Data Wrangling with Python

Creating actionable data from raw sources

www.packt.com

Dr. Tirthajyoti Sarkar and
Shubhadeep Roychowdhury

Data Wrangling with Python

Creating actionable data from raw sources

Dr. Tirthajyoti Sarkar and Shubhadeep Roychowdhury

Packt

Data Wrangling with Python

Authors: Dr. Tirthajyoti Sarkar and Shubhadeep Roychowdhury

Managing Editor: Steffi Monteiro

Acquisitions Editor: Kunal Sawant

Production Editor: Nitesh Thakur

First Published: February 2019

Production Reference: 1280219

ISBN: 978-1-78980-011-1

Published by Packt Publishing Ltd.

Livery Place, 35 Livery Street

Birmingham B3 2PB, UK

Table of Contents

Preface

About

This section briefly introduces the author(s), the coverage of this book, the technical skills you'll need to get started, and the hardware and software requirements required to complete all of the included activities and exercises.

About the Book

For data to be useful and meaningful, it must be curated and refined. *Data Wrangling with Python* teaches you all the core ideas behind these processes and equips you with knowledge about the most popular tools and techniques in the domain.

The book starts with the absolute basics of Python, focusing mainly on data structures, and then quickly jumps into the NumPy and pandas libraries as the fundamental tools for data wrangling. We emphasize why you should stay away from the traditional way of data cleaning, as done in other languages, and take advantage of the specialized pre-built routines in Python. Thereafter, you will learn how, using the same Python backend, you can extract and transform data from a diverse array of sources, such as the internet, large database vaults, or Excel financial tables. Then, you will also learn how to handle missing or incorrect data, and reformat it based on the requirements from the downstream analytics tool. You will learn about these concepts through real-world examples and datasets.

By the end of this book, you will be confident enough to handle a myriad of sources to extract, clean, transform, and format your data efficiently.

About the Authors

Dr. Tirthajyoti Sarkar works as a senior principal engineer in the semiconductor technology domain, where he applies cutting-edge data science/machine learning techniques to design automation and predictive analytics. He writes regularly about Python programming and data science topics. He holds a Ph.D. from the University of Illinois, and certifications in artificial intelligence and machine learning from Stanford and MIT.

Shubhadeep Roychowdhury works as a senior software engineer at a Paris-based cybersecurity start-up, where he is applying state-of-the-art computer vision and data engineering algorithms and tools to develop cutting-edge products. He often writes about algorithm implementation in Python and similar topics. He holds a master's degree in computer science from West Bengal University of Technology and certifications in machine learning from Stanford.

Learning Objectives

- Use and manipulate complex and simple data structures
- Harness the full potential of DataFrames and numpy.array at run time
- Perform web scraping with BeautifulSoup4 and html5lib
- Execute advanced string search and manipulation with RegEX
- Handle outliers and perform data imputation with Pandas
- Use descriptive statistics and plotting techniques
- Practice data wrangling and modeling using data generation techniques

Approach

Data Wrangling with Python takes a practical approach to equip beginners with the most essential data analysis tools in the shortest possible time. It contains multiple activities that use real-life business scenarios for you to practice and apply your new skills in a highly relevant context.

Audience

Data Wrangling with Python is designed for developers, data analysts, and business analysts who are keen to pursue a career as a full-fledged data scientist or analytics expert. Although, this book is for beginners, prior working knowledge of Python is necessary to easily grasp the concepts covered here. It will also help to have rudimentary knowledge of relational database and SQL.

Minimum Hardware Requirements

For the optimal student experience, we recommend the following hardware configuration:

- Processor: Intel Core i5 or equivalent
- Memory: 8 GB RAM
- Storage: 35 GB available space

Software Requirements

You'll also need the following software installed in advance:

- OS: Windows 7 SP1 64-bit, Windows 8.1 64-bit or Windows 10 64-bit, Ubuntu Linux, or the latest version of macOS
- version of OS X
- Processor: Intel Core i5 or equivalent
- Memory: 4 GB RAM (8 GB Preferred)
- Storage: 35 GB available space

Conventions

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: " This will return the value associated with it- **`["list_element1", 34]`**"

A block of code is set as follows:

```
list_1 = []

    for x in range(0, 10):

    list_1.append(x)

list_1
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on **New** and choose **Python 3**."

Installation and Setup

Each great journey begins with a humble step. Our upcoming adventure in the land of data wrangling is no exception. Before we can do awesome things with data, we need to be prepared with the most productive environment. In this short section, we shall see how to do that.

The only prerequisite regarding the environment for this book is to have Docker installed. If you have never heard of Docker or you have only a very faint idea what it is, then fear not. All you need to know about Docker for the purpose of this book is this: Docker is a lightweight containerization engine that runs on all three major platforms (Linux, Windows, and macOS). The main idea behind Docker is give you safe, easy, and lightweight virtualization on top of your native OS.

Install Docker

1. To install Docker on a Mac or Windows machine, create an account on Docker and download the latest version. It's easy to install and set up.
2. Once you have set up Docker, open a shell (or Terminal if you are a Mac user) and type the following command to verify that the installation has been successful:

   ```
   docker version
   ```

   If the output shows you the server and client version of Docker, then you are all set up.

Pull the image

1. Pull the image and you will have all the necessary packages (including Python 3.6.6) installed and ready for you to start working. Type the following command in a shell:

   ```
   docker pull rcshubhadeep/packt-data-wrangling-base
   ```

2. If you want to know the full list of all the packages and their versions included in this image, you can check out the **`requirements.txt`** file in the **`setup`** folder of the source code repository of this book. Once the image is there, you are ready to roll. Downloading it may take time, depending on your connection speed.

Run the environment

1. Run the image using the following command:

   ```
   docker run -p 8888:8888 -v 'pwd':/notebooks -it rcshubhadeep/packt-data-
   wrangling-base
   ```

   This will give you a ready-to-use environment.

2. Open a browser tab in Chrome or Firefox and go to **`http://localhost:8888`**. You will be prompted to enter a token. The token is **`dw_4_all`**.
3. Before you run the image, create a new folder and navigate there from the shell using the **`cd`** command.

   Once you create a notebook and save it as **`ipynb`** file. You can use *Ctrl* +C to stop running the image.

Introduction to Jupyter notebook

Project Jupyter is open source, free software that gives you the ability to run code, written in Python and some other languages, interactively from a special notebook, similar to a browser interface. It was born in 2014 from the **`IPython`** project and has since become the default choice for the entire data science workforce.

1. Once you are running the Jupyter server, click on **New** and choose **Python 3**. A new browser tab will open with a new and empty notebook. Rename the Jupyter file:

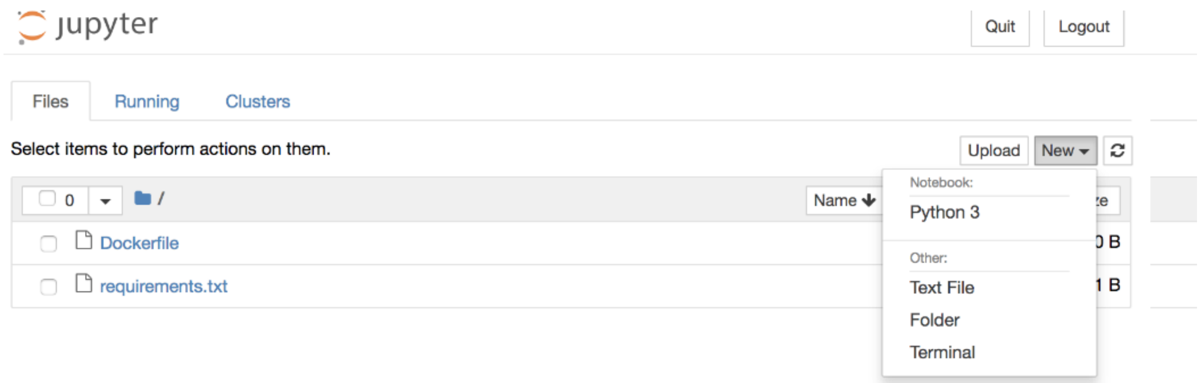
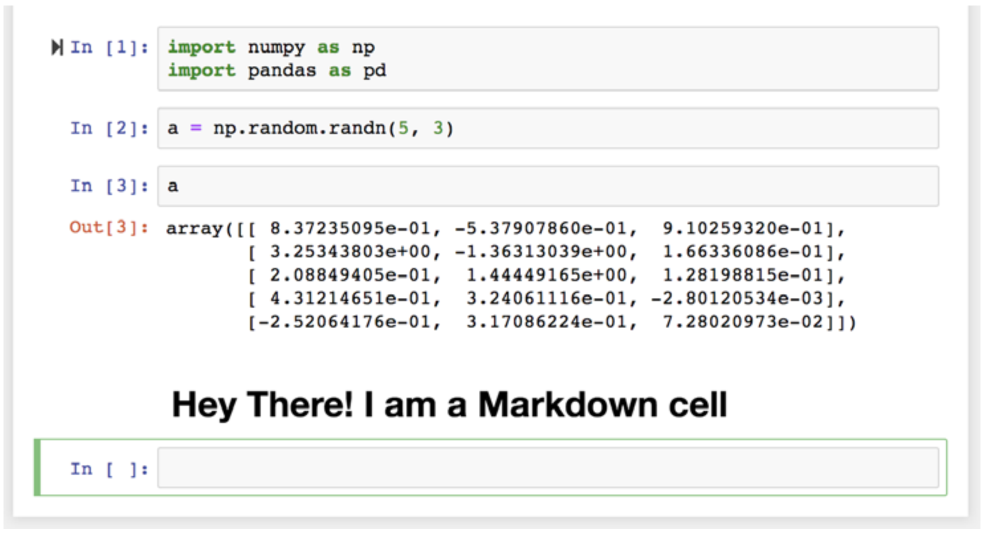


Figure 0.1: Jupyter server interface

The main building blocks of Jupyter notebooks are cells. There are two types of cells: **In** (short for input) and **Out** (short for output). You can write code, normal text, and Markdown in **In** cells, press *Shift* + *Enter* (or *Shift* + *Return*), and the code written in that particular **In** cell will be executed. The result will be shown in an **Out** cell, and you will land in a new **In** cell, ready for the next block of code. Once you get used to this interface, you will slowly discover the power and flexibility it offers.

2. One final thing you should know about Jupyter cells is that when you start a new cell, by default, it is assumed that you will write code in it. However, if you want to write text, then you have to change the type. You can do that using the following sequence of keys: *Escape->m->Enter*:



```
In [1]: import numpy as np
        import pandas as pd

In [2]: a = np.random.randn(5, 3)

In [3]: a

Out[3]: array([[ 8.37235095e-01, -5.37907860e-01,  9.10259320e-01],
               [ 3.25343803e+00, -1.36313039e+00,  1.66336086e-01],
               [ 2.08849405e-01,  1.44449165e+00,  1.28198815e-01],
               [ 4.31214651e-01,  3.24061116e-01, -2.80120534e-03],
               [-2.52064176e-01,  3.17086224e-01,  7.28020973e-02]])
```

Hey There! I am a Markdown cell

```
In [ ]:
```

Figure 0.2: Jupyter notebook

3. And when you are done with writing the text, execute it using *Shift* + *Enter*. Unlike the code cells, the result of the compiled Markdown will be shown in the same place as the "**In**" cell.

Note

To have a "Cheat sheet" of all the handy key shortcuts in Jupyter, you can bookmark this Gist: https://gist.github.com/kidpixo/f4318f8c8143adee5b40. With this basic introduction and the image ready to be used, we are ready to embark on the exciting and enlightening journey that awaits us!

Installing the Code Bundle

Copy the code bundle for the class to the **`C:/Code`** folder.

Additional Resources

The code bundle for this book is also hosted on GitHub at: https://github.com/TrainingByPackt/Data-Wrangling-with-Python.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

1

Introduction to Data Wrangling with Python

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Define the importance of data wrangling in data science
- Manipulate the data structures that are available in Python
- Compare the different implementations of the inbuilt Python data structures

This chapter describes the importance of data wrangling, identifies the important tasks to be performed in data wrangling, and introduces basic Python data structures.

Introduction

Data science and analytics are taking over the whole world and the job of a data scientist is routinely being called the coolest job of the 21st century. But for all the emphasis on data, it is the science that makes you – the practitioner – truly valuable.

To practice high-quality science with data, you need to make sure it is properly sourced, cleaned, formatted, and pre-processed. This book teaches you the most essential basics of this invaluable component of the data science pipeline: data wrangling. In short, data wrangling is the process that ensures that the data is in a format that is clean, accurate, formatted, and ready to be used for data analysis.

A prominent example of data wrangling with a large amount of data is the one conducted at the Supercomputer Center of University of California San Diego (UCSD). The problem in California is that wildfires are very common, mainly because of the dry weather and extreme heat, especially during the summers. Data scientists at the UCSD Supercomputer Center gather data to predict the nature and spread direction of the fire. The data that comes from diverse sources such as weather stations, sensors in the forest, fire stations, satellite imagery, and Twitter feeds might still be incomplete or missing. This data needs to be cleaned and formatted so that it can be used to predict future occurrences of wildfires.

This is an example of how data wrangling and data science can prove to be helpful and relevant.

Importance of Data Wrangling

Oil does not come in its final form from the rig; it has to be refined. Similarly, data must be curated, massaged, and refined to be used in intelligent algorithms and consumer products. This is known as wrangling. Most data scientists spend the majority of their time data wrangling.

Data wrangling is generally done at the very first stage of a data science/analytics pipeline. After the data scientists identify useful data sources for solving the business problem (for instance, in-house database storage or internet or streaming sensor data), they then proceed to extract, clean, and format the necessary data from those sources.

Generally, the task of data wrangling involves the following steps:

- Scraping raw data from multiple sources (including web and database tables)
- Imputing, formatting, and transforming – basically making it ready to be used in the modeling process (such as advanced machine learning)
- Handling read/write errors
- Detecting outliers
- Performing quick visualizations (plotting) and basic statistical analysis to judge the quality of your formatted data

This is an illustrative representation of the positioning and essential functional role of data wrangling in a typical data science pipeline:

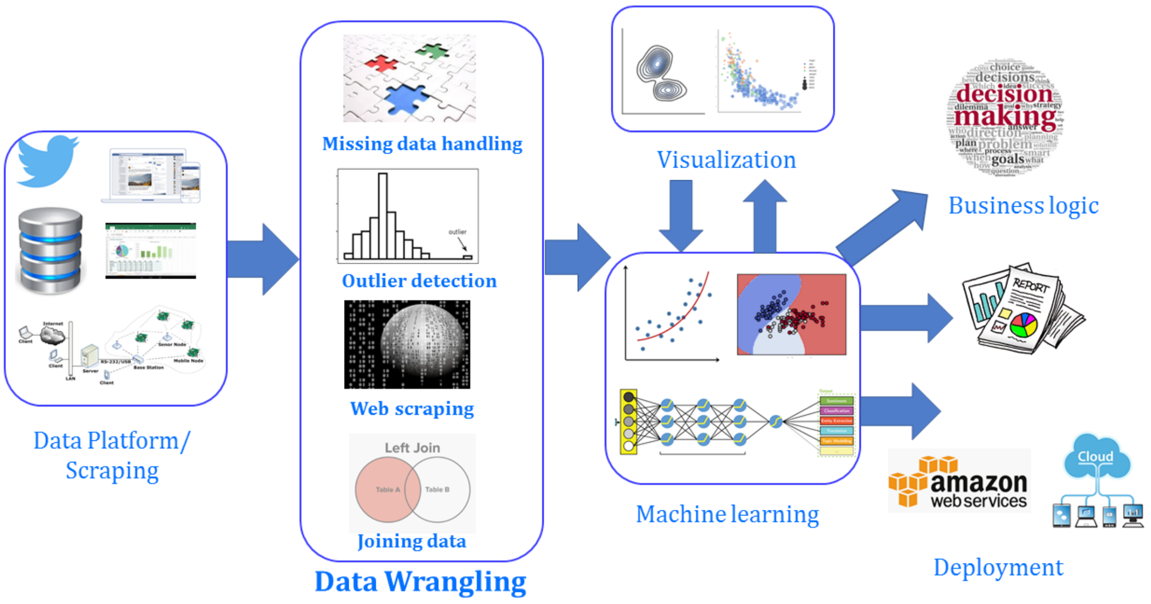


Figure 1.1: Process of data wrangling

The process of data wrangling includes first finding the appropriate data that's necessary for the analysis. This data can be from one or multiple sources, such as tweets, bank transaction statements in a relational database, sensor data, and so on. This data needs to be cleaned. If there is missing data, we will either delete or substitute it, with the help of several techniques. If there are outliers, we need to first detect them and then handle them appropriately. If data is from multiple sources, we will have to perform join operations to combine it.

In an extremely rare situation, data wrangling may not be needed. For example, if the data that's necessary for a machine learning task is already stored in an acceptable format in an in-house database, then a simple SQL query may be enough to extract the data into a table, ready to be passed on to the modeling stage.

Python for Data Wrangling

There is always a debate on whether to perform the wrangling process using an enterprise tool or by using a programming language and associated frameworks. There are many commercial, enterprise-level tools for data formatting and pre-processing that do not involve much coding on the part of the user. These examples include the following:

- General purpose data analysis platforms such as Microsoft Excel (with add-ins)
- Statistical discovery package such as **JMP** (from SAS)
- Modeling platforms such as **RapidMiner**
- Analytics platforms from niche players focusing on data wrangling, such as **Trifacta**, **Paxata**, and **Alteryx**

However, programming languages such as Python provide more flexibility, control, and power compared to these off-the-shelf tools.

As the volume, velocity, and variety (the three Vs of **big data**) of data undergo rapid changes, it is always a good idea to develop and nurture a significant amount of in-house expertise in data wrangling using fundamental programming frameworks so that an organization is not beholden to the whims and fancies of any enterprise platform for as basic a task as data wrangling:

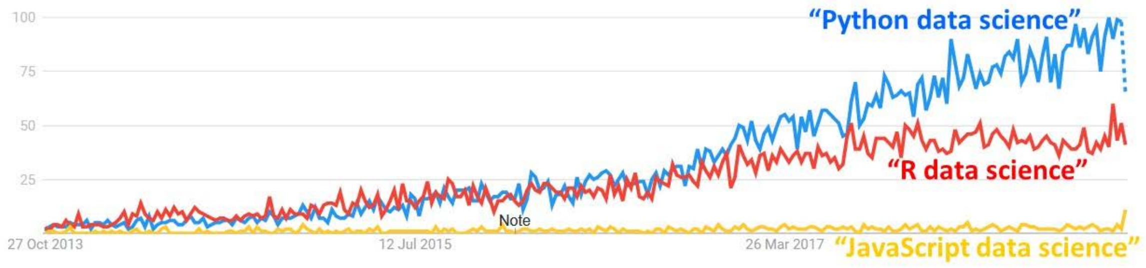


Figure 1.2: Google trend worldwide over the last Five years

A few of the obvious advantages of using an open source, free programming paradigm such as Python for data wrangling are the following:

- General purpose open source paradigm putting no restriction on any of the methods you can develop for the specific problem at hand
- Great ecosystem of fast, optimized, open source libraries, focused on data analytics
- Growing support to connect Python to every conceivable data source type
- Easy interface to basic statistical testing and quick visualization libraries to check data quality
- Seamless interface of the data wrangling output with advanced machine learning models

Python is the most popular language of choice of machine learning and artificial intelligence these days.

Lists, Sets, Strings, Tuples, and Dictionaries

Now that we have learned the importance of Python, we will start by exploring various basic data structures in Python. We will learn techniques to handle data. This is invaluable for a data practitioner.

We can issue the following command to start a new Jupyter server by typing the following in to the Command Prompt window:

```
docker run -p 8888:8888 -v 'pwd':/notebooks -it rcshubhadeep/packt-data-
wrangling-base:latest ipython
```

This will start a jupyter server and you can visit it at **`http://localhost:8888`** and use the passcode **`dw_4_all`** to access the main interface.

Lists

Lists are fundamental Python data structures that have continuous memory locations, can host different data types, and can be accessed by the index.

We will start with a list and list comprehension. We will generate a list of numbers, and then examine which ones among them are even. We will sort, reverse, and check for duplicates. We will also see how many different ways we can access the list elements, iterating over them and checking the membership of an element.

The following is an example of a simple list:

```
list_example = [51, 27, 34, 46, 90, 45, -19]
```

The following is also an example of a list:

```
list_example2 = [15, "Yellow car", True, 9.456, [12, "Hello"]]
```

As you can see, a list can contain any number of the allowed datatype, such as **`int`**, **`float`**, **`string`**, and **`Boolean`**, and a list can also be a mix of different data types (including nested lists).

If you are coming from a strongly typed language, such as C, C++, or Java, then this will probably be strange as you are not allowed to mix different kinds of data types in a single array in those languages. Lists are somewhat like arrays, in the sense that they are both based on continuous memory locations and can be accessed using indexes. But the power of Python lists come from the fact that they can host different data types and you are allowed to manipulate the data.

> **Note**
>
> Be careful, though, as the very power of lists, and the fact that you can mix different data types in a single list, can actually create subtle bugs that can be very difficult to track.

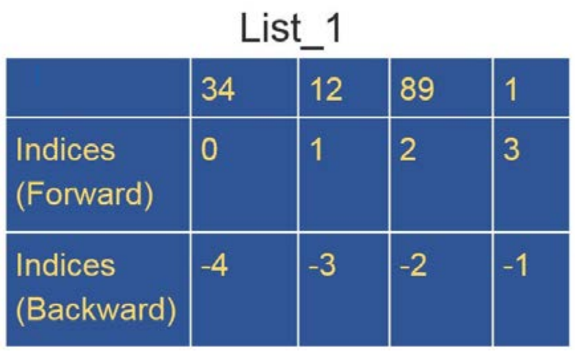
Exercise 1: Accessing the List Members

In the following exercise, we will be creating a list and then observing the different ways of accessing the elements:

1. Define a list called **`list_1`** with four integer members, using the following command:

    ```
    list_1 = [34, 12, 89, 1]
    ```

    The indices will be automatically assigned, as follows:



List_1

	34	12	89	1
Indices (Forward)	0	1	2	3
Indices (Backward)	-4	-3	-2	-1

Figure 1.3: List showing the forward and backward indices

2. Access the first element from **`list_1`** using its forward index:

   ```
   list_1[0] #34
   ```

3. Access the last element from **`list_1`** using its forward index:

   ```
   list_1[3] #1
   ```

4. Access the last element from **`list_1`** using the **`len`** function:

   ```
   list_1[len(list_1) - 1] #1
   ```

   The **`len`** function in Python returns the length of the specified list.

5. Access the last element from **`list_1`** using its backward index:

   ```
   list_1[-1] #1
   ```

6. Access the first three elements from **`list_1`** using forward indices:

   ```
   list_1[1:3] # [12, 89]
   ```

   This is also called list slicing, as it returns a smaller list from the original list by extracting only, a part of it. To slice a list, we need two integers. The first integer will denote the start of the slice and the second integer will denote the end-1 element.

> **Note**
>
> Notice that slicing did not include the third index or the end element. This is how list slicing works.

7. Access the last two elements from **`list_1`** by slicing:

   ```
   list_1[-2:] # [89, 1]
   ```

8. Access the first two elements using backward indices:

   ```
   list_1[:-2] # [34, 12]
   ```

   When we leave one side of the colon (**`:`**) blank, we are basically telling Python either to go until the end or start from the beginning of the list. It will automatically apply the rule of list slices that we just learned.

9. Reverse the elements in the string:

   ```
   list_1[-1::-1] # [1, 89, 12, 34]
   ```

> **Note**
>
> The last bit of code is not very readable, meaning it is not obvious just by looking at it what it is doing. It is against Python's philosophy. So, although this kind of code may look clever, we should resist the temptation to write code like this.

Exercise 2: Generating a List

We are going to examine various ways of generating a list:

1. Create a list using the **append** method:

   ```
   list_1 = []
   for x in range(0, 10):
       list_1.append(x)
   list_1
   ```

   The output will be as follows:

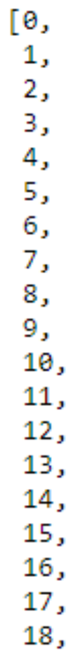
   ```
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
   ```

   Here, we started by declaring an empty list and then we used a **for** loop to append values to it. The **append** method is a method that's given to us by the Python list data type.

2. Generate a list using the following command:

   ```
   list_2 = [x for x in range(0, 100)]
   list_2
   ```

The partial output is as follows:



```
[0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10,
 11,
 12,
 13,
 14,
 15,
 16,
 17,
 18,
```

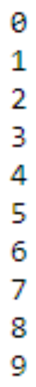
Figure 1.4: List comprehension

This is list comprehension, which is a very powerful tool that we need to master. The power of list comprehension comes from the fact that we can use conditionals inside the comprehension itself.

3. Use a **`while`** loop to iterate over a list to understand the difference between a **`while`** loop and a **`for`** loop:

```
i = 0
while i < len(list_1) :
    print(list_1[i])
    i += 1
```

The partial output will be as follows:



```
0
1
2
3
4
5
6
7
8
9
```

Figure 1.5: Output showing the contents of list_1 using a while loop

4. Create **`list_3`** with numbers that are divisible by **`5`**:

   ```
   list_3 = [x for x in range(0, 100) if x % 5 == 0]
   list_3
   ```

   The output will be a list of numbers up to 100 in increments of 5:

   ```
   [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 
   95]
   ```

5. Generate a list by adding the two lists:

   ```
   list_1 = [1, 4, 56, -1]
   list_2 = [1, 39, 245, -23, 0, 45]
   list_3 = list_1 + list_2
   list_3
   ```

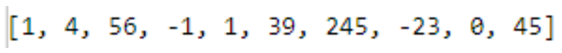
   The output is as follows:

   ```
   [1, 4, 56, -1, 1, 39, 245, -23, 0, 45]
   ```

6. Extend a string using the extend keyword:

   ```
   list_1.extend(list_2)
   list_1
   ```

The partial output is as follows:



```
[1, 4, 56, -1, 1, 39, 245, -23, 0, 45]
```

Figure 1.6: Contents of list_1

The second operation changes the original list (list_1) and appends all the elements of list_2 to it. So, be careful when using it.

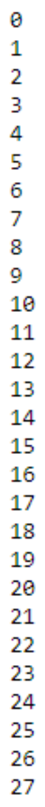
Exercise 3: Iterating over a List and Checking Membership

We are going to iterate over a list and test whether a certain value exists in it:

1. Iterate over a list:

   ```
   list_1 = [x for x in range(0, 100)]
   for i in range(0, len(list_1)):
       print(list_1[i])
   ```

   The output is as follows:



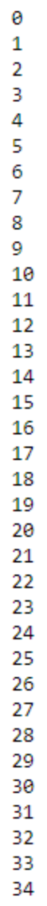
```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
```

Figure 1.7: Section of list_1

2. However, it is not very Pythonic. Being Pythonic is to follow and conform to a set of best practices and conventions that have been created over the years by thousands of very able developers, which in this case means to use the **`in`** keyword, because Python does not have index initialization, bounds checking, or index incrementing, unlike traditional languages. The Pythonic way of iterating over a list is as follows:

   ```
   for i in list_1:
       print(i)
   ```

   The output is as follows:

   ```
   0
   1
   2
   3
   4
   5
   6
   7
   8
   9
   10
   11
   12
   13
   14
   15
   16
   17
   18
   19
   20
   21
   22
   23
   24
   25
   26
   27
   28
   29
   30
   31
   32
   33
   34
   ```

   Figure 1.8: A section of list_1

   Notice that, in the second method, we do not need a counter anymore to access the list index; instead, Python's **`in`** operator gives us the element at the *i* th position directly.

3. Check whether the integers 25 and -45 are in the list using the **`in`** operator:

   ```
   25 in list_1
   ```

   The output is **True.**

   ```
   -45 in list_1
   ```

   The output is **False.**

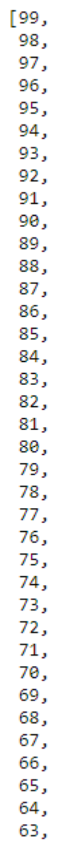
Exercise 4: Sorting a List

We generated a list called **`list_1`** in the previous exercise. We are going to sort it now:

1. As the list was originally a list of numbers from **`0`** to **`99`**, we will sort it in the reverse direction. To do that, we will use the **`sort`** method with **`reverse=True`**:

   ```
   list_1.sort(reverse=True)
   list_1
   ```

   The partial output is as follows:



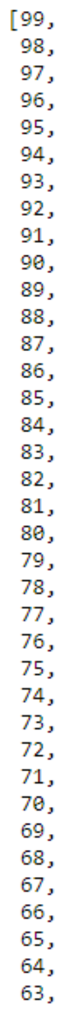
```
[99,
 98,
 97,
 96,
 95,
 94,
 93,
 92,
 91,
 90,
 89,
 88,
 87,
 86,
 85,
 84,
 83,
 82,
 81,
 80,
 79,
 78,
 77,
 76,
 75,
 74,
 73,
 72,
 71,
 70,
 69,
 68,
 67,
 66,
 65,
 64,
 63,
```

Figure 1.9: Section of output showing the reversed list

2. We can use the **reverse** method directly to achieve this result:

   ```
   list_1.reverse()
   list_1
   ```

   The output is as follows:



```
[99,
 98,
 97,
 96,
 95,
 94,
 93,
 92,
 91,
 90,
 89,
 88,
 87,
 86,
 85,
 84,
 83,
 82,
 81,
 80,
 79,
 78,
 77,
 76,
 75,
 74,
 73,
 72,
 71,
 70,
 69,
 68,
 67,
 66,
 65,
 64,
 63,
```

Figure 1.10: Section of output after reversing the string

> **Note**
>
> The difference between the sort function and the reverse function is the fact that we can use sort with custom sorting functions to do custom sorting, whereas we can only use reverse to reverse a list. Here also, both the functions work in-place, so be aware of this while using them.

Exercise 5: Generating a Random List

In this exercise, we will be generating a **list** with random numbers:

1. Import the **random** library:

   ```
   import random
   ```

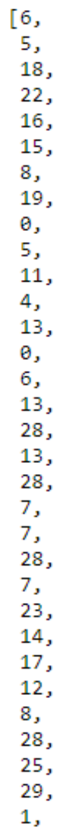
2. Use the **randint** function to generate random integers and add them to a list:

   ```
   list_1 = [random.randint(0, 30) for x in range (0, 100)]
   ```

3. Print the list using **print(list_1)**. Note that there will be duplicate values in **list_1**:

   ```
   list_1
   ```

   The sample output is as follows:



```
[6,
 5,
 18,
 22,
 16,
 15,
 8,
 19,
 0,
 5,
 11,
 4,
 13,
 0,
 6,
 13,
 28,
 13,
 28,
 7,
 7,
 28,
 7,
 23,
 14,
 17,
 12,
 8,
 28,
 25,
 29,
 1,
```

Figure 1.11: Section of the sample output for list_1

There are many ways to get a list of unique numbers, and while you may be able to write a few lines of code using a for loop and another list (you should actually try doing it!), let's see how we can do this without a for loop and with a single line of code. This will bring us to the next data structure, sets.

Activity 1: Handling Lists

In this activity, we will generate a **`list`** of random numbers and then generate another **`list`** from the first one, which only contains numbers that are divisible by three. Repeat the experiment three times. Then, we will calculate the average difference of length between the two lists.

These are the steps for completing this activity:

1. Create a **`list`** of 100 random numbers.
2. Create a new **`list`** from this random **`list`**, with numbers that are divisible by 3.
3. Calculate the length of these two lists and store the difference in a new variable.
4. Using a loop, perform steps 2 and 3 and find the difference variable three times.
5. Find the arithmetic mean of these three difference values.

> **Note**
>
> The solution for this activity can be found on page 282.

Sets

A set, mathematically speaking, is just a collection of well-defined distinct objects. Python gives us a straightforward way to deal with them using its **`set`** datatype.

Introduction to Sets

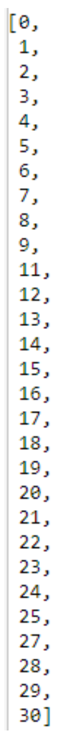
With the last list that we generated, we are going to revisit the problem of getting rid of duplicates from it. We can achieve that with the following line of code:

```
list_12 = list(set(list_1))
```

If we print this, we will see that it only contains unique numbers. We used the **`set`** data type to turn the first list into a set, thus getting rid of all duplicate elements, and then we used the **`list`** function on it to turn it into a list from a set once more:

```
list_12
```

The output will be as follows:



```
[0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 11,
 12,
 13,
 14,
 15,
 16,
 17,
 18,
 19,
 20,
 21,
 22,
 23,
 24,
 25,
 27,
 28,
 29,
 30]
```

Figure 1.12: Section of output for list_21

Union and Intersection of Sets

This is what a union between two sets looks like:

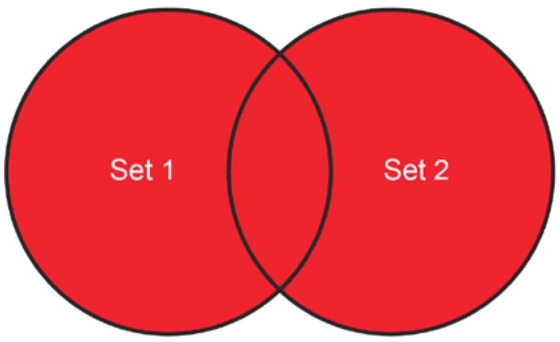


Figure 1.13: Venn diagram showing the union of two sets

This simply means take everything from both sets but take the common elements only once.

We can create this using the following code:

```
set1 = {"Apple", "Orange", "Banana"}
set2 = {"Pear", "Peach", "Mango", "Banana"}
```

To find the union of the two sets, the following instructions should be used:

```
set1 | set2
```

The output would be as follows:

```
{'Apple', 'Banana', 'Mango', 'Orange', 'Peach', 'Pear'}
```

Notice that the common element, Banana, appears only once in the resulting set. The common elements between two sets can be identified by obtaining the intersection of the two sets, as follows:

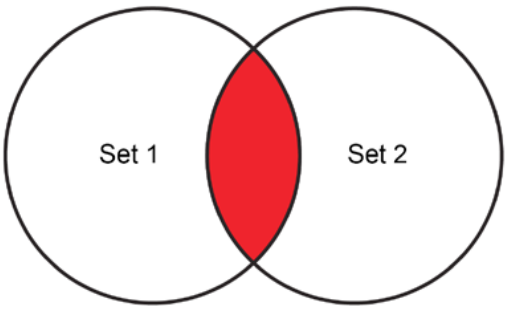


Figure 1.14: Venn diagram showing the intersection of two sets

We get the intersection of two sets in Python as follows:

```
set1 & set2
```

This will give us a set with only one element. The output is as follows:

```
{'Banana'}
```

> **Note**
>
> You can also calculate the difference between sets (also known as complements). To find out more, refer to this link: https://docs.python.org/3/tutorial/datastructures.html#sets.

Creating Null Sets

You can create a null set by creating a set containing no elements. You can do this by using the following code:

```
null_set_1 = set({})
null_set_1
```

The output is as follows:

```
set()
```

However, to create a dictionary, use the following command:

```
null_set_2 = {}
null_set_2
```

The output is as follows:

```
{}
```

We are going to learn about this in detail in the next topic.

Dictionary

A dictionary is like a list, which means it is a collection of several elements. However, with the dictionary, it is a collection of key-value pairs, where the key can be anything that can be hashed. Generally, we use numbers or strings as keys.

To create a dictionary, use the following code:

```
dict_1 = {"key1": "value1", "key2": "value2"}
dict_1
```

The output is as follows:

```
{'key1': 'value1', 'key2': 'value2'}
```

This is also a valid dictionary:

```
dict_2 = {"key1": 1, "key2": ["list_element1", 34], "key3": "value3",
          "key4": {"subkey1": "v1"}, "key5": 4.5}
dict_2
```

The output is as follows:

```
{'key1': 1,
 'key2': ['list_element1', 34],
 'key3': 'value3',
 'key4': {'subkey1': 'v1'},
 'key5': 4.5}
```

The keys must be unique in a dictionary.

Exercise 6: Accessing and Setting Values in a Dictionary

In this exercise, we are going to access and set values in a dictionary:

1. Access a particular key in a dictionary:

   ```
   dict_2["key2"]
   ```

   This will return the value associated with it as follows:

   ```
   ['list_element1', 34]
   ```

2. Assign a new value to the key:

   ```
   dict_2["key2"] = "My new value"
   ```

3. Define a blank dictionary and then use the key notation to assign values to it:

   ```
   dict_3 = {}  # Not a null set. It is a dict
   dict_3["key1"] = "Value1"
   dict_3
   ```

   The output is as follows:

   ```
   {'key1': 'Value1'}
   ```

Exercise 7: Iterating Over a Dictionary

In this exercise, we are going to iterate over a dictionary:

1. Create **dict_1**:

```
dict_1 = {"key1": 1, "key2": ["list_element1", 34], "key3": "value3",
"key4": {"subkey1": "v1"}, "key5": 4.5}
```

2. Use the looping variables **k** and **v**:

```
for k, v in dict_1.items():
    print("{} - {}".format(k, v))
```

The output is as follows:

```
key1 - 1
key2 - ['list_element1', 34]
key3 - value3
key4 - {'subkey1': 'v1'}
key5 - 4.5
```

> **Note**
>
> Notice the difference between how we did the iteration on the list and how we are doing it here.

Exercise 8: Revisiting the Unique Valued List Problem

We will use the fact that dictionary keys cannot be duplicated to generate the unique valued list:

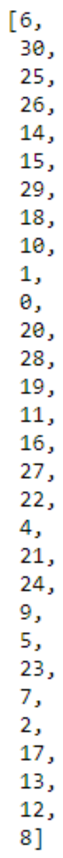
1. First, generate a random list with duplicate values:

```
list_1 = [random.randint(0, 30) for x in range (0, 100)]
```

2. Create a unique valued list from **list_1**:

```
list(dict.fromkeys(list_1).keys())
```

The sample output is as follows:



```
[6,
 30,
 25,
 26,
 14,
 15,
 29,
 18,
 10,
 1,
 0,
 20,
 28,
 19,
 11,
 16,
 27,
 22,
 4,
 21,
 24,
 9,
 5,
 23,
 7,
 2,
 17,
 13,
 12,
 8]
```

Figure 1.15: Output showing the unique valued list

Here, we have used two useful functions on the dict data type in Python, **`fromkeys`** and **`keys`**. **`fromkeys`** creates a dict where the keys come from the **`iterable`** (in this case, which is a list), values default to None, and **`keys`** give us the keys of a dict.

Exercise 9: Deleting Value from Dict

In this exercise, we are going to delete a value from a **`dict`**:

1. Create **`list_1`** with five elements:

   ```
   dict_1 = {"key1": 1, "key2": ["list_element1", 34], "key3": "value3",
             "key4": {"subkey1": "v1"}, "key5": 4.5}
   dict_1
   ```

   The output is as follows:

   ```
   {'key1': 1,
    'key2': ['list_element', 34],
    'key3': 'value3',
    'key4': {'subkey1': 'v1'},
    'key5': 4.5}
   ```

2. We will use the **`del`** function and specify the element:

   ```
   del dict_1["key2"]
   ```

   The output is as follows:

   ```
   {'key3': 'value3', 'key4': {'subkey1': 'v1'}, 'key5': 4.5}
   ```

> **Note**
>
> The **`del`** operator can be used to delete a specific index from a list as well.

Exercise 10: Dictionary Comprehension

In this final exercise on **`dict`**, we will go over a less used comprehension than the list one: dictionary comprehension. We will also examine two other ways to create a **`dict`**, which will be useful in the future.

A dictionary comprehension works exactly the same way as the list one, but we need to specify both the keys and values:

1. Generate a dict that has **`0`** to **`9`** as the keys and the square of the key as the values:

   ```
   list_1 = [x for x in range(0, 10)]
   dict_1 = {x : x**2 for x in list_1}
   dict_1
   ```

The output is as follows:

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Can you generate a **`dict`** using **`dict`** comprehension where the keys are from **`0`** to **`9`** and the values are the square root of the keys? This time, we won't use a list.

2. Generate a **`dictionary`** using the **`dict`** function:

```
dict_2 = dict([('Tom', 100), ('Dick', 200), ('Harry', 300)])
dict_2
```

The output is as follows:

```
{'Tom': 100, 'Dick': 200, 'Harry': 300}
```

You can also generate **`dictionary`** using the **`dict`** function, as follows:

```
dict_3 = dict(Tom=100, Dick=200, Harry=300)
dict_3
```

The output is as follows:

```
{'Tom': 100, 'Dick': 200, 'Harry': 300}
```

It is pretty versatile. So, both the preceding commands will generate valid dictionaries.

The strange looking pair of values that we had just noticed ('Harry', 300) is called a **`tuple`**. This is another important fundamental data type in Python. We will learn about tuples in the next topic.

Tuples

A tuple is another data type in Python. It is sequential in nature and similar to lists.

A tuple consists of values separated by commas, as follows:

```
tuple_1 = 24, 42, 2.3456, "Hello"
```

Notice that, unlike lists, we did not open and close square brackets here.