such as Median polish, Trimean and Ordination, etc., (3) Dimensionality reduction techniques; such as below:

1. Multidimensional scaling
2. Principal component analysis (PCA)
3. Multilinear PCA
4. Nonlinear dimensionality reduction (NLDR)
5. Iconography of correlations
6. and many more ...

We will discuss PCA, Factor Analysis and Cluster Analysis in this section.

### 7.4.2 Principal component analysis (PCA)

Principal component analysis (PCA) is a method of reducing the dimensionality of data and is used to improve data visualization and speed up machine learning model training. To understand the value of using PCA for data visualization, the first part of this tutorial post goes over a basic visualization of the Iris data set after applying PCA. The second part, explores how to use PCA to speed up a machine learning algorithm (logistic regression) on the Modified National Institute of Standards and Technology (MNIST) data set.

For a lot of machine learning applications, it helps to visualize your data. Visualizing two- or three-dimensional data is not that challenging. However, even the Iris data set used in this part of the tutorial is four-dimensional. You can use PCA to reduce that four-dimensional data into two or three dimensions so that you can plot, and hopefully, understand the data better.

### 1. STEP 1: Load the data

The iris data set comes with scikit-learn and doesn't require you to download any files from some external websites. The code below will load the Iris data set.

```
import pandas as pd
>>> url = "https://archive.ics.uci.edu/ml/machine-
    ↪ learning-databases/iris
    ↪ /iris.data"
>>> df = pd.read_csv(url, names=['sepal length','sepal
    ↪ width','petal length',
    ↪ petal width','target'])
>>> df.head()
   sepal length  sepal width  petal length  petal width
   ↪           target
0            5.1          3.5          1.4          0.2
   ↪ Iris-setosa
1            4.9          3.0          1.4          0.2
   ↪ Iris-setosa
2            4.7          3.2          1.3          0.2
   ↪ Iris-setosa
3            4.6          3.1          1.5          0.2
   ↪ Iris-setosa
4            5.0          3.6          1.4          0.2
   ↪ Iris-setosa
```

### 2. STEP 2: Standardize The Data

PCA is affected by scale, so you need to scale the features in your data before applying PCA. Use `StandardScaler` to help you standardize the data set's features onto unit scale (mean = 0 and variance = 1), which is a requirement for the optimal performance of many machine learning algorithms. If you don't scale your data, it can have a negative effect on your algorithm.

```
>>> from sklearn.preprocessing import StandardScaler
>>> features = ['sepal length', 'sepal width', 'petal
   ↪ length', 'petal width']
>>> x = df.loc[:, features].values
>>> type(x)
<class 'numpy.ndarray'>
>>> x[1:10]
array([[4.9,  3. ,  1.4,  0.2],
       [4.7,  3.2,  1.3,  0.2],
       [4.6,  3.1,  1.5,  0.2],
       [5. ,  3.6,  1.4,  0.2],
       [5.4,  3.9,  1.7,  0.4],
       [4.6,  3.4,  1.4,  0.3],
       [5. ,  3.4,  1.5,  0.2],
       [4.4,  2.9,  1.4,  0.2],
       [4.9,  3.1,  1.5,  0.1]])
>>> y = df.loc[:,['target']].values
>>> x = StandardScaler().fit_transform(x)
>>> type(x)
<class 'numpy.ndarray'>
>>> x[1:10]
SyntaxError: closing parenthesis ')' does not match
   ↪ opening parenthesis '['
>>> x[1:10]
array([[-1.14301691, -0.1249576 , -1.3412724 , -1.31297673
   ↪ ],
       [-1.38535265,  0.33784833, -1.39813811, -1.31297673
   ↪ ],
       [-1.50652052,  0.10644536, -1.2844067 , -1.31297673
   ↪ ],
       [-1.02184904,  1.26346019, -1.3412724 , -1.31297673
   ↪ ],
       [-0.53717756,  1.95766909, -1.17067529, -1.05003079
   ↪ ],
       [-1.50652052,  0.80065426, -1.3412724 , -1.18150376
   ↪ ],
       [-1.02184904,  0.80065426, -1.2844067 , -1.31297673
   ↪ ],
       [-1.74885626, -0.35636057, -1.3412724 , -1.31297673
   ↪ ],
       [-1.14301691,  0.10644536, -1.2844067 , -1.44444497
   ↪ ]])
```

3. STEP 3: PCA Projection to 2D The original data has four columns (sepal length, sepal width, petal length and petal width).

In this section, the code projects the original data, which is four-dimensional, into two dimensions. After dimensionality reduction, there usually isn't a particular meaning assigned to each principal component. The new components are just the two main dimensions of variation.

```

from sklearn.decomposition import PCA
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(x)

principalDf = pd.DataFrame(data = principalComponents
                            , columns = ['principal component 1',
                                         'principal
                                         component
                                         2'])

finalDf = pd.concat([principalDf, df[['target']]], axis =
                     1)

type(finalDf)
finalDf.head()
   PC 1      PC 2      target
0 -2.264542  0.505704  Iris-setosa
1 -2.086426 -0.655405  Iris-setosa
2 -2.367950 -0.318477  Iris-setosa
3 -2.304197 -0.575368  Iris-setosa
4 -2.388777  0.674767  Iris-setosa

```

#### 4. STEP 4: Visualize 2D projection

This section is just plotting two-dimensional data. Notice on the graph below that the classes seem well separated from each other.

```

>>> import matplotlib.pyplot as plt
>>> fig = plt.figure(figsize = (8,8))
>>> ax = fig.add_subplot(1,1,1)

>>> ax.set_xlabel('Principal Component 1', fontsize = 15)
Text(0.5, 0, 'Principal Component 1')

>>> ax.set_ylabel('Principal Component 2', fontsize = 15)
Text(0, 0.5, 'Principal Component 2')

>>> ax.set_title('2 component PCA', fontsize = 20)

```

```
Text(0.5, 1.0, '2 component PCA')

>>> targets = ['Iris-setosa', 'Iris-versicolor', 'Iris-
    ↪ virginica']
>>> colors = ['r', 'g', 'b']

>>> for target, color in zip(targets,colors):
...     indicesToKeep = finalDf['target'] == target
...     ax.scatter(finalDf.loc[indicesToKeep, 'PC 1']
...                 , finalDf.loc[indicesToKeep, 'PC 2']
...                 , c = color
...                 , s = 50)

>>> ax.legend(targets)
>>> ax.grid()
>>> plt.show()
```

## 5. Step 5: Explained variance

The explained variance tells you how much information (variance) can be attributed to each of the principal components. This is important because while you can convert four-dimensional space to a two-dimensional space, you lose some of the variance (information) when you do this. By using the attribute `explained_variance_ratio_`, you can see that the first principal component contains 72.77 percent of the variance, and the second principal component contains 23.03 percent of the variance. Together, the two components contain 95.80 percent of the information.

```
>>> pca.explained_variance_ratio_
array([0.72770452, 0.23030523])
```

## Tasks

1. Simulate bivariate and multivariate data sets using Numpy such a way that the variables of the dataset is closely correlated. Try to fit regression and make scatter plots. You may use `np.linalg.cholesky()`, `np.random.standard_normal()` for better results.
2. Study the following fit measures for multiple regression
  - (a)  $SS_{res}$
  - (b)  $SS_{mean}$
  - (c)  $R^2$
  - (d) MSE
  - (e) RMSE
  - (f) F Statistic
3. The MNIST database of handwritten digits is more suitable, as it has 784 feature columns (784 dimensions), a training set of 60,000 examples and a test set of 10,000 examples. Try to perform PCA on the aforementioned data set using scikitlearn package. You may follow below steps.
  - (a) Download and load the data  
(Use `fetch_openml` in `sklearn` package.)
  - (b) Split data into training and test sets (Use `train_test_split` from `sklearn` package)
  - (c) Standardize the data  
(Use `StandardScaler` from `sklearn` package)
  - (d) Apply PCA (Use `PCA` class available in `sklearn` package)

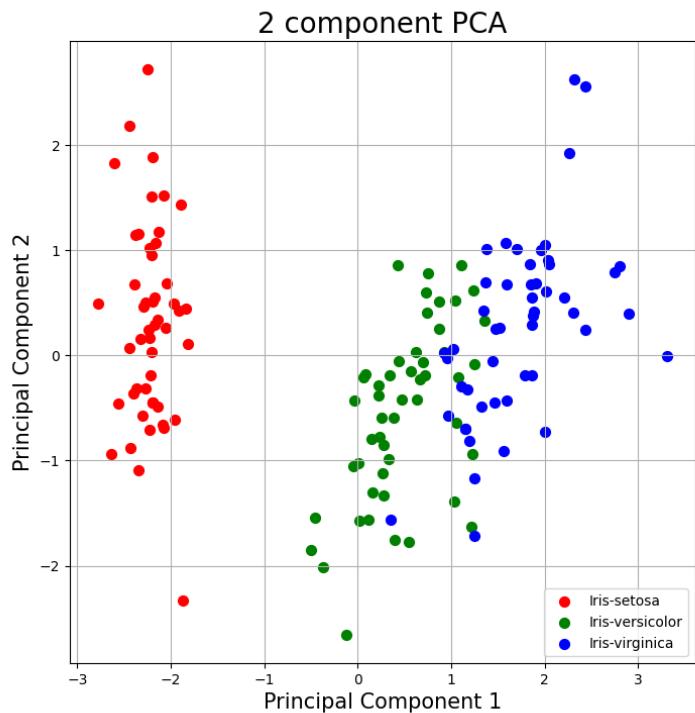


Figure 7.3: PCA graph for Iris data

# Chapter 8

## Visualization

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension *NumPy*. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like *Tkinter*, *wxPython*, *Qt*, or *GTK*. There is also a procedural "pylab" interface based on a state machine (like *OpenGL*), designed to closely resemble that of MATLAB, though its use is discouraged. *SciPy* makes use of *Matplotlib*.

Matplotlib was originally written by *John D. Hunter*. Since then it has had an active development community and is distributed under a BSD-style license. *Michael Droettboom* was nominated as matplotlib's lead developer shortly before John Hunter's death in August 2012 and was further joined by *Thomas Caswell*. Matplotlib is a *NumFOCUS* fiscally sponsored project. Matplotlib 2.0.x supports Python versions 2.7 through 3.10. Python 3 support started with Matplotlib 1.2. Matplotlib 1.4 is the last version to support Python 2.6. Matplotlib has pledged not to support Python 2 past 2020 by signing the Python 3 Statement.

## 8.1 A simple example

Matplotlib graphs your data on Figures (e.g., windows, Jupyter widgets, etc.), each of which can contain one or more *Axes*, an area where points can be specified in terms of x-y coordinates (or theta-r in a polar plot, x-y-z in a 3D plot, etc.). The simplest way of creating a Figure with an Axes is using `pyplot.subplots`. We can then use `Axes.plot` to draw some data on the Axes:

```
fig, ax = plt.subplots() # Create a figure containing a single
                         # ↪ axes.
ax.plot([1, 2, 3, 4], [1, 4, 2, 3]) # Plot some data on the
                         # ↪ axes.
```

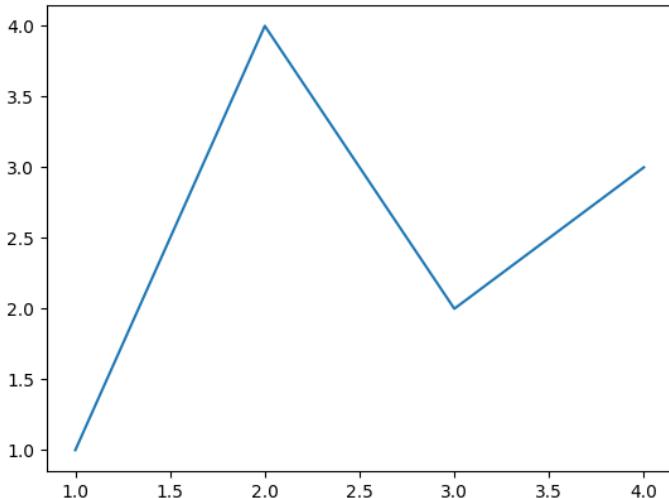


Figure 8.1:

Note that to get this Figure to display, you may have to call `plt.show()`, depending on your backend. For more details of Figures and backends, see Creating, viewing, and saving Matplotlib Figures.

## 8.2 Parts of a Figure

Here are the components of a Matplotlib Figure.

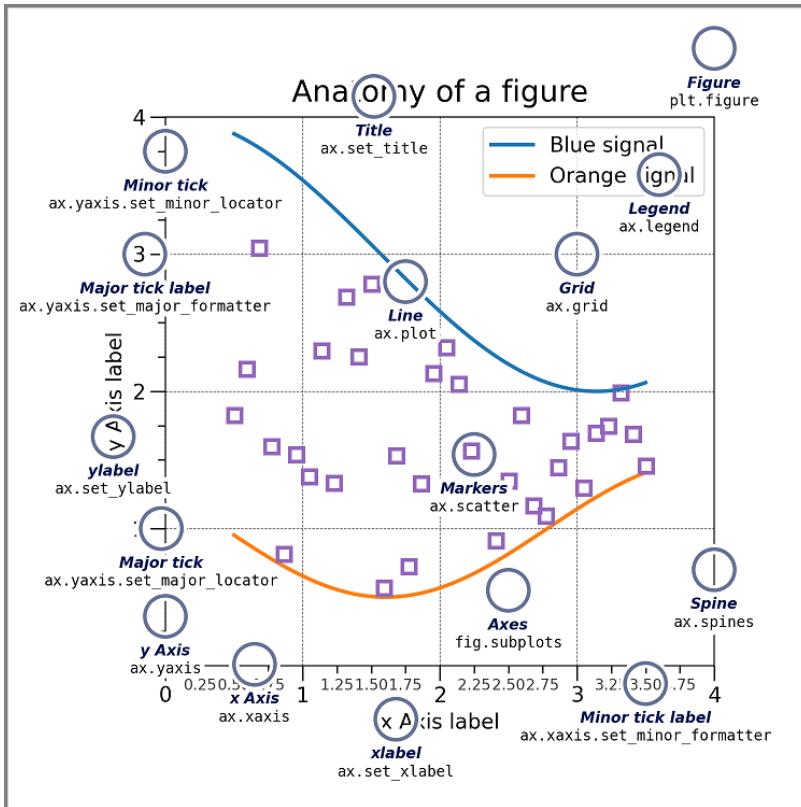


Figure 8.2:

### 8.2.1 Figure

The Figure keeps track of all the child Axes, a group of 'special' Artists (titles, figure legends, colorbars, etc), and even nested subfigures. The easiest way to create a new Figure is with pyplot:

```
fig = plt.figure() # an empty figure with no Axes
fig, ax = plt.subplots() # a figure with a single Axes
fig, axs = plt.subplots(2, 2) # a figure with a 2x2 grid of
                             # → Axes
# a figure with one axes on the left, and two on the right:
fig, axs = plt.subplot_mosaic([['left', 'right-top'],
                               ['left', 'right_bottom']])
```

It is often convenient to create the Axes together with the Figure, but you can also manually add Axes later on. Note that many Matplotlib backends support zooming and panning on figure windows. For more on Figures, see Creating, viewing, and saving Matplotlib Figures.

### 8.2.2 Axes

An Axes is an Artist attached to a Figure that contains a region for plotting data, and usually includes two (or three in the case of 3D) Axis objects (be aware of the difference between Axes and Axis) that provide ticks and tick labels to provide scales for the data in the Axes. Each Axes also has a title (set via `set_title()`), an x-label (set via `set_xlabel()`), and a y-label set via `set_ylabel()`. The Axes class and its member functions are the primary entry point to working with the OOP interface, and have most of the plotting methods defined on them (e.g. `ax.plot()`, shown above, uses the `plot` method)

### 8.2.3 Axis

These objects set the scale and limits and generate ticks (the marks on the Axis) and ticklabels (strings labeling the ticks). The location of the ticks is determined by a Locator object and the ticklabel strings are formatted by a Formatter. The combination of the correct Locator and Formatter gives very fine control over the tick locations and labels.

### 8.2.4 Artist

Basically, everything visible on the Figure is an Artist (even Figure, Axes, and Axis objects). This includes Text objects, Line2D objects, collections objects, Patch objects, etc. When the Figure is rendered, all of the Artists are drawn to the canvas. Most Artists are tied to

an Axes; such an Artist cannot be shared by multiple Axes, or moved from one to another.

## 8.3 Coding styles

As noted above, there are essentially two ways to use Matplotlib:

1. Explicitly create Figures and Axes, and call methods on them (the “object-oriented (OO) style”).
2. Rely on `pyplot` to implicitly create and manage the Figures and Axes, and use `pyplot` functions for plotting.

See Matplotlib Application Interfaces (APIs) for an explanation of the tradeoffs between the implicit and explicit interfaces. So one can use the OO-style

```
x = np.linspace(0, 2, 100) # Sample data.

# Note that even in the OO-style, we use `pyplot.figure` to
# → create the Figure.
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
ax.plot(x, x, label='linear') # Plot some data on the axes.
ax.plot(x, x**2, label='quadratic') # Plot more data on the
# → axes...
ax.plot(x, x**3, label='cubic') # ... and some more.
ax.set_xlabel('x label') # Add an x-label to the axes.
ax.set_ylabel('y label') # Add a y-label to the axes.
ax.set_title("Simple Plot") # Add a title to the axes.
ax.legend() # Add a legend.
```

or the `pyplot`-style:

```
x = np.linspace(0, 2, 100) # Sample data.
plt.figure(figsize=(5, 2.7), layout='constrained')
plt.plot(x, x, label='linear') # Plot some data on the (
# → implicit) axes.
plt.plot(x, x**2, label='quadratic') # etc.
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
```

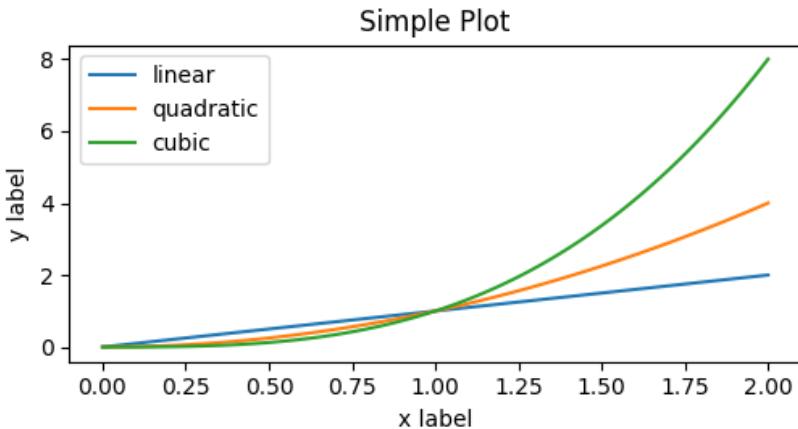


Figure 8.3:

Matplotlib’s documentation and examples use both the OO and the pyplot styles. In general, we suggest using the OO style, particularly for complicated plots, and functions and scripts that are intended to be reused as part of a larger project. However, the pyplot style can be very convenient for quick interactive work.

## 8.4 Colors

Matplotlib has a very flexible array of colors that are accepted for most Artists; see the colors tutorial for a list of specifications. Some Artists will take multiple colors. i.e. for a scatter plot, the edge of the markers can be different colors from the interior:

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.scatter(data1, data2, s=50, facecolor='CO', edgecolor='k')
```

## 8.5 Linewidths, linestyles, and markersizes

Line widths are typically in typographic points (1 pt = 1/72 inch) and available for Artists that have stroked lines. Similarly, stroked lines

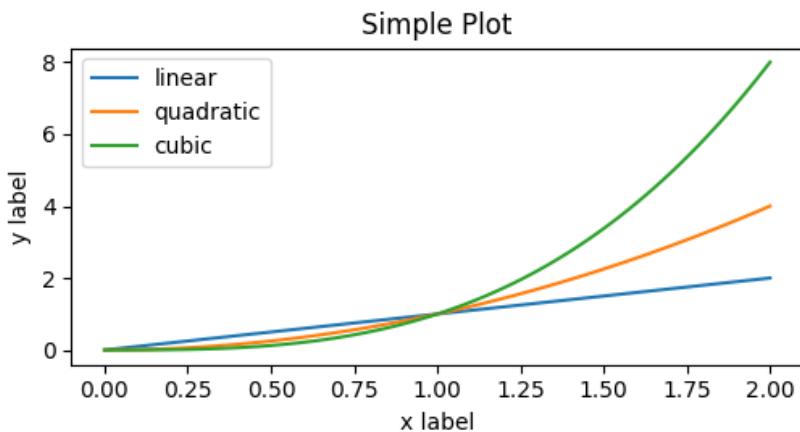


Figure 8.4:

can have a linestyle. See the `linestyles` example.

Marker size depends on the method being used. `plot` specifies `markerSize` in points, and is generally the “diameter” or width of the marker. `scatter` specifies `markerSize` as approximately proportional to the visual area of the marker. There is an array of `markerStyles` available as string codes (see `markers`), or users can define their own `MarkerStyle` (see `Marker` reference):

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(data1, 'o', label='data1')
ax.plot(data2, 'd', label='data2')
ax.plot(data3, 'v', label='data3')
ax.plot(data4, 's', label='data4')
ax.legend()
```

## 8.6 Labelling plots

### 8.6.1 Axes labels and text

`set_xlabel`, `set_ylabel`, and `set_title` are used to add text in the indicated locations (see `Text` in Matplotlib Plots for more discussion).

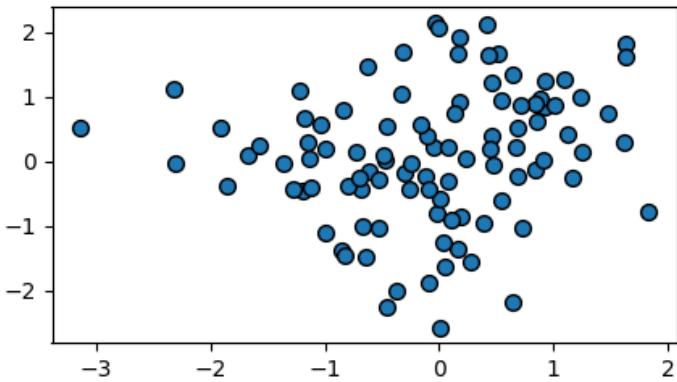


Figure 8.5:

Text can also be directly added to plots using text:

```

mu, sigma = 115, 15
x = mu + sigma * np.random.randn(10000)
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
# the histogram of the data
n, bins, patches = ax.hist(x, 50, density=True, facecolor='C0',
                           alpha=0.75)

ax.set_xlabel('Length [cm]')
ax.set_ylabel('Probability')
ax.set_title('Aardvark lengths\n(not really)')
ax.text(75, .025, r'$\mu=115,\ \sigma=15$')
ax.axis([55, 175, 0, 0.03])
ax.grid(True)

```

All of the text functions return a `matplotlib.text.Text` instance. Just as with lines above, you can customize the properties by passing keyword arguments into the text functions:

```
t = ax.set_xlabel('my data', fontsize=14, color='red')
```

These properties are covered in more detail in Text properties and layout.

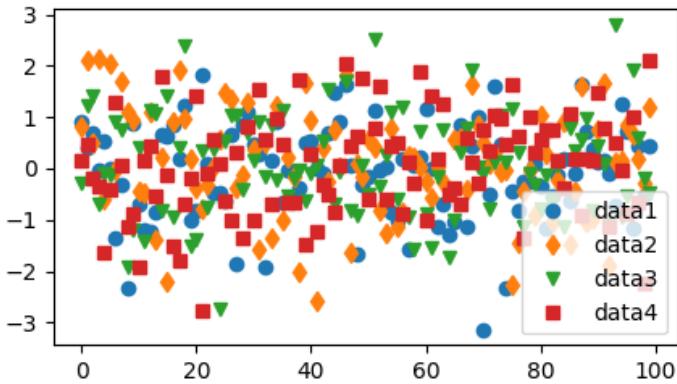


Figure 8.6:

### 8.6.2 Annotations

We can also annotate points on a plot, often by connecting an arrow pointing to *xy*, to a piece of text at *xytext*:

```
fig, ax = plt.subplots(figsize=(5, 2.7))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2 * np.pi * t)
line, = ax.plot(t, s, lw=2)

ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.set_xlim(-2, 2)
```

In this basic example, both *xy* and *xytext* are in data coordinates. There are a variety of other coordinate systems one can choose.

### 8.6.3 Legends

Often we want to identify lines or markers with a `Axes.legend`:

```
fig, ax = plt.subplots(figsize=(5, 2.7))
```

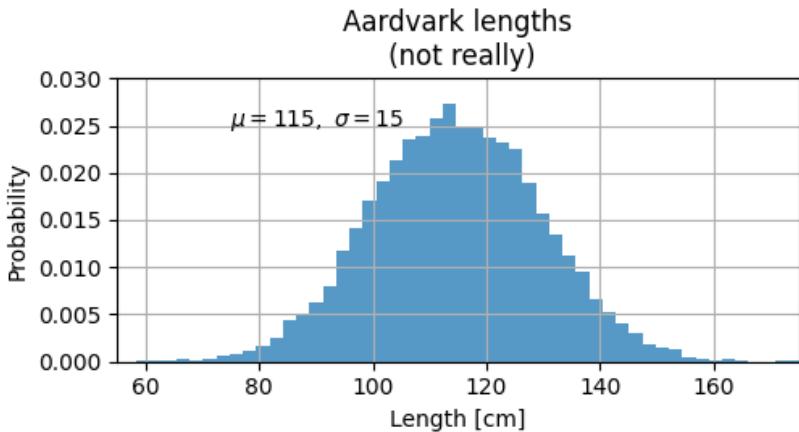


Figure 8.7:

```

ax.plot(np.arange(len(data1)), data1, label='data1')
ax.plot(np.arange(len(data2)), data2, label='data2')
ax.plot(np.arange(len(data3)), data3, 'd', label='data3')
ax.legend()

```

Legends in Matplotlib are quite flexible in layout, placement, and what Artists they can represent. They are discussed in detail in Legend guide.

## 8.7 Plotting data

Ploting for statistical data goes in tandem with the concept of level of data measurement or scale of measure, which is a classification that describes the nature of information within the values assigned to variables.<sup>72</sup> Psychologist Stanley Smith Stevens developed the best-known classification with four levels, or scales, of measurement: nominal, ordinal, interval, and ratio. This framework of distinguishing levels of measurement originated in psychology and has since had a complex history, being adopted and extended in some disciplines and by some scholars, and criticized or rejected by others.<sup>73</sup> Other classifications

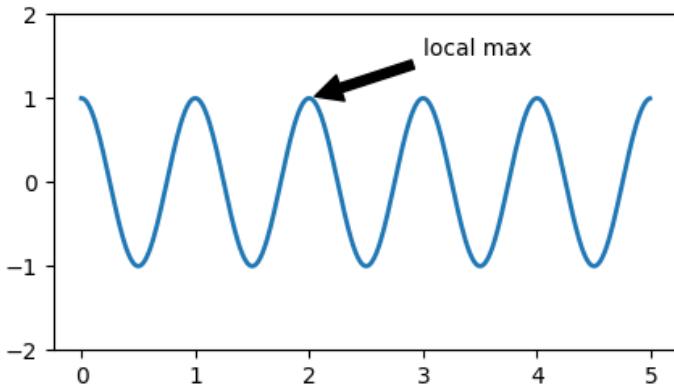


Figure 8.8:

include those by Mosteller and Tukey, and by Chrisman.<sup>74 75</sup>

In the theory of data analytics, there are four types of measurement levels. They are

1. Nominal
2. Ordinal
3. Interval
4. Ratio

### 8.7.1 Nominal

The nominal type differentiates between items or subjects based only on their names or (meta-)categories and other qualitative classifications they belong to; thus dichotomous data involves the construction of classifications as well as the classification of items. Discovery of an exception to a classification can be viewed as progress. Numbers may be used to represent the variables but the numbers do not have numerical value or relationship: for example, a globally unique identifier. Examples of these classifications include gender, nationality, ethnicity,

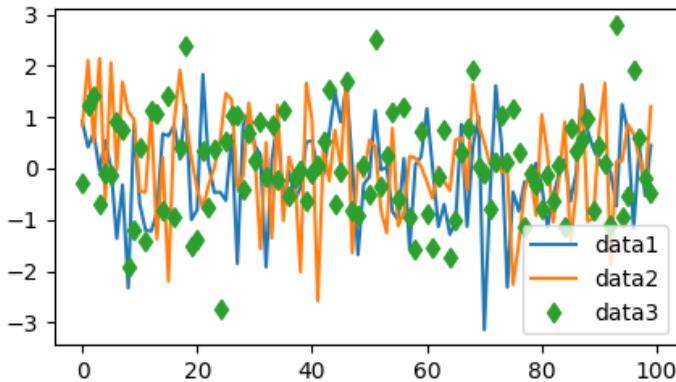


Figure 8.9:

language, genre, style, biological species, and form. In a university one could also use hall of affiliation as an example.

Plot a pie chart of animals and label the slices. To add labels, pass a list of labels to the labels parameter

```
import matplotlib.pyplot as plt
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]

fig, ax = plt.subplots()
ax.pie(sizes, labels=labels)
```

Each slice of the pie chart is a `patches.Wedge` object; therefore in addition to the customizations shown here, each wedge can be customized using the `wedgeprops` argument, as demonstrated in Nested pie charts. Pass a function or format string to `autopct` to label slices.

```
fig, ax = plt.subplots()
ax.pie(sizes, labels=labels, autopct='%1.1f%%')
```

By default, the label values are obtained from the percent size of the slice.<sup>76</sup>

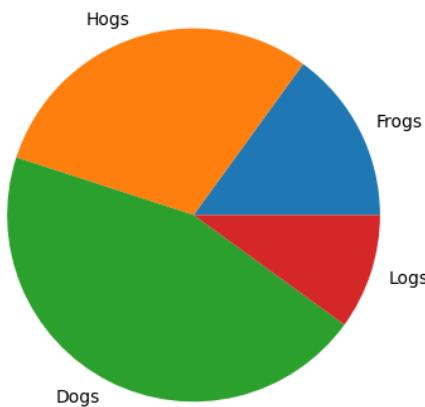


Figure 8.10:

### 8.7.2 Ordinal scale

The ordinal type allows for rank order (1st, 2nd, 3rd, etc.) by which data can be sorted but still does not allow for a relative degree of difference between them. Examples include, on one hand, dichotomous data with dichotomous (or dichotomized) values such as 'sick' vs. 'healthy' when measuring health, 'guilty' vs. 'not-guilty' when making judgments in courts, 'wrong/false' vs. 'right/true' when measuring truth value, and, on the other hand, non-dichotomous data consisting of a spectrum of values, such as 'completely agree', 'mostly agree', 'mostly disagree', 'completely disagree' when measuring opinion.

The ordinal scale places events in order, but there is no attempt to make the intervals of the scale equal in terms of some rule. Rank orders represent ordinal scales and are frequently used in research relating to qualitative phenomena. A student's rank in his graduation class

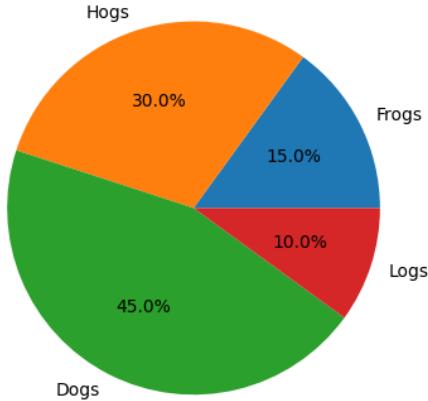


Figure 8.11:

involves the use of an ordinal scale. One has to be very careful in making a statement about scores based on ordinal scales.

With Pyplot, you can use the `bar()` function to draw bar graphs:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x,y)
plt.show()
```

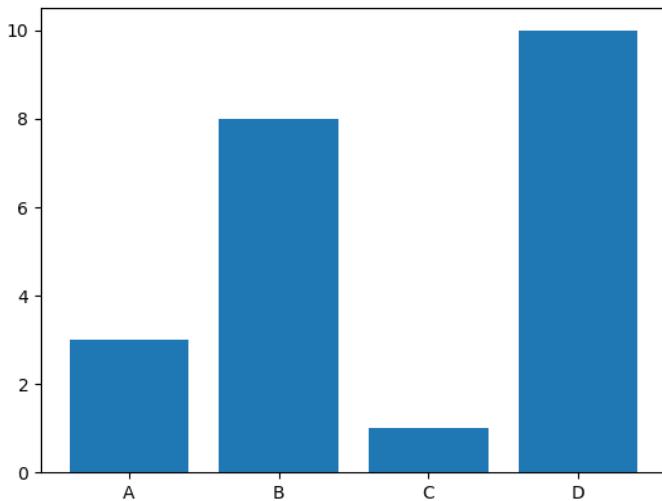


Figure 8.12:

### 8.7.3 Interval scale

The interval type allows for the degree of difference between items, but not the ratio between them. Examples include temperature scales with the Celsius scale, which has two defined points (the freezing and boiling point of water at specific conditions) and then separated into 100 intervals, date when measured from an arbitrary epoch (such as AD), location in Cartesian coordinates, and direction measured in degrees from true or magnetic north. Ratios are not meaningful since 20 degrees Celsius cannot be said to be “twice as hot” as 10 degrees Celsius (unlike temperature in Kelvins), nor can multiplication/division be carried out between any two dates directly. However, ratios of differences can be expressed; for example, one difference can be twice another. Interval type variables are sometimes also called “scaled variables”, but the formal mathematical term is an affine space (in this case an affine line).

If you want the bars to be displayed for interval data use the either `bar()` or `barh()` function.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.barh(x, y)
plt.show()
```

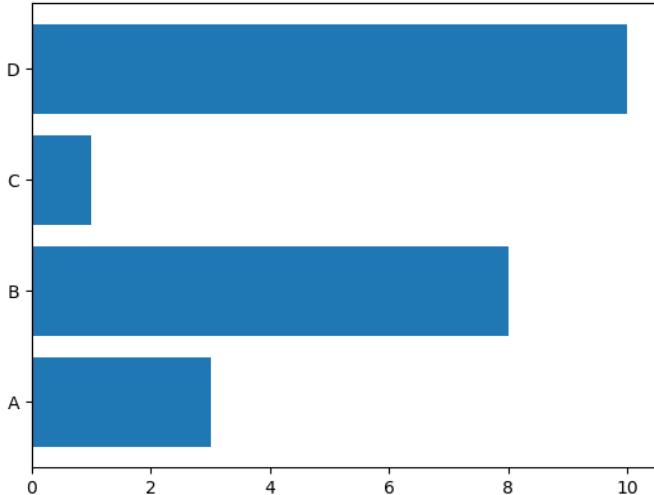


Figure 8.13:

#### 8.7.4 Ratio scale

The ratio type takes its name from the fact that measurement is the estimation of the ratio between a magnitude of a continuous quantity and a unit of measurement of the same kind. Most measurement in

the physical sciences and engineering is done on ratio scales. Examples include mass, length, duration, plane angle, energy and electric charge. In contrast to interval scales, ratios can be compared using division. Very informally, many ratio scales can be described as specifying “how much” of something (i.e. an amount or magnitude). Ratio scale is often used to express an order of magnitude such as for temperature in Orders of magnitude (temperature).

To generate a 1D histogram we only need a single vector of numbers. For a 2D histogram we'll need a second vector. We'll generate both below, and show the histogram for each vector.

```
N_points = 100000
n_bins = 20

# Generate two normal distributions
dist1 = rng.standard_normal(N_points)
dist2 = 0.4 * rng.standard_normal(N_points) + 5

fig, axs = plt.subplots(1, 2, sharey=True, tight_layout=True)

# We can set the number of bins with the *bins* keyword
# → argument.
axs[0].hist(dist1, bins=n_bins)
axs[1].hist(dist2, bins=n_bins)
```

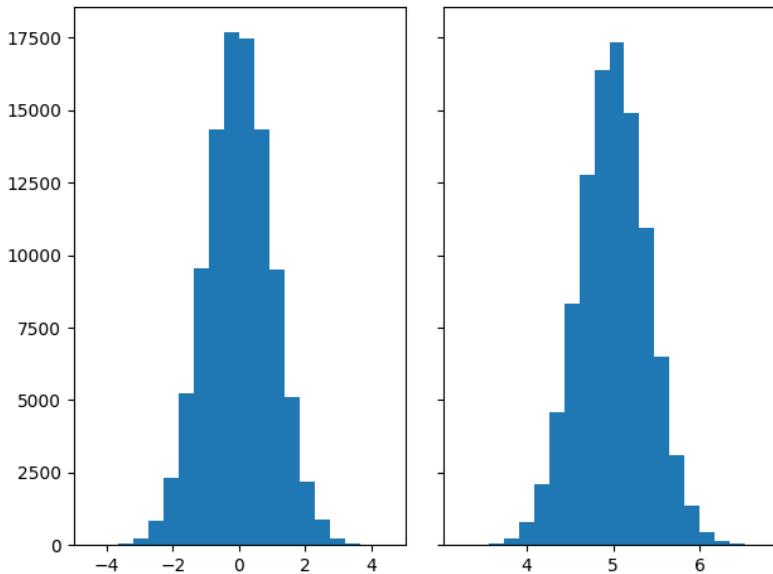


Figure 8.14:

## Notes

<sup>72</sup>Kirch, Wilhelm, ed. (2008). "Level of Measurement". Encyclopedia of Public Health. Vol. 2. Springer. pp. 851–852. doi:10.1007/978-1-4020-5614-7\_1971. ISBN 978 – 1 – 4020 – 5613 – 0.

<sup>73</sup>Michell, J. (1986). "Measurement scales and statistics: a clash of paradigms". *Psychological Bulletin*. 100 (3): 398–407. doi:10.1037/0033-2909.100.3.398.

<sup>74</sup>Mosteller, Frederick (1977). Data

analysis and regression : a second course in statistics. Reading, Mass: Addison-Wesley Pub. Co. ISBN 978-0201048544.

<sup>75</sup>Chrisman, Nicholas R. (1998). "Rethinking Levels of Measurement for Cartography". *Cartography and Geographic Information Science*. 25 (4): 231–242. doi:10.1559/152304098782383043. ISSN 1523-0406

<sup>76</sup>Visit [https://matplotlib.org/stable/gallery/pie\\_and\\_polar\\_charts/pie\\_features.html](https://matplotlib.org/stable/gallery/pie_and_polar_charts/pie_features.html) for more details.





# Index

Editors, 27  
Execution Modes, 28  
  
Febunaci, 22  
  
Jupyter, 43  
  
literate programming, 43  
  
markdown, 44  
  
reproducible research, 43