PYTHON SERIES

LEARNING PANDAS FROM ZERO

A self-learning minibook for Python Pandas

SHING-CHI LEUNG

Learning Pandas from Zero

Shing-Chi Leung

July 1, 2021

A self-learning minibook for Python Pandas

July 2021

Version 0.9.0

Contents

Preface

Welcome to my first programming textbook Learning Pandas from Zero! This is a short tutorial note style textbook aiming at beginning and intermediate user. It aims at helping you pick up Python library Pandas for you to process dataset and prepare data for the later machine learning.

In writing this textbook, I keep in mind to introduce the functionality and structure of the Pandas library progressively. Methods that are used in examples are introduced and explained beforehand. In the first 8 chapters, I focus on introducing the fundamental use of Pandas. The basic procedure in how one can operates on a Pandas Dataframe is introduced one by one. The methods and attributes are grouped accordingly to their functionality so that the entire chapter can be organically relevant. Then in the last 4 chapters, I focus on blending the fundamental skills with new methods for applications. In the examples I designed, I try to demonstrate how differ-

ent methods are used together in order to do a complete data transformation.

In each chapter, we will first briefly summarize the material in the previous chapter, then we proceed to introducing new methods and attributes for dataframe. Each one is accompanied with a number of examples and demonstrations. Each chapter is ended with a series of exercises to help the readers to master the new techniques.

There are a number of motivations for me to start this book project. The first one comes from my learning experience of Pandas. After applying Pandas for a many different private data science project, I started to retrospect with myself how I learned Pandas in the first place. It turns out the process is very bumpy. Most of the time I did not use it efficiently because the approached I tried to do does not exactly fit the data structure of Pandas. Thus I want to gather my experience in how to apply Pandas efficiently for various scenarios, so that readers will not need to repeat the same trouble in their learning experience.

The second one is to demonstrate the versatility of Pandas. The more I worked on my data science project, the more I feel that the same results can be achieved in many different ways, depending on the data available. Knowing all these ways can help us to use Pandas more fluidly without hard memorizing the specific procedures. In this book I try to explore with the readers how this

can be done.

To use this book, we can simply go through the chapters one by one, read the examples and demonstrations, and try to repeat them on your own Python compiler (Jupyter Notebook suggested). It will be useful to work out the exercise and check if you can reach the solution confidently! If so, then let's move on to the next chapter. If no, try to work out a few more exercises to test your understanding.

Finally I want to dedicate this mini-textbook to my family. I received a lot of support when I wrote my very first book project as a part of my portfolio.

Most of the learning materials and update are available on my research homepage[1] and GitHub repository[2]. Feel free to take a look there and see if you can find something interesting!

Shing-Chi Leung
Pasadena, California
28 June 2021

[1] https://sites.google.com/view/scleung
[2] https://github.com/scleung-astro

Chapter 1

Introduction

In this chapter we will briefly introduce the background of the Python package Pandas. We will go through the essential features of Pandas, its pros and cons, and some pre-requisite knowledge before our exploration in Pandas.

1.1 What is Pandas?

Pandas is the abbreviation of "Python Data Analysis" or "Panel Data". The connection between the full terms and the name "Pandas" is actually indirect. The name Pandas is designed for easy memorization of this package. Panel data is a terminology often used in economics and statistics. It corresponds to a set of data taken at different time for one or more quantities observed. Panel

Table 1.1: A sample panel data (units omitted)

time	temperature 1	temperature 2
1	28	36
2	29	34
3	32	30

data is usually presented in the form of a table. Shown in Table 1.1, we show a small panel data example where the temperature of two places at different time is presented.

Pandas is developed by Wes McKinney from 2007 - 2010 for the analysis of financial data. It is developed on NumPy and Cython for making use the efficient array manipulation in C and C++ language, meanwhile implementing functionality of creating, modify and structuring tables. It also contains a number of statistical and graph plotting tools for providing a first diagnosis of the data.

In the literature, panel data is often referred as a relational database. Its opposite is the non-relational database. In a relational database, all data consists of a unique key, followed by a range of attributes as columns. It is possible to visualize a relational database in the form of a table. The first column corresponds to the key (or the index of the entry) and other columns are the attributes. In our example of Table 1, the time can be regarded as the key while temperature 1 and 2 are the attributes.

1.2 When do we use Pandas?

As its name Pandas (panel data) suggested, Pandas is a package specific for handling table type data. In the Ecosphere of Python packages (Figure 1.1 below), Pandas belongs to the middle part of the system. The package is not targeting for advanced numerical processing such as machine learning (e.g., scikit-learn) or network analysis (network X), nor is it a fundamental package such as NumPy which aims at handling basic data structure (array in this case). It serves as a data processing tool which assists us in handling a large dataset and processes the dataset such that it becomes suitable for further analysis by other numerical packages.

There are four scenarios where Pandas is highly specialized for its operation.

- Time series

  When we need to handle a set of time series data, which can be a collection of temperature of different places at a range of time, or a collection of the closing prices of many stocks at different days, Pandas contains a number of useful tools for rapidly digesting the data within a few lines of code.

- Tables

  As described above, the major aim of Pandas is to process a dataframe. In Pandas, a table is referred

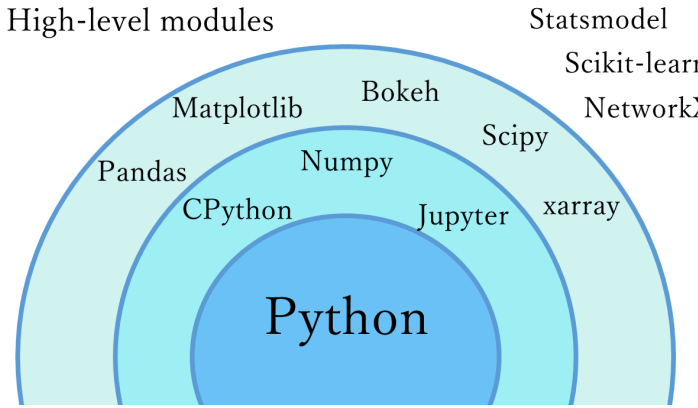


Figure 1.1: Schematic diagram of Python and its libraries. Modified from the figure from here with simplification for the chapter here.

as a dataframe. When we need to handle a large table such as the academic results of all students in a school, or the transactions record of a company within a year, the C-binding feature of pandas will have a much superior performance to process the data efficiently.

- Data analysis

  Pandas contains a number of statistical tools such as counting, average, variance, correlation and so on which help users to extract some preliminary

statistical features in the data. It also contains tools to help users remove or handle missing data. It also has the backend of Matplotlib where users can generate sophisticated visualization within a few lines of codes. Furthermore, it keeps the flexibility of fine-tuning the figures for presentation in different scenarios.

- Data processing

  As a tool for panel data or relational database, Pandas allows users to make query to find and to select a subset of the table for further processing. It contains method to let user merge multiple tables or to split a table into multiple tables easily. On top of that, there are options such as passing data through a set of procedure as a pipeline or generating a pivot table from the dataframe. Many of these methods provide an alternative view to the data to assist us to understand the data better.

1.3 Features of Pandas

Efficiency

The C-binding of Numpy, used by Pandas helps Python to loop over a large array efficiently. This feature is particularly notable compared to other standard spreadsheet

software, when the table size reaches above 10000 lines or above. Time for scrolling in the database for selecting or filtering any necessary data increases. It also takes time for the system to handle the graphical interface. On the other hand, the pure scripting style in Python, and hence Pandas, makes it much faster to go through essential items in a table.

Convenience

The integrated environment (data process, analysis and visualization) allows users to achieve a number of tasks by only using Pandas. The notation, which is similar to MySQL in data query, provides a direct approach to access, select and filter the desired information from a large dataset.

Simplicity

One of the main benefits of using Pandas is its easy-to-learn commands. Most commands are written in a pythonic way, and therefore the codes can be presented in a human-readable way. This makes the details of the data process very clear. This feature is particularly clear when we use workbook-like platform such as Jupyter notebook. We can examine and store the intermediate results so that we can design and present a clean procedure for others to understand the logic of the data process.

1.4 Other packages

Even though this ebook is fully on Pandas, I should also point out that there are other numerical packages in Python which is used in handling a large dataset. One of which is Datatable. The syntax in Datatable is very similar to MySQL such as the use of **SELECT**, **WHERE**, **ORDER BY** and so on. However, Datatable does not fully replicate all functionality from SQL-type software. It does not contain statistical analysis tools and visualization backend which Pandas contains. Thus Datatable is more specialized in processing large datasets.

1.5 Before using Pandas

Pandas is a package of Python. Certainly a Python Interpreter is indispensable for generating the results presented in this book. For readers interested in trying all the code on their own python Interpreter (or in some integrated environment such as Visual Studio Code or Jupyter Notebook), here are the setting of my Python interpretor and packages.

- Python 3 (version 3.9.2)
- NumPy (version 1.18.5)
- Pandas (version 1.0.3

- MatPlotLib (version 3.3.2)

In case when the above packages are not yet installed in your machine, one can simply type

```
!pip install pandas
```

to install the library accordingly through the Jupyter Notebook interface, or

```
pip install pandas
```

in the Anacornda Prompt. In both cases, the installation only needs to be done once before the first time we use the library. Other libraries can be installed in the same manner.

Nowadays software is updating in an unprecedented rate that almost everyday there are some new patches for some software package available. It is very possible that the exact version of your package when you read this book, the version is different. As long as the functionality described is available in your version of package, it will be fine for the practice purpose.

1.6 The Best Way to Learn

Just like all other textbooks in programming you may have encountered, the best way to make the data processing skills using Pandas will be to practice while you

read this book! I encourage all readers to follow the short code examples and type them on your own machine for practice. Besides the code examples, I also included a number of coding exercises at the end of each chapter. Certainly, I strongly encourage all readers to try solving one or two to check that you have grasped the new ideas in the first reading.

Chapter 2

Creating your first dataframe

In this chapter, we will go through the fundamental steps about how to construct your first tables. We will examine the methods **DataFrame** and **read_csv** and go through how we can convert different types of data into a dataframe.

2.1 Why DataFrame?

Before we start, we need to go through the notation **DataFrame**. Dataframe is the main object class used in Pandas. Tables are stored and processed as dataframes. Each entry is stored as a row with a unique index as

an identifier. The dataframe class contains a lot of useful methods and attributes to help you deal with many routine operations.

In Pandas, another main datatype is called **Series**. It is a one-dimensional array data with each entry corresponding to a key. A series can be viewed as a slice of a dataframe. Since most options available in series are a subset of that of the DataFrame, we do not explicitly discuss series in this book.

2.2 Build a table from a dictionary

Now let us assume this scenario. You have a table as following (Table 2.1). The table contains data about the number of different fruits in some shops. Instead of using the more familiar Microsoft Excel to handle, we want to experiment with Pandas and see what we can do with this package.

The first step is to pass this set of data. The first way is to make this set of data into a dictionary. Then we make a dataframe object by passing the dictionary into this object, namely

```
df = pandas.DataFrame(dictionary, index=...,
    columns=...)
```

Table 2.1: The reference dataframe containing the amounts of fruits surveyed in different shops

Shop	Apple	Orange	Kiwi
A	18	23	30
B	23	18	29
C	31	37	30

The first argument is compulsory, it can be an iterable such as a list, array or dictionary. Here we consider a dictionary first. There are a number of auxiliary argument you can pass during declaration. But do not worry if the immediate output of the Dataframe does not fit your expectation. We can also set it after the declaration. Here we consider the simplest case first.

To make the above Table 2.1 into a dictionary which can be converted into a dataframe directly, we used the following code:

```
# not forget to import this library!
import pandas as pd

data = {
  "Shop":["A","B","C"],
  "Apple":[18,21,31],
  "Orange":[23,18,37],
  "Kiwi":[30,29,30]
}
```

```
In [82]: print(df)
```

	Shop	Apple	Orange	Kiwi
0	A	18	23	30
1	B	21	18	29
2	C	31	37	30

Figure 2.1: Output of the dataframe df

```
df = pd.DataFrame(data)
```

You might notice that in preparing the dictionary, we set the keys to be the header of all columns. The values for each key correspond to all values under that header. And notice that the dataframe can contain numerical and string inputs.

The results of df can be immediately checked by printing the object out (Figure 1).

One thing to notice is that Pandas by default assign integers 0, 1, 2 and so on as the keys for the entry when there is no specification during declaration. In some context they are referred as indices. We will return to this later in this chapter.

If you get the same as that shown in Figure 2.1, congratulations! This is the very first step in your excursion in Pandas.

We can check the data type of the variable **df** by typing

```
type(df)
```

and we will receive the class 'pandas.core.frame.DataFrame'.

2.3 Build a table from a list

The flexibility of Pandas allows us to choose other types of iterable as the data source. Now we repeat the process by creating a list which contains the data. We will use the following code to construct a list and then pass the list to the dataframe object.

```
import pandas as pd
data = [
  ["A",18,23,30],
  ["B",21,18,29],
  ["C",31,37,30]
]
column_names = ["Shop", "Apple", "Orange", "Kiwi"]
df = pd.DataFrame(data, columns=column_names)
```

By examining the code, there are four major differences compared to the above code for using dictionary, including:

- The square brackets

- Each entry in the list corresponds to the data from any given row
- No column headers passed in the list
- The column headers stored in a separate list

The output dataframe will be identical to Figure 2.1. Again, this will be very helpful to practice once on your own to experience the dataframe declaration process!

What if we omit the column setting?

In the version using list, you might have noticed the notation **columns=column_names** is used in order to obtain a dataframe which is identical to the version using dictionary. Then, what will happen if we do not pass this setting during declaration?

To experiment, let us modify the code a little by writing

```
import pandas as pd
data = [
  ["A",18,23,30],
  ["B",21,18,29],
  ["C",31,37,30]
]
column_names = ["Shop", "Apple", "Orange", "Kiwi"]
df = pd.DataFrame(data)
```

```
In [87]: print(df)
   0   1   2   3
0  A  18  23  30
1  B  21  18  29
2  C  31  37  30
```

Figure 2.2: Output of the dataframe df without column headers

When we print the dataframe again, this time we obtain a new dataframe in Figure 2.2.

It becomes clear that Pandas by default name all rows and columns by integers 0, 1, 2. When the user offers lists of column headers or row indices, Pandas will overwrite these default integers by the elements from those lists.

In light of that, one may be tempted to question if a similar treatment can be applied to the keys. The answer is... YES! By passing a list which contains the names to be used for the indices, the output dataframe will use the elements from the list automatically. In fact, giving meaningful row index and column headers will be very useful for processing the dataframe effectively as we will show in later chapters.

To do so, we first modified the list version of the code as follows:

```
In [89]: print(df)
```

	Shop	Apple	Orange	Kiwi
Shop 1	A	18	23	30
Shop 2	B	21	18	29
Shop 3	C	31	37	30

Figure 2.3: Output of the dataframe df with index and column names

```
import pandas as pd
data = [
  ["A",18,23,30],
  ["B",21,18,29],
  ["C",31,37,30]
]
column_names = ["Shop", "Apple", "Orange", "Kiwi"]
index_names = ["Shop 1", "Shop 2", "Shop 3"]
df = pd.DataFrame(data, columns=column_names,
    index=index_names)
```

When we examine the dataframe, we will obtain the result shown in Figure 2.3.

Other types of data available

It is also possible to pass a JSON type data, as it has a structure very similar to a dictionary, to create a dataframe. However, we do not discuss the detailed process as treating JSON type data will involve further methods used in **json** module and other web fetching interface in order to achieve a complete discussion.

2.4 Build a dataframe from a csv file

In the above examples, we have paid attention at how to create a table from scratch. However, this will mostly be applicable to a small-size table for self-practice. When we need to read a large dataset or datafile, we want to ask Pandas to read the source file for us. Here let us examine the case when the datafile is a csv file.

If you are new about what a csv file is, a csv file is the short form of comma-separated values file. It is a compact file format to store an array of data. Each line corresponds to one row of data, with a comma to separate between consecutive data. The first row is usually used as the header for the column names. Notice that no space is needed after each comma or otherwise the machine might confuse the comma as a necessary part in the string. In

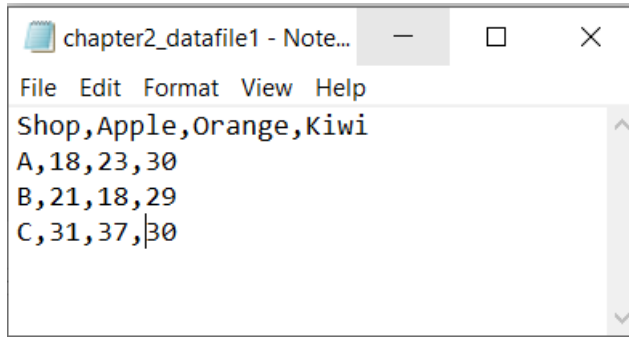


Figure 2.4: An example of a csv file

Figure 4 we show an example where we convert the table used in this chapter into a csv file.

Python has package specific for processing and reading csv files by using the **csv** library. For Pandas, we do not need to call that module explicitly because Pandas contains all tools necessary for configuring and accessing csv file. We use the command

```
df = pandas.read_csv(file, header=...,
    index_col=...)
```

The filename (or with the path to the file) is the compulsory argument for this command. Others are auxiliary and can be set according to the users' specific needs. Some useful ones include:

- header: An integer which tells Pandas not to read the line before that, and make the header line as column names
- index_col: An integer which tells Pandas not to set that column as part of the dataframe, but to use that column as the index

One can find the documentation of the setting at here. Since this book serves as the introductory guide to grasp the essential technique useful in Pandas, we will not discuss each configuration one-by-one.

By using the above csv file example, we can set up the code to read the csv file by

```
import pandas as pd
filename = "chapter2_datafile1.csv"
df = pd.read_csv(filename)
```

The procedure is simpler than using list and dictionary because the effort in preparing the data is done while preparing the csv file. We will obtain a dataframe identical to Figure 2.1 of this chapter.

2.5 Build a dataframe from an excel file

The last command we want to discuss in this chapter is to ask pandas to read an excel file. It usually becomes a

dilemma when there is a spreadsheet software available, why bothers to use Pandas again? There are two reasons here. First, when the table is large (10000 lines or above), the performance of Pandas is much better than standard spreadsheet software because Python does not need to handle the expensive graphical user interface in the operation. Most resource is used for only storing and processing the data.

Second, in standard spreadsheet software like Excel, graph plotting and other calculation can be limited by the design. The setting in Pandas can be fine-tuned: for example, the colour template in each plot or choice of partial data to be plotted, can be adjusted one by one.

Again, we will use the same table (Table 2.1) as a reference to demonstrate how to read the excel file. We will first prepare an excel file containing all entry in the table (Figure 2.5).

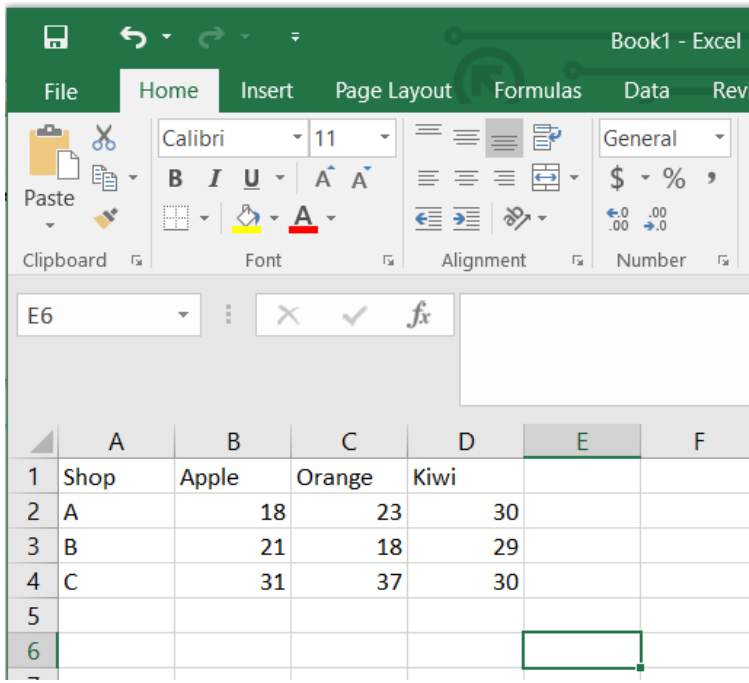
Then we will use the command

```
df = pandas.read_excel(file, sheet_name=...,
    header=..., index_col=...)
```

to convert the data in the excel table to a dataframe. The first argument *file* is compulsory which is the file name, others are auxiliary settings which depend on the need.

Some useful settings include:

- sheet_name: integer or a list of integer, the name of the sheet to be converted in the excel file



	A	B	C	D	E	F
1	Shop	Apple	Orange	Kiwi		
2	A	18	23	30		
3	B	21	18	29		
4	C	31	37	30		
5						
6						

Figure 2.5: An example of a Excel file

- header: integer, similar to read_csv, it controls Pandas to skip previous lines and set the header line for column names

- index_col: integer, similar to read_csv, it controls Pandas to pick a selected columns as the keys of each entry

```
import pandas as pd
filename = "chapter2_datafile2.xlsx"
df = pd.read_excel(filename)
```

When we output the datafile to check, we will obtain identical results as Figure 2.1.

2.6 Summary

In this chapter we have explored multiple ways to generate a dataframe by using data available in a list, a dictionary, a csv file and an excel file. We have learnt that the index (key) and column names are interchangeable in Pandas, and can be named arbitrarily. We have covered three essential methods in the Pandas package, including:

```
# Passing iterables to a dataframe
df = pandas.DataFrame(data, ...)
```

Table 2.2: Caption

Name	ID	Age	Gender
Ann	A01	20	F
Ben	A02	35	M
Carla	A03	28	F
Doug	A04	31	M

```
# Read a csv file and generate a dataframe
df = pandas.read_csv(csv_file, ...)

# Read an excel file and generate a dataframe
df = pandas.read_excel(excel_file, ...)
```

2.7 Exercises

The exercises of this eBook has two objectives. First, they serve as a programming exercises to some of the coding skills presented in that chapter. Second, they serve as a prompt for readers to explore different functionality available in Pandas with real examples. I encourage all readers to attempt a few questions to confirm the methods and attributes are well mastered in each chapter.

1. Your friend gave you a table (see Table 2.2).

   Try to use

   - a dictionary,

- a list,
- a csv file and
- an excel file

to build a dataframe object used the data.

2. If we want to make the output similar to Figure 2.3 for the case of a csv or an excel file in Exercise 1, which setting should we use? Can you do it by adding a new list or by modifying the csv file?

3. Your friend tried to copy the code and made a csv file for that. Now your friend wants to make the table shorter by skipping the first two entries (i.e., without the entries of Ann and Ben). What will you suggest?

Chapter 3

Calling elements in the DataFrame

In the previous chapter we have examined multiple ways to set up a dataframe from an existing dataset of a table. We showed that by printing the dataframe, it displays the whole table. However, in general, we often need to access only a subset of the table. In this chapter, we will discuss some common approaches.

3.1 A quick recap of NumPy array

As we mentioned in Chapter 1, Pandas shares the same high-performance array manipulation features from NumPy.

Table 3.1: A table showing the position notation

Row	Column 0	Column 1	Column 2	Column 3
0	[0,0]	[0,1]	[0,2]	[0,3]
1	[1,0]	[1,1]	[1,2]	[1,3]
2	[2,0]	[2,1]	[2,2]	[2,3]

NumPy is the short form of Numerical Python. It is one of the fundamental packages on which many other advanced packages are built. One important feature of NumPy is the notation of array similar to the array appeared in compiled languages such as C, C++ and Fortran. The C-binding of NumPy allows Python to access multi-dimensional arrays similar to a standard list object. How to call certain elements in a NumPy array has many similarities to that in Pandas. Therefore, we shall first review this feature in NumPy and then extend it to Pandas dataframe.

We will focus on 2-dimensional NumPy arrays as they correspond naturally to the structure of a dataframe. For a 2-dimensional array (or a rank-2 array), the element at **i**-row and **j**-column in an array **A** has an index **A**[i, j] (See Table 3.1 below). There are three ways to address elements in an array.

1. Direct notation

   We can use the notation **A**[i,j] to refer to an element in a NumPy array **A**.

2. Colon notation

   When more than one element is referred to, similar to the list operation, a notation of **A**[i1:i2, j1:j2] may allow the user to points to a range of rows (i1, i1+1, i1+2, ..., i2-1) and columns (j1, j1+1, j1+2, ..., j2-1). Notice that the end values i2 and j2 are not included.

3. List notation

   When we want to access specific rows or columns in the array, and when they are not necessarily consecutive to each other, we can use a list to include the specific row and column numbers. For example, a notation **A**[[i1,i2,i3], j1] will correspond to an subarray made of the data in rows i1, i2, i3 and column j1. Notice that unlike colon notation, only one list can be used for either row or column.

Let us use a real example to illustrate these notations explicitly. Below, I will use NumPy to generate a table filled with random integers. We use an array filled with different numbers such that we can tell which rows and columns the numbers refer to. Here is the code

```
import numpy as np
a = np.random.randint(min=0, max=100, size=(5,3))
```

Let us examine the code script. It creates an array **a** which is a rank-2 array with 5 rows and 3 columns, filled

```
In [110]: print(a)

[[91  0 85]
 [97 14 17]
 [10 53 50]
 [50 69 17]
 [46 69 48]]
```

Figure 3.1: The random number array generated by my laptop

with random integer from 0 to 100. In my laptop, the array *a* has the form displayed in Figure 3.1. Because of its random nature, it is very likely that your values in the array are different from those shown here.

A short demonstration using the three notations listed above is shown in Figure 3.2.

3.2 Calling dataframe elements by their address

Now we can proceed to discuss the three ways to call the elements in a Pandas dataframe. For a dataframe, Pandas provides three distinctive methods to refer to the elements. This includes:

1. The **iloc** method

```
In [111]: a[2,1]

Out[111]: 53

In [112]: a[1:3,0:2]

Out[112]: array([[97, 14],
                 [10, 53]])

In [113]: a[[1,2,4],:]

Out[113]: array([[97, 14, 17],
                 [10, 53, 50],
                 [46, 69, 48]])
```

Figure 3.2: Query of the array **a** using direct, colon and list notation

2. The **loc** method
3. The column name as its index

The iloc method

The iloc method can be applied to a dataframe *df* by using the pattern:

```
df.iloc[row_indices, column_indices]
```

```
In [125]: df.iloc[[1,2],[1,2,3]]
```

Out[125]:

	Apple	Orange	Kiwi
Shop 2	21	18	29
Shop 3	31	37	30

Figure 3.3: Passing two lists to the **iloc** method to refer to specific rows and columns

The notation of the row and column indices follows the three notation styles we have described for NumPy arrays. For example, the call by

```
df.iloc[[2,3,5],1]
```

corresponds to the elements of rows 2, 3, 5 at column 1.

One major difference for **iloc** is that we can pass two lists of indices to select individual rows and columns simultaneously. We demonstrate this feature in Figure 3.3.

The loc method

Pandas provides another method in calling the elements. Instead of counting the row and column number in order to find which integer we should provide in the **iloc** method, Pandas let us pass the index and column names

(headers) as a representative of the integer index. Do you remember about the naming of indices and columns in Chapter 2? In fact, providing a meaningful names to the columns, will provide very much convenience in calling the data without frequently referring to the original dataset.

The **loc** method follows a similar pattern:

```
df.loc[row_indices, column_indices]
```

The three notations for NumPy array can also be used in this method. Now let us reuse the dataframe in Figure 2.2 of Chapter 2, we can use

```
df_loc["Shop 1":"Shop 2", "Apple":"Kiwi"]
```

to refer the rows from "Shop 1" to "Shop 2" and columns from "Apple" to "Kiwi". In Figure 3.4 we display the results on a Jupyter Notebook.

If we pay attention, we notice a minor difference compared to the **iloc** method. The colon notation here actually includes the end value, which is not the case for the colon notation in the **iloc** method and in NumPy arrays. Also, the **loc** method also allows using two lists for both row and column for reference (See Figure 3.5).

The column name as an index

The last method for referring to elements in a dataframe is to use the column names as the index. When we cre-

```
In [118]: df.loc["Shop 1":"Shop 2","Apple":"Kiwi"]
```

Out[118]:

	Apple	Orange	Kiwi
Shop 1	18	23	30
Shop 2	21	18	29

Figure 3.4: Calling subset of a dataframe using the **loc** method

```
In [121]: df.loc[["Shop 1","Shop 2"],["Apple","Orange","Kiwi"]]
```

Out[121]:

	Apple	Orange	Kiwi
Shop 1	18	23	30
Shop 2	21	18	29

Figure 3.5: Passing two lists to the **loc** method for referring to specific rows and columns

ate a dataframe, in the **loc** and **iloc** methods the first index refers to the rows of the dataframe. However, the convention changes when we treat the dataframe as an array. The object does not store the data rows by rows, instead, from its attribute, data is grouped column by column. That means, when we consider the same table as above and type

```
In [133]: df["Apple"]

Out[133]: Shop 1    18
          Shop 2    21
          Shop 3    31
          Name: Apple, dtype: int64
```

Figure 3.6: Referring to an "index" of a dataframe

```
df["Shop 1"]
```

will yield an error message from the Python interpreter that no such column exists. In fact, Pandas stores the whole column as an attribute or index in the Dataframe object. Therefore, it is valid when we type

```
df["Apple"]
```

And the interpret will return us the results shown in Figure 3.6. Notice that the output is a *Series* because the data itself is one-dimensional.

The list notation gives identical results as in the **loc** and **iloc** methods (See Figure 3.7).

However, the colon notation does not generate similar results. It only returns a frame containing the related column headers in the query. In Figure 3.8 we show the

```
In [131]: df[["Apple", "Kiwi"]]
```

Out[131]:

	Apple	Kiwi
Shop 1	18	30
Shop 2	21	29
Shop 3	31	30

Figure 3.7: Passing a list to the indices of a dataframe

```
In [131]: df[["Apple", "Kiwi"]]
```

Out[131]:

	Apple	Kiwi
Shop 1	18	30
Shop 2	21	29
Shop 3	31	30

Figure 3.8: Passing colon notation to the indices of a dataframe

results. This is an exception case we need to be careful during colon notation.

The column name as an attribute

Another minor note is that, we can also refer to the whole array as an attribute of the dataframe. Pandas returns the same column when we call by the column headers as an attribute of the dataframe. For example, we can input

```
df.Apple
```

to obtain the same series data as in Figure 6. However, this method has more constraints than passing the column name as an index. First, the attribute must not contain any symbol, space or numbers. Second, it only accommodates one column at a time. For operation containing multiple columns, this method does not fit.

3.3 Summary

In this chapter we have reviewed briefly three different notations to refer elements in a NumPy array elements, including:

- direct notation
- colon notation
- list notation

These notations are used in Pandas in three distinctive methods to call elements in a dataframe based on the three notations above. They include:

Table 3.2: Comparison of using three index notations in three element calling methods in Pandas

Notation	**loc** method	**iloc** method	index
direct notation	Yes	Yes	Yes
colon notation	Yes (end value)	Yes (no end value)	No
list notation	Yes	Yes	Yes

- the **loc** method (refer to the row indices and column headers)
- the **iloc** method (refer to the array position of the rows and columns)
- as an index or attribute (applicable to only columns)

Notice that there are minor differences in the exact operation using the three notations in these three methods. In Table 3.2 below, we list out how they are different from each other.

3.4 Exercises

1. Given the dataframe **df** below (See Table 3.3), work in mind and then on a your notebook the expected output.

Table 3.3: The expected output

Index	A	B	C	D	E
Ann2	31	33	35	37	39
Bart	11	13	15	17	19
Chris	21	23	25	27	29
Dane	41	43	45	47	49
Eaton	51	53	55	57	59

```
# exercises on iloc
df.iloc[2,4]
df.iloc[1:3,2:3]
df.iloc[[1,3,4],[3,2]]

# exercises on loc
df.loc["C","Bart"]
df.loc["Bart":"Chris", "C":"D"]
df.loc[["Dane", "Bart"]]

# exercises on column name as index
# and attribute
df["Ann2"]
df.Ann2
df["A":"B"]
df[["B","C"]]
```

2. Your friend wants to get an output as following

Table 3.4: The expected output

Index	E	B	A
Eaton	59	53	51
Chris	29	23	21
Ann2	39	33	31

(Table 3.4), which notation and method will you use? And what are the values of the passing rows and columns?

Chapter 4

Adding, editing and removing entries

In previous chapters, we have learnt how to create a dataframe and to query the values of elements in a dataframe. In this chapter, we will study how to modify the values of these elements. We will discuss how to add new rows and columns in a dataframe, then how to modify the values in a dataframe element-wise, row-wise and column-wise. Finally we discuss how to remove rows and columns. For the examples in this chapter, we shall reuse Table 3 in Chapter 2 for our discussion. Here we copy the original table in Figure 4.1 for our quick reference:

```
In [89]: print(df)
```

	Shop	Apple	Orange	Kiwi
Shop 1	A	18	23	30
Shop 2	B	21	18	29
Shop 3	C	31	37	30

Figure 4.1: The reference dataframe for this chapter

4.1 Adding new items

Adding new row(s)

Pandas allows adding a new row through the use of **loc** method. When we pass a row index and the corresponding column data, Pandas first check if the row name or row index exists in the dataframe, if yes, then Pandas uses the passed data to modify the existing entry; if not, Pandas generates a new row with the index provided as the key. For example:

```
df.loc["Shop 4"] = ["D", 10, 20, 30]
```

Since there is no row "Shop 4" existing in the dataframe **df**, Pandas will create a new row where the value of each column corresponds to the element in the list, as shown in Figure 4.2. However, while passing a list, the length

```
df.loc["Shop 4"] = ["D",10,20,30]
print(df)
```

```
        Shop  Apple  Orange  Kiwi
Shop 1    A     18      23    30
Shop 2    B     21      18    29
Shop 3    C     31      37    30
Shop 4    D     10      20    30
```

Figure 4.2: Add a row by using a new index key as a reference

of the list must match the columns exactly, otherwise Python regards the input as invalid and raises an error.

We can also pass a dictionary similar to how we declare a dataframe by using the method *append*. To do so, we need to define a dictionary where the key is the column header and the value to be the data for that column. The following script will generate the same dataframe as Figure 4.2.

```
new_entry = {"Shop":D, "Apple":10, "Orange":20,
    "Kiwi":30}
df = df.append(new_entry)
```

Pay attention that we need to use **df = df.append(...)** to make sure the dataframe containing the new entry.

It is because the method **df.append** returns the modified dataframe. It does not directly modify the original dataframe. If the new result is not stored in any variable, the appended dataframe will be lose after the script is run.

The **append** allows us to contains columns where no value is given. In that case, we need to specific the flag **ignore_index = True** and Pandas fills the blanks by NaN. For example, it is valid to append the following dictionary.

```
new_entry = {"Shop":D, "Apple":10, "Kiwi":30}
df = df.append(new_entry, ignore_index=True)
```

The output dataframe is given in Figure 4.3.

When we need to add more than one rows to the array, it is more convenient to create a new dataframe containing the new entries, and concatenate the new dataframe to the original one to form a larger dataframe. We demonstrate this method below.

```
# pay attention to the list for each dictionary
# entry!
new_df = pd.DataFrame({"Shop":[D], "Apple":[10],
    "Kiwi":[30]})
df = df.append(new_df)
```

Pandas includes another function **concat** which has a similar format. It provides another functionality com-

```
#new_entry = {"Shop":"D",
   "Apple":10, "Kiwi":30}
df.append(new_entry, ignore_index=True)
```

	Shop	Apple	Orange	Kiwi
0	A	18	23.0	30
1	B	21	18.0	29
2	C	31	37.0	30
3	D	10	NaN	30

Figure 4.3: Adding new rows with blanks to the dataframe

pared to **append** that it allows adding new rows and columns, and flexibility when a dataframe with multiple indices is passed. For the time being, we focus on the basic use of this function.

```
new_df = pd.DataFrame({"Shop":[D], "Apple":[10],
    "Kiwi":[30]})
df = pd.concat([df, new_df])
```

Adding new column(s)

In Pandas, remember in Chapter 3 we have discussed that each column is an attribute or an index of the dataframe. As a result, adding new columns are equal to adding a new attribute to the dataframe object. Using the dataframe in Figure 4.1 as an example, we can do the following operation to add a column:

```
df["Pineapple"] = [2, 3, 5]
```

The output dataframe is shown in Figure 4.4. Similar to adding a new row by a list, the list here must match the number of rows in the original dataframe.

Notice that we can add the new column by assigning the data as an attribute of the dataframe.

```
df.Pineapple = [2, 3, 5]
```

Again, the column header used by this approach allows only alphabets. No numeric, symbols or space is allowed.

Similar to adding rows, the function **concat** provides another way to add one or more columns efficiently. Notice that we use the parameter **axis=1** in the method. By default **axis=0** corresponds to adding the new dataframe as new rows. To tell Pandas that we are adding new columns, we set **axis=1**.

```
df["Pineapple"] = [2, 3, 5]
df
```

	Shop	Apple	Orange	Kiwi	Pineapple
Shop 1	A	18	23	30	2
Shop 2	B	21	18	29	3
Shop 3	C	31	37	30	5

Figure 4.4: Adding a new column by the index feature of Pandas

```
new_df = pd.DataFrame({"Pineapple":[2,3,5]},
    index=["Shop1","Shop2","Shop3"])
df = pd.concat([df, new_df], axis=1)
```

Here, one may observe that we need to pass the index name for the dataframe **new_df**. It is because when Pandas merges two tables, it uses the index as the key to bind the two tables. When no index name is passed, the dataframe **new_df** contains index (0, 1, 2) only. Thus when the two dataframes merge, Pandas cannot find the index (0, 1, 2) in the dataframe **df**. What Pandas will do is adding three new entries with these indices and it results in a dataframe of 6 rows.

```
In [171]: df.loc["Shop 2":"Shop 3", "Apple"] = [2,3]
          df
```

Out[171]:

	Shop	Apple	Orange	Kiwi
Shop 1	A	18	23	30
Shop 2	B	2	18	29
Shop 3	C	3	37	30

Figure 4.5: Modifying elements in a dataframe

4.2 Editing a dataframe

Modifying elements

We have introduced the notation **loc** and **iloc** for declaring and adding entries in a dataframe. These two functions can also be used to modifying the values in a dataframe accordingly. The new values will overwrite the original one. In Figure 4.5 we show the modified dataframe.

```
df.loc["Shop 2":"Shop 3", "Apple"] = [2, 3]
```

We remind that the passed list needs to conform with the implied array size by the colon notation or list notation.

```
df.loc["Shop 2"] = ["D", 15, 17, 19]
print(df)
```

```
        Shop  Apple  Orange  Kiwi
Shop 1    A     18      23    30
Shop 2    D     15      17    19
Shop 3    C     31      37    30
```

Figure 4.6: Replacing a row in a dataframe

Modifying rows

The modification of rows can be done easily similar to adding new rows. By using the existing index as a reference, we can modify the whole row according. For example, using the dataframe in Figure 4.1, the following script

```
df.loc["Shop 2"] = ["D", 15, 17, 19]
```

will rewrite the original dataframe to that shown in Figure 6.

When we replace multiple rows, we can use a list to store all the rows and pass the corresponding index names in a loop.

```
df.Kiwi = [35, 12, 37]
print(df)
```

```
       Shop  Apple  Orange  Kiwi
Shop 1    A     18      23    35
Shop 2    B     21      18    12
Shop 3    C     31      37    37
```

Figure 4.7: Replacing a column in a dataframe

Modifying columns

The operation is again very similar to adding a new column as discussed above. By stating the column names as the index of the dataframe (or as an attribute), we can pass the modified data to the dataframe and overwrite the original dataset. By adding the following script to the dataframe (Figure 4.1), we obtain a new **Kiwi** column as show in Figure 4.7.

```
df.Kiwi = [35,12,37] # or df["Kiwi"]
```

The length of the list to be passed in this manner again needs to respect the original number of rows in the dataframe.

4.3 Removing entries

In Pandas, we use the command **drop** which is one of the dataframe object methods. It has the form

```
df.drop(labels=..., axis=..., index=...,
    columns=..., inplace=...)
```

This method is flexible that there are different ways to specify which rows or columns to be removed. The notation is very similar to those when we generate a new dataframe. The **axis** again specifies whether the names in **labels** are the row names of index names similar to **concat**. We need to specify the axis to tell Pandas whether we are deleting rows (axis=0) or columns (axis=1).

Here we introduce an important parameter **inplace**. This is a boolean parameter (True or False). Whenever, we do operation like this to a dataframe, the operation returns a new dataframe which is modified. However, the new dataframe does not overwrite the original data by default. The following two scripts are equivalent:

```
df(..., inplace=True)
```

and

```
df = df(..., inplace=False)
```

For example, we may delete the column "Kiwi" in the Table by:

```
df.drop(labels="Kiwi", axis=1,
    inplace=True)
df
```

	Shop	Apple	Orange
Shop 1	A	18	23
Shop 2	B	21	18
Shop 3	C	31	37

Figure 4.8: Delete a column in a dataframe

```
df.drop(labels="Kiwi", axis=1, inplace=True)
```

The modified dataframe **df** is shown in Figure 4.8. When we want to delete multiple rows or columns, we pass a list instead of a string to **labels**.

4.4 Modifying index and column names

Besides the elements in the dataframe, we can also change the index and column name by the method **rename** of the dataframe. Renaming can be important if we want

to process rows or columns in a batch. The format is given by:

```
df.rename(mapper=..., index=..., columns=...,
    axis=..., inplace=...)
```

Again, there are more than one way in using the rename operation. Either one of mapper, index and columns can be chosen to pass the dictionary where the key is the original names in the dataframe and the value being the new names. When **mapper** is passed, the axis value (0 for row and 1 for column) is needed for specification. The **inplace** allows us to change the original dataframe directly. The following two commands give the same results for the dataframe **df**:

```
df.rename(mapper={"Shop":"Shop code",
    "Apple":"Pineapple"}, axis=1, inplace=True)
```

and

```
df = df.rename(columns={"Shop":"Shop code",
    "Apple":"Pineapple"})
```

Both commands will modify the dataframe in the state displayed in Figure 4.9.

```
df.rename(mapper={"Shop":"Shop code",
   "Apple":"Pineapple"},
   axis=1, inplace=True)
df
```

	Shop code	Pineapple	Orange	Kiwi
Shop 1	A	18	23	30
Shop 2	B	21	18	29
Shop 3	C	31	37	30

Figure 4.9: Renaming columns in a dataframe

4.5 Exercises

1. We want to edit the dataframe shown in Table 4.1. You receive this table which contains the information of the new staffs of some companies. Use the procedures described above to implement changes to the table. Try to use as many ways as you can for practice.

    - Add a new staff with the following information Name: Fox, ID: A06, Branch: New York, Age: 40, Gender: F

Table 4.1: Reference dataframe for Exercises 1 and 2

Staff	ID	Branch	Age	Gender
Ann	A01	New York	26	F
Bart	A02	New York	35	F
Chris	A03	Los Angeles	27	F
Doug	A04	Los Angeles	34	M
Eaton	A05	Los Angeles	41	M

- Change the name Bart to Barth
- Change the gender of Barth and Chris to male.
- Replace the information of Eaton as follows:
  Name: Eaton, ID: A07, Branch: New York, Age: 38, Gender: F
- Change the branch of Ann and Bart to Seattle and their gender to M
- Remove the entry of Eaton
- Remove the column "Gender"

2. One of the most tricky parts of handling a large database is the missing values. In Pandas we can pass the dataframe with missing values first and then fill the missing value with some pre-defined values. One useful dataframe method **fillna** does all the work to fill in the blank space by the designated value which appears in the table.

Table 4.2: New entry for the dataframe

Name	ID	Age	Gender
Fox	A06	40	M

```
df.fillna(value=..., inplace=...)
```

The argument value is the number or string or list/dictionary you assign to replace the entry with a "NaN", inplace makes sure that your changes are stored in the original dataframe. Let us experiment with this dataframe method in this exercise. Let us start from the dataframe which contains Table 4.1.

a. Add a new staff with the information contained in Table 4.2:

Do you see there is one column now filled with NaN?

b. Now use the method *fillana* on this dataframe object and replace the NaN by the String "undefined"

c. Then we know now Fox will move to Los Angeles for his branch. Change the table accordingly.

3. In Figure 4.5, we have demonstrated how to modify a dataframe by using the **loc** method. If we want to use **iloc** instead, what are the indices to be passed?

Chapter 5

Describing a dataframe

In previous chapters we have learnt how to create, calling and modifying a dataframe. That will be sufficient for the most of our needs in managing a table. Now let's move on to something more advanced! In this chapter, we will discuss some of the useful Pandas methods which summarize and describe the features about the dataframe. These methods assist us to access the key features of the dataframe efficiently without really going over the dataframe by naked eyes. For demonstration purpose, we will use the Table 3 in Chapter 2 again in most of the examples here. In Figure 5.1 I show the table again for a quick reference.

```
In [89]: print(df)
```

	Shop	Apple	Orange	Kiwi
Shop 1	A	18	23	30
Shop 2	B	21	18	29
Shop 3	C	31	37	30

Figure 5.1: The reference dataframe iof this chapter

5.1 Dataframe size and dimension

The dataframe in Pandas inherits many features from the NumPy arrays. We can access the number of rows and columns by the shape attribute

```
df.shape
```

For example, the dataframe of Figure 5.1 has a shape (3, 4). This corresponds to 3 rows and 4 columns.

And we can access the number of rows by using

```
len(df)
```

but notice that it cannot access the number of columns directly. If we really need to use this method to access the number of columns, we first extract one row by **loc** or **iloc**, and then use the **len** method, i.e.

```
len(df.loc["Shop 1"])
```

Notice that the choice of row does not matter.

A less often used attribute **size** reports the number of elements in the dataframe:

```
df.size
```

For example, the table in Figure 5.1 has a size 12.

At last, the attribute **ndims** tells if the table is a dataframe by checking its dimension:

```
df.ndims
```

A dataframe has **ndims=2** and a series has **ndims=1**.

5.2 Dataframe index and column names

Besides by observing of the table directly, we can use the dataframe attribute **index** to call all the index in the dataframe:

```
df.index
```

Similarly we use the attribute **columns** to call all the column names used in the dataframe, we type

```
df.columns
```

```
df.index
```

```
Index(['Shop 1', 'Shop 2', 'Shop 3',
'Shop 2'], dtype='object')
```

```
df.columns
```

```
Index(['Shop', 'Apple', 'Orange',
'Kiwi'], dtype='object')
```

Figure 5.2: Querying the indices and columns of a dataframe

Notice that both attributes return a Pandas series, which is also an iterable. We can loop over these series to access the rows or columns one by one for specific operations. In Figure 5.2 we show the results of the above two commands.

5.3 Dataframe datatype

A good practice for using Pandas is that data in one single column uses a unified datatype. This makes sure

```
df.dtypes
```

```
Shop      object
Apple      int64
Orange     int64
Kiwi       int64
dtype: object
```

Figure 5.3: Querying the data types of a dataframe

that operations by column will have a lower chance of Type Error. In particular, it can be difficult to screen out invalid entries in a large dataset. Knowing the global data types will help us distinguish if some entries require data preparation. We use

```
df.dtypes
```

to check the data type of each column. The sample result for the table in Figure 5.1 is shown in Figure 5.3.

5.4 Dataframe sample data

When we work on a large dataframe, it is useful for us to only examine a few rows of a table to gain the first impression about the data. It will assist us to decide

```
df.sample(1)
```

	Shop	Apple	Orange	Kiwi
2	C	31	37	30

Figure 5.4: Querying a random row of a dataframe

what further processing steps are necessary. The methods **head**, **tail** and **sample** of a dataframe are useful for these tasks. Namely, we use

```
df.head(5)
```

to display the first 5 rows of the dataframe,

```
df.tail(5)
```

to display the last 5 rows of the dataframe, and

```
df.sample(5)
```

to ask Pandas to return 5 rows randomly from the dataframe. The number here can be changed according to the extent we want to examine the data. In Figure 5.4 we ask Pandas to randomly return a row from the dataframe **df**.

5.5 Unique values and counting

In looking at a table, it is useful to extract the unique values appearing in the dataframe. In Pandas we can use the method **nunique** to count the unique values in each column in a dataframe. We can use **value_counts** to find the unique values of a column and number of times they appeared. Notice that the method **value_count** applies on a specific column. If no column is given, it results in an error. In Figure 5.5 we demonstrate these features to our dataframe **df**. We see that there are 3 unique values in all columns except "Kiwi", where the 2 unique values are 29 and 30 respectively.

5.6 Describe a dataframe

The last useful command introduced in this Chapter is the **describe** method. This gives a set of statistics description of the dataframe column by column. This will help us to quickly grasp the statistical features of a dataframe. In Figure 5.6 we demonstrate this attribute to our **df** dataframe. We observe that Pandas not only report statistical quantities such as count, mean and the range of values, but also the percentile of the data. Notice that Pandas automatically ignores columns which contain non-numerical data, such as the "Shop" column in our example.

```
df.nunique()
```

```
Shop      3
Apple     3
Orange    3
Kiwi      2
dtype: int64
```

```
df["Kiwi"].value_counts()
```

```
30    2
29    1
Name: Kiwi, dtype: int64
```

Figure 5.5: Querying the unique values and their frequencies of a dataframe

5.7 Summary

In this short chapter, we have examined multiple useful attributes and methods which give an overview of the dataframe we are processing. They include:

- shape, size, ndims
- index, columns
- dtypes

```
df.describe()
```

	Apple	Orange	Kiwi
count	3.000000	3.000000	3.000000
mean	23.333333	26.000000	29.666667
std	6.806859	9.848858	0.577350
min	18.000000	18.000000	29.000000
25%	19.500000	20.500000	29.500000
50%	21.000000	23.000000	30.000000
75%	26.000000	30.000000	30.000000
max	31.000000	37.000000	30.000000

Figure 5.6: Querying the basic statistics of a dataframe

- head, tail and samples
- nunique and value_counts
- describe

Using these attributes and methods flexibly will greatly help us to analyze the data structure of the dataframe we are working on.

Table 5.1: The practice table for exercise 1

Rank	Shop	Apples	Oranges	Kiwi	Mangoes
1	A	18	19	22	30
2	B	24	14	25	13
5	C	42	34	34	13
4	D	15	36	23	54
3	E	15	14	24	13
6	F	42	14	53	36

5.8 Exercises

1. Let's try it on your own. This time let us prepare a slightly larger dataset as follows (Table 5.1). And try to use the above methods and attributes to analyze the dataframe. (And not by observation of naked eyes!)

    - Build the dataframe using the data in Table 1
    - How many rows and columns are there?
    - Display the first 3 rows in the dataframe.
    - Write a for loop and display the column names one by one.
    - What are the average number of apples and kiwi?
    - What is the maximum number of oranges and minimum number of mangoes?

- What is the 75% percentile of Kiwi?
- Which column has the most unique values?
- What are the unique values found in the column "Apple"?

Chapter 6

Query and Column operations (I)

In previous chapters we have learnt the basic of building, calling, modifying and describing the dataframe. They can serve as the fundamental manipulation of a table. In this chapter we shall explore the essence of Pandas, including data query and column operations. With these operations, we may handle a large dataset by imposing certain modification rules without going through the data one by one.

Again, we will use our old friend, Table 2.2 in Chapter 2 as our example (Figure 6.1).

```
In [89]: print(df)
```

	Shop	Apple	Orange	Kiwi
Shop 1	A	18	23	30
Shop 2	B	21	18	29
Shop 3	C	31	37	30

Figure 6.1: The reference table we will modify in this chapter

Query the subset of a table

To obtain a subset of the dataframe, we provide the criterion to the dataframe as its index. The method is similar to calling the data by column names. For example: the command

```
df[df["Orange"] < 30]
```

corresponds to the dataframe subset where the value in the "Orange" column is less than 30. This will give us a dataframe of only 2 rows as shown in Figure 2.

Notice that if we type only

```
df["Orange"] < 30
```

this corresponds to a series of boolean of whether the "Orange" column in each row is less than 30 or not. It does not return a subset of dataframe.

```
df[df["Orange"] < 30]
```

	Shop	Apple	Orange	Kiwi
Shop 1	A	18	23	30
Shop 2	B	21	18	29

Figure 6.2: Filtering a dataframe with a criterion

Multiple query can be used by combining criteria with "&" (and) and "|" (or). For example,

```
df[(df["Orange"] < 30) & (df["Apple"] > 20)]
```

will give us the dataframe consisting of only Shop 1 because this is the only row which contains the "Orange" column with a value less than 30 and the "Apple" column with a value greater than 20. Notice that we always need a bracket to enclose each of the argument. In Figure 6.3, we show how this operation filters the dataframe in Figure 6.1.

In some cases, we only want to select some specific values instead of using inequalities, in that case we need to use the method **isin** of the dataframe. The method **isin** takes an iterable (e.g., list) to check and return the subset of the dataframe whose values appear in the it-

```
df[(df["Orange"] < 30) & (df["Apple"] > 20)]
```

	Shop	Apple	Orange	Kiwi
Shop 2	B	21	18	29

Figure 6.3: Filtering a dataframe with multiple criteria

```
df[df["Apple"].isin([18,21])]
```

	Shop	Apple	Orange	Kiwi
Shop 1	A	18	23	30
Shop 2	B	21	18	29

Figure 6.4: Filtering a dataframe with selected values

erable. For example, we may select the row where the "Apple" column has a value 18 or 21, then we can write:

```
df[df["Apple"].isin([18,21])]
```

In Figure 6.4, we show how this method can be used to a dataframe of Figure 6.1.

Basic column operations

After learning how to select the part of the dataframe we find interesting, then we can begin our operations. Pandas allows direct arithmetic operation on the data in the same column such as :

```
df["Orange"] += 1
```

We may observe in Figure 5 that all the values in Column "Orange" have increased by 1. Others, such as "-=", "*=", "/=" are also valid operations to the whole array. If we want to use the method approach, we can use the methods **add**, **sub**, **mul** and **div** to a dataframe. The above command is equivalent to writing

```
df = df["Orange"].add(1)
```

But notice that these methods return a new dataframe instead of modifying the original dataframe.

When we need to change more than one column, instead of a scalar, we can pass a list so that the operation depends on the elements in the list. Let us try to modify the dataframe of Figure 6.1 by the following command (In case you have modified the original dataframe, rerun the dataframe declaration!):

```
df[["Apple", "Orange"]] += [2,3]
```

The results are shown in Figure 6.6 for illustration.

```
df["Orange"] += 1
df
```

	Shop	Apple	Orange	Kiwi
Shop 1	A	18	24	30
Shop 2	B	21	19	29
Shop 3	C	31	38	30

Figure 6.5: Modifying a dataframe in the same column

```
df[["Apple", "Orange"]] += [2,3]
df
```

	Shop	Apple	Orange	Kiwi
Shop 1	A	20	26	30
Shop 2	B	23	21	29
Shop 3	C	33	40	30

Figure 6.6: Modifying a dataframe in the same column by multiple values

Again, pay attention that the shape of the dataframe (i.e., the number of columns) should match the size of the list for consistency.

Similar operations can be done on data in a row. However, this method is seldom used in general. Along the same row, there can be columns which contain strings. In that case, Python will raise an error when we attempt to do an algebraic operation on these columns. In that case, we need to use the **loc** method to select columns containing only numbers and apply operations on these columns, such as:

```
df.loc["Shop 1", "Apple":"Orange"] += 1
```

and

```
df.loc["Shop 1", "Apple":"Orange"] = df.loc[
    "Shop 1", "Apple":"Orange"].add(1)
```

will provide the same dataframe result in Figure 6.7 using the dataframe in Figure 6.1. Again, it is possible to pass a list so that each column is operated with multiple values.

Multiple-column operations

In most of the data analysis, we need to combine data from multiple columns to create a new column which contains the digested data. The procedure is similar to how we create a new column, but with the data computed from the current columns. For example, we can write

```
In [37]: df.loc["Shop 1", "Apple":"Orange"] += 1
         df
```

Out[37]:

	Shop	Apple	Orange	Kiwi
Shop 1	A	19	24	30
Shop 2	B	21	18	29
Shop 3	C	31	37	30

Figure 6.7: Modifying a dataframe in the same row

```
df["Sum"] = df["Apple"] + df["Orange"] + df["Kiwi"]
```

to create a new column called "Sum" and its value is the sum of three columns, "Apple", "Orange" and "Kiwi". In Figure 6.8, we do this operation on the Figure 6.1 dataframe.

Notice that, in Pandas there are a number of built-in methods which can speed up simple statistical method like this. They include **sum**, **std**, **min**, **max**, **median**, **mode** and so on. For example, the above example can be replaced by the operation:

```
df["Sum"] = df[["Apple", "Orange", "Kiwi"]]. \
    sum(axis=1)
```

This command means that we extract the columns "Apple", "Orange" and "Kiwi", compute their sum and

```
In [55]: df["Sum"] = df["Apple"] + df["Orange"] + df["Kiwi"]
         df
```

Out[55]:

	Shop	Apple	Orange	Kiwi	Sum
Shop 1	A	18	23	30	71
Shop 2	B	21	18	29	68
Shop 3	C	31	37	30	98

Figure 6.8: Creating a new column by operating on multiple columns

form a series. The series has a length equals to the number of rows in the original dataframe. Then we attach this series in the Column "Sum" to the original dataframe. We need to define **axis=1** to specify Pandas that we are calculating the sum along the same row, not along the same column. The **axis** setting applies to all these statistical methods.

6.1 Transforming data by where and mask

When we need to modify a partial part of the row, we may need to extract the data, modify it and then replace the data. In that case, we need to use **mask** and **here**. Both functions have very similar functions expect

```
df["Apple"].mask(df["Apple"] < 25,
    20, inplace=True)
df
```

	Shop	Apple	Orange	Kiwi
Shop 1	A	20	23	30
Shop 2	B	20	18	29
Shop 3	C	31	37	30

Figure 6.9: Extract rows and replace rows with selected data

that **mask** acts on rows where the condition is true while **where** does the opposite. A script below can be used to extract the Row where the Column "Apple" has a value less than 25 (Shop 1 and 2 satisfy this criterion). Then Pandas replaces the original value by the value 20. The function **inplace** makes sure that the operation modifies the original data.

Another indirect way to change the row data can be

done by

- extracting relevant rows by filtering as shown at the beginning of the chapter,
- modify the data accordingly,
- overwrite the original data with the modified data.

A major shortcoming in using **mask** is that the value to replace the original data cannot be a variable. This largely limits the application of this method to a general dataframe.

6.2 Transforming data by apply

In fact, the method **a**pply in Pandas provide a powerful tool to connect a dataframe to more sophisticated functions or methods for advanced processing. To use the **apply** method we need to define a named method or a lambda method so that we pass this method to the method **apply** as an argument, and the method **apply** uses this function to operate on data in the dataframe.

Let us define a method **add_one** which, suggested by its name, adds one to the input argument and returns it. We use **apply** to pass this method to the dataframe and create a new column called "Apple2".

```
def add_one(value):
```

```
In [75]: def add_one(value):
             return value + 1

         df["Apple2"] = df["Apple"].apply(add_one)
         df
```

Out[75]:

	Shop	Apple	Orange	Kiwi	Apple2
Shop 1	A	18	23	30	19
Shop 2	B	21	18	29	22
Shop 3	C	31	37	30	32

Figure 6.10: Extract rows and replace rows with selected data

```
    return value+1df

["Apple2"] = df["Apple"].apply(add_one)
```

In Figure 6.10 we demonstrate how we can combine the **apply** method with our self-defined **add_one** method.

When a function needs more than one parameter as argument, we need to bypass the limitation of the method **apply** as Pandas does not pass individual columns separately. We need to use a lambda function to call this multi-argument method, and extract the individual values one by one. For example, if we want to create a new column defined by the operation

```
df["New sum"] = df["Apple"] + 2 \times df["Orange"] +
    3 \timex df["Kiwi"]
```

In fact the above line is appropriate to create a new column by the above formula. For pedagogical purpose, let us demonstrate the **apply** method explicitly. We need to construct the script as follows:

```
def new_sum(value0, value1, value2):
    return value0 + 2 * value1 + 3 * value2
    df["New sum"] = df.apply(lambda row:
        new_sum(row["Apple"],
        row["Orange"], row["Kiwi"]))
```

We do not need to modify the function being passed. Instead we need to use a lambda function to call this function before using the method **apply** because we want to extract values from the required columns. We show in Figure 6.11 how this method act on our reference dataframe shown in Figure 6.1.

6.3 Summary

In this chapter we have done an extensive overview in some useful methods to extract data and modify them. We have studied how to make a query to extract rows which satisfy the given conditions. We have also discussed how to modify data in an entire row or column.

```
def new_sum(value0, value1, value2):
    return value0 + 2 * value1 + 3 * value2

df["New sum"] = df.apply(lambda row: new_sum(
   row["Apple"], row["Orange"], row["Kiwi"]), axis=1)
df
```

	Shop	Apple	Orange	Kiwi	New sum
Shop 1	A	18	23	30	154
Shop 2	B	21	18	29	144
Shop 3	C	31	37	30	195

Figure 6.11: Extract rows and overwrite with multi-argument functions

For partial column modification of the data, we need to use

- mask: apply to rows which satisfy the condition
- where: apply to rows which do not satisfy the condition

We also demonstrated how to use the **apply** method to modify algebraic data.

6.4 Exercises

1. Consider the dataframe in Figure **??**, which shows some fictional data of bank users, write the query to extract the data with the following conditions:

   - Bank users with an age below 20
   - Bank users with an age above 20 and with cash more than 50
   - Bank users with cash or saving more than 50

   (Do you remember how to check how many users satisfy these conditions?)

2. Do the following modification by row or column:

   - Add all the age of the users by 1
   - Add cash by 10 for users whose age is below 20 (Try use both **mask** and **where** methods)
   - Add Dorothy and Felix's saving by 10

3. Then let us do more operation on the column data

   - Use **apply** method to generate a new column called *total* which is the sum of cash and saving
   - Find how many users have a total above 100

	ID	Name	Age	Cash	Saving
0	A001	Ann	13	30	40
1	A002	Brad	15	50	60
2	A003	Carla	27	40	70
3	A004	Dorothy	20	50	80
4	A005	Eaton	21	70	100
5	A006	Felix	24	90	20
6	A007	Gigi	28	100	30
7	A008	Henry	30	10	40

Figure 6.12: Dataframe for Exercises 1 to 3

- Assume that at the end of the year, the bank pays an interest rate according to their age by the following formula

  ```
  rate = (age - 10) # in percent
  ```

  Use this rate formula to calculate how much interest received by all users, and then update the saving accordingly.

Chapter 7

Query and Column operations (II)

In the previous chapter we have introduced a number of method which can apply on columns for an efficient modification of a dataframe. We have focused on numerical data only. In this chapter we will extend our discussion on textual data. In particular, we will focus on how to select, filter and modify **String** data collectively.

In this chapter we shall use a new reference dataframe which contains more textual information. We show this dataframe in Figure 7.1. The dataframe contains fiducial information about some staff and their location, email and programming language.

	Name	ID	Location	Email	Language
0	Abel	A01	UK	abel@gmail.com	Python
1	Belinda	B01	AU	belinda@yahoo.com	C
2	Carla	A02	UK	carla@yahoo.com	C
3	Don	A03	US	don@hotmail.com	Python
4	Enson	B02	CA	enson@gmail.com	C++
5	Felipe	B03	NZ	felipe@yahoo.com	Java
6	Georg	A04	CA	georg@hotmail.com	C++
7	Humprey	B04	US	humprey@yahoo.com	Python

Figure 7.1: Reference dataframe for chapter 7

7.1 Sorting

For a large dataframe, we often encounter needs to rearrange the table accordingly to the alphabetic order for some of the columns. The **sort_values** method in Pandas provides similar functionality for textual data and it is also a good way to group the data. For example, if we want to rearrange the reference dataframe according to their programming language, we write

```
df.sort_values(by="Language", inplace=True)
```

```
df2 = df.sort_values(by=["Language"])
df2
```

	Name	ID	Location	Email	Language
1	Belinda	B01	AU	belinda@yahoo.com	C
2	Carla	A02	UK	carla@yahoo.com	C
4	Enson	B02	CA	enson@gmail.com	C++
6	Georg	A04	CA	georg@hotmail.com	C++
5	Felipe	B03	NZ	felipe@yahoo.com	Java
0	Abel	A01	UK	abel@gmail.com	Python
3	Don	A03	US	don@hotmail.com	Python
7	Humprey	B04	US	humprey@yahoo.com	Python

Figure 7.2: Sorting a dataframe according to its column

The **inplace=True** guarantees that the sorted table data overwrites the original one. Some of the options in this method, such as **ascending** and **key**, have their correspondence to Python standard method **sorted** and are very useful in many applications.

However, notice that once the table is sorted and overwrites the original one, the corresponding index will be swapped. For example, in Figure 7.2, the **iloc=0** no longer corresponds to the entry of *Abel*, but the entry of

```
In [11]: df2.iloc[0]

Out[11]: Name                  Belinda
         ID                        B01
         Location                   AU
         Email       belinda@yahoo.com
         Language                    C
         Name: 1, dtype: object
```

Figure 7.3: A sorted dataframe has different index orders

Belinda (See Figure 7.3). That means, calling the entry by the **loc** method will make sure that the results do not depend on whether it is sorted or not.

7.2 Data Query

The method we introduced in Chapter 6 how query works. This can be done on a dataframe of textual data too. Certainly, some of the operation, such as ">" (greater than) or "<" (less than), will have ambiguous meaning for textual data (but they still work). We will focus on the **isin** method. For example, we can query the members whose locations are "UK" and "AU" by the following script:

```
df[df["Location"].isin("AU","UK")]
```

```
df[df["Location"].isin(["AU","UK"])]
```

	Name	ID	Location	Email	Language
0	Abel	A01	UK	abel@gmail.com	Python
1	Belinda	B01	AU	belinda@yahoo.com	C
2	Carla	A02	UK	carla@yahoo.com	C

Figure 7.4: Getting entries with selected criteria

The query result is shown in Figure 7.4 for our reference dataframe.

Sometimes we want to collect entries where part of the string fits the pattern that we are searching. For example, in our reference dataframe, we want to collect entries where the "email" has a domain "yahoo.com". To do so, we need to make use of the built-in string operation methods available in Pandas. In Pandas, many of the Python native methods for string are available under the **str** attribute (e.g., **upper**, **lower**, **len** and so on). For example, we can query the length of the email column by

```
df["Email"].str.len()
```

In Figure 7.5 we apply this query to our reference dataframe.

```
In [16]: df["Email"].str.len()

Out[16]: 0    14
         1    17
         2    15
         3    15
         4    15
         5    16
         6    17
         7    17
         Name: Email, dtype: int64
```

Figure 7.5: Using built in functions for string in Pandas

Notice that we cannot use these built-in methods directly on a column. For example, the following script has a complete context from the above one:

```
len(df["Email"])
```

which only asks how many items are there in the Column *email*, which is 8. The **str** attribute of the dataframe is vital if we want to use these methods to the strings concerned.

Similarly, we can use these String methods for our query. For example, we can ask for entries whose location starts with the letter "U" by the method **startswith** (also see **endswith**):

```
df[df["Location"].str.startswith("U")]
```

	Name	ID	Location	Email	Language
0	Abel	A01	UK	abel@gmail.com	Python
2	Carla	A02	UK	carla@yahoo.com	C
3	Don	A03	US	don@hotmail.com	Python
7	Humprey	B04	US	humprey@yahoo.com	Python

Figure 7.6: Querying string with selected features

```
df[df["Location"].str.startswith("U")]
```

and the result for our reference dataframe is shown in Figure 7.6.

To extract a part of the String and see if we this part matches the criteria we specific, we use **contains** (for regex expression). The names of these two methods are explicit and need no further description. For example, we can check if any user in the reference dataframe uses an email with a domain "yahoo.com" by

```
df[df["Email"].str.contains("yahoo.com")]
```

The result is shown in Figure 7.7.

We can also use regex to carry out our search. The script equivalent to above is given by:

```
df[df["Email"].str.contains("yahoo.com")]
```

	Name	ID	Location	Email	Language
1	Belinda	B01	AU	belinda@yahoo.com	C
2	Carla	A02	UK	carla@yahoo.com	C
5	Felipe	B03	NZ	felipe@yahoo.com	Java
7	Humprey	B04	US	humprey@yahoo.com	Python

Figure 7.7: Check if part of the string satisfies the criteria

```
df[df["Email"].str.match(r"[A-Za-z]*@yahoo.com")]
```

In Figure 7.8 we show that the query provides the same output for the reference dataframe.

7.3 Data modification

Due to the inherent methods for string in Pandas, we can choose to process the column with string by the **apply** method discussed in last chapter, or by the standard string methods in native Python. Here we will briefly discuss how both approaches take place for modification in general. Some of the standard methods include:

- upper

```
df[df["Email"].str.match(r"[A-Za-z]*@yahoo.com")]
```

	Name	ID	Location	Email	Language
1	Belinda	B01	AU	belinda@yahoo.com	C
2	Carla	A02	UK	carla@yahoo.com	C
5	Felipe	B03	NZ	felipe@yahoo.com	Java
7	Humprey	B04	US	humprey@yahoo.com	Python

Figure 7.8: Check if part of the string satisfies the criteria by regular expression

- lower
- strip
- partition
- replace

Among all these methods, the method **replace** is a versatile method which is worth to be further discussed. Let us consider our reference dataframe and assume that we want to replace the suffix in all emails from ".com" to ".net". To do so, we type

```
df["Email"] = df["Email"].str.replace(r"com","net")
```

```
df["Email"] = df["Email"].str.replace(r"com","net")
df
```

	Name	ID	Location	Email	Language
0	Abel	A01	UK	abel@gmail.net	Python
1	Belinda	B01	AU	belinda@yahoo.net	C
2	Carla	A02	UK	carla@yahoo.net	C
3	Don	A03	US	don@hotmail.net	Python
4	Enson	B02	CA	enson@gmail.net	C++
5	Felipe	B03	NZ	felipe@yahoo.net	Java
6	Georg	A04	CA	georg@hotmail.net	C++
7	Humprey	B04	US	humprey@yahoo.net	Python

Figure 7.9: Replacing string by standard string method

In Figure 7.9 we show the final result. We remind that we need to assign the modified column to some variable because this method does not change the original dataframe **inplace**. The change will not be kept if the output is not stored somewhere.

An identical dataframe can be achieved by using:

```
import re
def change_com_to_net(row):
    new_email = re.sub("com", "net", row["Email"])
```

Table 7.1: Sample input and output for the fictitious operation

input	output
UK, Python	uk-pyt
CA, C++	ca-c++

```
    return new_email

df["Email"] = df.apply(lambda x:
    change_com_to_net(x), axis=1)
```

To understand these lines of code, we first define a short method **change_com_to_net** which will identify any *com* in the string, and then replace by *net*. This will act on the whole dataframe row by row. In each row, Pandas calls the method to return a string. Then, Pandas use the string to overwrite the original Email column.

Another example we want to demonstrate here is this: Assume we want to create a new column called "Branch". This column stores an abbreviation which is the combination of the column Location and the column Language. We will not use the whole word available in the column *Language*. Instead, we will only use the three characters in the string. Also, all the letter should be in small letter. In Table 7.1 we tabulate some example input and output.

To do so, we need to combine standard string opera-

tion and the method **apply**. It is because the standard string operation can act only on one column. However, this time we need to combine results of two columns.

```
def get_first_3char(row):
    return row[:3]

prefix = df["Location"].str.lower()
suffix = df["Language"].str.lower().\
    apply(get_first_3char)
df["Branch"] = prefix + "-" + suffix
```

In Figure 7.10 we show the result of our operations. We remind that for the entries where the Language is "C", there is only one character in the suffix. The above code is self-understanding from its notation. First, we set up a short method which returns the first 3 characters from a given string. Then we prepare the **prefix** and **suffix** accordingly by taking the string and transform into lower alphabets. For the suffix we further use the defined method **get_first_3char** to extract the first three characters. Then we combine the two columns like strings.

Notice that to obtain the above results, we cannot write:

```
df["Language"][:3]
```

```
def get_first_3char(row):
    return row[:3]

prefix = df["Location"].str.lower()
suffix = df["Language"].str.lower().apply(get_first_3char)
df["Branch"] = prefix + "-" + suffix

df
```

	Name	ID	Location	Email	Language	Branch
0	Abel	A01	UK	abel@gmail.net	Python	uk-pyt
1	Belinda	B01	AU	belinda@yahoo.net	C	au-c
2	Carla	A02	UK	carla@yahoo.net	C	uk-c
3	Don	A03	US	don@hotmail.net	Python	us-pyt
4	Enson	B02	CA	enson@gmail.net	C++	ca-c++
5	Felipe	B03	NZ	felipe@yahoo.net	Java	nz-jav
6	Georg	A04	CA	georg@hotmail.net	C++	ca-c++
7	Humprey	B04	US	humprey@yahoo.net	Python	us-pyt

Figure 7.10: Combined operations on strings

It is because this corresponds to getting the column data *Language*, and then select the first three elements, which is a series ["Python", "C", "C"].

7.4 Summary

In this chapter we focus on another major part of Pandas, the operation of column data with textual data. We have shown that by combining the method **apply** for operation, together with default string operation methods, Pandas allows us to create a lot of manipulate flexibly the string for many different uses.

We have also examined how we can create a query and sort the dataframe when textual data is involved. Together with the manipulation, it completes our discussion in how to globally modify the dataframe for any type of data (numerical and lexical). These skills form the basis for future data processing which will be covered elsewhere.

7.5 Exercises

Now let us practice the string methods covered in this chapter to strengthen our understanding. Let us consider the same dataframe in Figure 7.1.

Your friend wants to process the dataframe for the task. But your friend again does not know what to do. Can you teach your friend how to finish these tasks?

1. Do a sorting so that the final dataframe is sorted according to *Location* and then *Language*

2. What are the unique elements in the column *Language*?

3. Write a query to collect members whose *ID* starts from "A". How many members of them are C users?

4. Write another query to collect members whose *ID* ends with "03". Can you extract a simple statistics the location of these members? (For sure we do not want to rely on our naked eye inspection!)

5. The company starts a new campaign for unifying all the email of the members. In this campaign, all member's new email become "Name" and then their *Location*, linked by a hyphen, with a domain "@company.com". All alphabets are in lower letter. Can you write a short program and store the new email in a new column *New Email*? For reference, Table 7.2 show some sample new emails.

Table 7.2: New entries for Exercise 1

Name	Location	New Email
Abel	UK	abel-uk@company.com
Humprey	US	humprey-us@company.com

Chapter 8

Indexing and Dataframe Merging

In the last two chapters, we have discussed in details how we can operate numerical and textual data in the column level. There are a number of methods involved but they are essential to most data processing routines for preparing a clean set of data for the future data analysis. It is therefore encouraged that the material in previous chapters are well mastered before we proceed to the more application side of Pandas.

In this chapter, we will discuss the importance of index in Pandas dataframe and how we merge multiple dataframe into one for our data processing. The index in Pandas is the key feature for many purposes. One of which is for the grouping of data (aggregation) for the

statistical analysis. Another purpose is that key can be used for matching multiple dataframe. Hence, knowing how to set the index and its hierarchy will provide the convenience in making multi-layer analysis of a dataframe within a few lines of code.

8.1 Indexing

In Chapter 2 we have discussed how to prepare a dataframe index by passing the list which contains all the keys. We can also request Pandas to set a certain column to be the list of index. In fact, in Pandas, besides the method **rename**, we can also collectively change the index. In general, renaming index can be useful when the original index is does not fully represet the data, for example the default integers. In that case we want to use tags which can provide direct insight to the data so that we can quickly grasp the overall content of the dataframe.

To illustrate this idea, let us consider the following dataframe (Figure 8.1). The dataframe contains some randomly selected cities, and their corresponding states and countries. The index is named as "City1", "City2", and so on. When we want to call the data of specific city, let's say "Texas", we need to type

```
df2.loc["City3"]
```

which leads to the results shown in Figure 2.

```
df2
```

Out[69]:

	Country	State	City
City1	USA	CA	SF
City2	USA	CA	LA
City3	USA	TX	Texas
City4	Canada	Ontario	Toronto
City5	Canada	Quebec	Montreal

Figure 8.1: Sample geographic data

```
In [70]: df2.loc["City3"]

Out[70]: Country      USA
         State         TX
         City       Texas
         Name: City3, dtype: object
```

Figure 8.2: Fetching Data by abstract keys

—

Indeed, such an approach is correct in syntax but is inconvenient in real data search. In particular, we observe that in the original dataframe, the data share com-

mon countries or states. It will be beneficial to make use these common features. For example, we can remove the original index and use "Country" as the index. In Figure 8.3 we demonstrate this by the method:

```
df2.set_index("Country", inplace=True)
```

Notice that this method changes the order of the dataframe. Therefore, Pandas in general generates a new dataframe to store the result unless the function **inplace=True** is chosen. Also, we remind that this method removes the original index. In the case where the original index also contains useful information, we need to set

```
df2.set_index("Country", drop=False,
    inplace=True)
```

such that the index column is returned to the original dataframe.

If we want to call the elements by **loc**, we use the new index directly. For example, in Figure 8.4 we show the result of calling the "USA" index by **loc**. There are multiple results are returned because, as there are multiple entries sharing the same Country value. All the entries are now the cities which are in USA. This method is good for quickly categorizing data within the same group.

The decision of using a certain column for the index needs not to be fixed throughout the analysis. It always

```
In [71]: df2.set_index("Country", inplace=True)
         df2
```

Out[71]:

Country	State	City
USA	CA	SF
USA	CA	LA
USA	TX	Texas
Canada	Ontario	Toronto
Canada	Quebec	Montreal

Figure 8.3: Setting "Country" as the index

depends on the task we are working on. For example, if we focus on the country as the basis, choosing "Country" as the index will allow us to access the global features of data conveniently. On the other hand, if we need to focus on features which are city-based, choosing the column "City" as the index will be a more natural choice.

When we finish the use of a certain column as the index, we can reset the index so that the column can be "unlocked" and returns to the dataframe. We can use the **reset_index** method as shown in Figure 8.5. Again, the operation changes the original dataframe. We need

```
In [72]: df2.loc["USA"]
```

Out[72]:

Country	State	City
USA	CA	SF
USA	CA	LA
USA	TX	Texas

Figure 8.4: Calling new index in the dataframe

to select **inplace=True** or **False** to decide whether we want to replace the original dataframe.

It is also possible to choose multiple columns to be the indices. This forms a hierarchical structure in how each row is stored. To do so, we need to pass the columns in a list for the **set_index** method. For example, in Figure 8.6 we demonstrate how to use country and then the state as the indices. Pandas will first group all the data based on the first index "Country", and then make further sub-groups within each group based on the second index "State". The example here naturally reflect this hierarchy due to their geographical location. In the first index "USA", we have further sub-groups "CA" and "TX", in each sub-group there are elements "SF", "LA"

```
In [73]: df2.reset_index(inplace=True)
         df2
```

Out[73]:

	Country	State	City
0	USA	CA	SF
1	USA	CA	LA
2	USA	TX	Texas
3	Canada	Ontario	Toronto
4	Canada	Quebec	Montreal

Figure 8.5: Reset the index in the dataframe

and so on.

When multiple indices are used and when we need to call the rows, we can pass single index or full index depending on which grouping we need to fetch. For example, by calling

```
df2.loc["USA"]
```

we can access the groups of rows where the first index is "USA", which will return us 3 entries. On the other hand, to access the sub-level, we need to pass the index names in a **tuple**. In Figure 8.7 we demonstrate how this can be done. In that example, we try to call the elements

```
In [74]: df2.set_index(["Country","State"], inplace=True)
         df2
```

Out[74]:

		City
Country	**State**	
USA	**CA**	SF
	CA	LA
	TX	Texas
Canada	**Ontario**	Toronto
	Quebec	Montreal

Figure 8.6: Setting multiple indices in a dataframe

where the primary key is "USA" and the secondary "key" is "TX", in the dataframe there is one entry satisfying these conditions and it returns us the city "Texas". In this example, we also use the command **sort_index** because the grouping needs not to follow alphabetical order in the first place. If the index is not sorted, Pandas will warn about the performance issue.

Finally, to check the structure of the indices, we use the same attribute **index** as described in Chapter 5. Pandas will return the tuple of indices in a structured manner as shown in Figure 8.8.

As a general rule of thumb, it is good to set the key where the data in the key does not participate in the sta-

```
In [77]: df2.sort_index(inplace=True)
         df2.loc[("USA","TX")]
```

Out[77]:

		City
Country	**State**	
USA	**TX**	Texas

Figure 8.7: Calling by multiple indices

```
df2.index
```

```
MultiIndex([('Canada', 'Ontario'),
            ('Canada',  'Quebec'),
            (   'USA',      'CA'),
            (   'USA',      'CA'),
            (   'USA',      'TX')],
           names=['Country', 'State'])
```

Figure 8.8: Checking the indices of the dataframe

tistical analysis or merging. It is because once the column is set as the index, methods such as **value_counts** or **sum** will not act on that column. It is also a good practice to use indices which are unique for elements. This provides the flexibility of calling individual rows without having to specific their row and column numbers explicitly.

8.2 Dataframe Merging

With the index structure explained above, now we can proceed to talk about methods in merging multiple dataframe. Pandas provides the method **merge** which allows us to combine multiple table based on specific keys we want Pandas to match among tables. We first define another table which we will merge in Figure 8.9. It is a simple dataframe which contains fiducial number of persons living in these cities. We remind that we intended to slightly rearrange the order of the row to show that Pandas is matching the row in two dataframes by the specific data in the selected column.

In order to merge the table, we use the Pandas method **merge**:

```
df_output = pd.merge(df_left, right=df_right, how=...,
    left_on=..., right_on=...)
```

```
df3
```

Out[99]:

	City	Populaton
0	LA	100
1	SF	300
2	Texas	200
3	Toronto	400
4	Montreal	500

Figure 8.9: Second dataframe for merging

The two dataframes for merging are compulsory arguments. A number of useful options we need to observe are:

- how: it takes "inner", "outer", "left" and "right" which defines whether or not taking entries which do not exist in either dataframe. We illustrate their relation in Figure 8.10.
- **left_on**: it takes the column names which exists in **df_left** which will be used by Pandas for the merging process
- **right_on**: it takes the column names which exists in **df_right** which will be used by Pandas for the

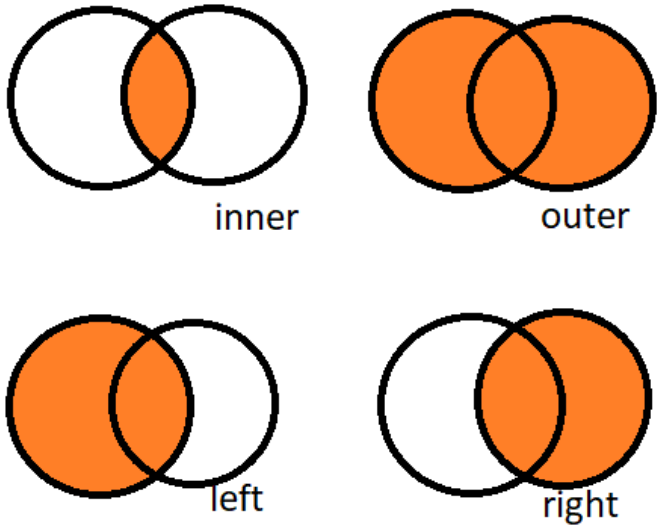


Figure 8.10: Schematic Venn Diagram on the **how** component in method **merge**

merging process

When the two dataframe shares the same column data, we can merge the dataframe directly by just calling the **merge** method and pass the two dataframe in it. In our example here, the column *City* exists in both dataframe. Therefore, Pandas will automatically merge

```
In [94]: df4 = pd.merge(df2, right=df3)
         df4
```

Out[94]:

	Country	State	City	Populaton
0	USA	CA	SF	300
1	USA	CA	LA	100
2	USA	TX	Texas	200
3	Canada	Ontario	Toronto	400
4	Canada	Quebec	Montreal	500

Figure 8.11: Merging the dataframe on a one-to-one level

the two dataframe row by row by matching the value in the column *City*. We show the results in Figure 8.11.

It is possible to merge the dataframe where there is not a one-to-one mapping of the row entries. In Figure 8.12 we show another small dataframe where the *State* data is stored as the common column. We want to demonstrate how Pandas merge when there is a many-to-one relation between the two dataframe. We show the merged dataframe in Figure 8.13. We observe that rows with the same *State* share the same entry from the dataframe **df5**.

It is possible that the two dataframe do not share the

```
df5
```

Out[95]:

	State	Weather
0	CA	Sunny
1	TX	Cloudy
2	Ontario	Rainy
3	Quebec	Snow

Figure 8.12: Another dataframe before merging

```
In [96]: df6 = pd.merge(df2, right=df5)
         df6
```

Out[96]:

	Country	State	City	Weather
0	USA	CA	SF	Sunny
1	USA	CA	LA	Sunny
2	USA	TX	Texas	Cloudy
3	Canada	Ontario	Toronto	Rainy
4	Canada	Quebec	Montreal	Snow

Figure 8.13: Merged dataframe for many-to-one mapping

```
df7
```

Out[97]:

	State Name	Weather
0	CA	Sunny
1	TX	Cloudy
2	Ontario	Rainy
3	Quebec	Snow

Figure 8.14: Another dataframe before merging

same column header. In that case, we may either rename the column header in one of the dataframe first, or we can make use of the **left_on** and **right_on** arguments to help Pandas to recognize which columns serve as the key. If no columns are common during merging, Pandas will raise an error. In Figure 8.14, we slightly change the dataframe so that the column is now called *State Name*. To do the same merging, in Figure 8.15, we indicate the left dataframe (**df2**) to use the column *State*, while the right dataframe **df8** to use the column *State Name*. The final results are similar to that in Figure 8.13. However, notice that Pandas preserves the matching columns. We might need to remove by using the **drop** method for the repeating columns.

```
df8 = pd.merge(df2,
    right=df7,
    left_on="State",
    right_on="State Name")

df8
```

	Country	State	City	State Name	Weather
0	USA	CA	SF	CA	Sunny
1	USA	CA	LA	CA	Sunny
2	USA	TX	Texas	TX	Cloudy
3	Canada	Ontario	Toronto	Ontario	Rainy
4	Canada	Quebec	Montreal	Quebec	Snow

Figure 8.15: Merged dataframe with no common columns

8.3 Summary

When you read up to here, congratulations! This chapter is indeed long and filled with a lot of technical details in how to prepare a clean dataframe for managing and for merging. We have explored how to use the index for classifying data according to their category. We have also discussed in details how we can merge two dataframe for creating a composite dataframe. Including this chapter, the first 8 chapters form the fundamental of using Pandas for handling tabular form of data. I hope the readers may now appreciate the very powerful use of Pandas for the efficient treatment of data with only a few lines of code.

Starting from the next chapter, we will explore more on the application of Pandas for more advanced treatment of the dataframe. Therefore, it will be very useful that readers can master all the skills we have explored in these chapters.

8.4 Exercises

1. Your friend has recently extended the dataframe as shown in Figure 8.16. Your friend wants to analyze the dataframe but not sure how, so your friend asked for your help. Can you tell which commands are needed for the following results?

```
df_ex1
```

	Name	ID	Location	Email	Language	Salary	Position	Gender
0	Abel	A01	UK	abel@gmail.com	Python	2000	Staff	M
1	Belinda	B01	AU	belinda@yahoo.com	C	3000	Staff	F
2	Carla	A02	UK	carla@yahoo.com	C	1000	Helper	F
3	Don	A03	US	don@hotmail.com	Python	1000	Staff	M
4	Enson	B02	CA	enson@gmail.com	C++	2000	Assistant	M
5	Felipe	B03	NZ	felipe@yahoo.com	Java	3000	Helper	F
6	Georg	A04	CA	georg@hotmail.com	C++	3000	Assistant	F
7	Humprey	B04	US	humprey@yahoo.com	Python	2000	Staff	M

Figure 8.16: Reference dataframe for Exercise 1

- Set up a new dataframe which looks like the Figure 8.17
- Write a query to show the number of language users, grouped by their gender
- Now try to reset the index and set *Position* to be the index
- Without using calculator or direct calculation, find the average salary of the entries of *Staff*
- What are the maximum and minimum salaries of *Staff* using *Python*? Again, try to use query and Pandas attributes or function to obtain the answer.

2. Then, your friend found two more dataframes which

```
df_ex2a
```

Out[133]:

	Language	Training
0	Python	Exercise
1	C	Bootcamp
2	C++	Revision

Figure 8.17: Target result for Exercise 1a

needs to be merged to the dataframe in Figure 8.16. Can you tell your friend how to do that? See Figure 8.18 for the provided dataframe.

- We want to keep all the rows of the original dataframe. Do you know how?
- There is one entry in the final dataframe filled with NaN. Which one is it?
- Do you know how to remove the repeating column?

```
df_ex2b
```

Out[134]:

	Country	Food
0	UK	Fish
1	US	Burger
2	CA	Maple
3	NZ	Beans
4	AU	Lobster

Figure 8.18: Target result for Exercise 1b

Chapter 9

Grouping and Aggregating Dataframe

In the previous section we have surveyed the important and fundemtnal skills for operating a dataframe. We have discussed extensively how to prepare and process a dataframe and its data. In this chapter we will study how to process the data for statistical analysis. We will first study how we can group the the data by using the method **groupby**, and then we will learn how to compute the aggregation of data by the method **agg**. These two methods are often used together to generate rapid results about the statistical features of a dataframe.

```
df_ex1
```

	Name	ID	Location	Email	Language	Salary	Position	Gender
0	Abel	A01	UK	abel@gmail.com	Python	2000	Staff	M
1	Belinda	B01	AU	belinda@yahoo.com	C	3000	Staff	F
2	Carla	A02	UK	carla@yahoo.com	C	1000	Helper	F
3	Don	A03	US	don@hotmail.com	Python	1000	Staff	M
4	Enson	B02	CA	enson@gmail.com	C++	2000	Assistant	M
5	Felipe	B03	NZ	felipe@yahoo.com	Java	3000	Helper	F
6	Georg	A04	CA	georg@hotmail.com	C++	3000	Assistant	F
7	Humprey	B04	US	humprey@yahoo.com	Python	2000	Staff	M

Figure 9.1: Reference dataframe for this Chapter

9.1 Grouping the Data

In the last chapter, we have examined how to use multiple indices to segregate the data into layers so that the data is classified according to their attribute. In fact, without using the **set_index** method, Pandas provides another method **groupby**, which also allows users to generate partitioned data based on the attributes of the entries.

For demonstration purpose let us reuse the dataframe in the exercise of Chapter 8 as shown in Figure 1.

In the last chapter we have used the coding language and the gender as the classification criteria. How do we do that using **groupby**? First, we start by grouping the entries by the gender. We need to set

```
df_grouped = df_ex1.groupby("Gender")
type(df_grouped)
```

```
pandas.core.groupby.generic.DataFrameGroupBy
```

Figure 9.2: The **groupby** method returns a groupby object

```
df_grouped = df_ex1.groupby("Gender")
```

This method does not return a dataframe object, but a **grouped_by** object (See Figure 2).

We do not directly access a groupby object like what we do for a dataframe. It is because a groupby object cannot be directly visualized unless it is transformed into a dataframe. To understand how this object classifies our **df_ex1** dataframe according to the gender, we use

```
df_grouped.groups
```

or

```
df_grouped.indices
```

to access the label of each of the subclass. In Figure 9.3 we show these attributes for our **df_ex** dataframe. In general, a group-by object acts as a dictionary where

```
df_grouped.groups
```

```
{'F': Int64Index([1, 2, 5, 6], dtype='int64'),
 'M': Int64Index([0, 3, 4, 7], dtype='int64')}
```

```
df_grouped.indices
```

```
{'F': array([1, 2, 5, 6], dtype=int64),
 'M': array([0, 3, 4, 7], dtype=int64)}
```

Figure 9.3: Accessing the classification in a groupby object

the group label is the key, while the values are the indices corresponding to the original dataframe. Similar to the **columns** or **indices** attributes of a dataframe, these two attributes return an iterable which can be used for looping.

There are some major differences between a GroupBy object compared to a dataframe object. Some general attributes such as **head**, **tail**, **nunique** do not have an exact result to a GroupBy object. In Figure **??** we show another new method **nth** which extracts the sample data from each subgroup. Most attribute provides information for each subgroup. In Figure 9.5 we show how the **size** function gives the size of each subgroup. It is therefore

```
df_grouped.nth(0)
```

Gender	Name	ID	Location	Email	Language	Salary	Position
F	Belinda	B01	AU	belinda@yahoo.com	C	3000	Staff
M	Abel	A01	UK	abel@gmail.com	Python	2000	Staff

Figure 9.4: Accessing the first n-th row entries from a GroupBy object

```
In [163]: df_grouped.size()

Out[163]: Gender
          F    4
          M    4
          dtype: int64
```

Figure 9.5: Access the size of each subgroup in a groupby object

advised for readers to go through in details which features their data analysis requires for the best use of this structure.

However, we want to emphasize that a groupby object is not exactly a dictionary as there is no correspondence

```
for index, group in df_grouped:
    print(index)
    print(group)
```

```
F
      Name   ID Location              Email Language  Salary   Position Gender
1  Belinda  B01       AU  belinda@yahoo.com        C    3000      Staff      F
2    Carla  A02       UK    carla@yahoo.com        C    1000     Helper      F
5   Felipe  B03       NZ   felipe@yahoo.com     Java    3000     Helper      F
6    Georg  A04       CA  georg@hotmail.com      C++    3000  Assistant      F
M
      Name   ID Location              Email Language  Salary   Position Gender
0     Abel  A01       UK     abel@gmail.com   Python    2000      Staff      M
3      Don  A03       US    don@hotmail.com   Python    1000      Staff      M
4    Enson  B02       CA    enson@gmail.com      C++    2000  Assistant      M
7  Humprey  B04       US  humprey@yahoo.com   Python    2000      Staff      M
```

Figure 9.6: Treating Groupby object as an iterable

to some native methods such as **items**, **keys**. In order to access all the labels and their subgroups, we treat the groupby object as an iterable itself, which gives us in each iteration a label and its corresponding dataframe:

```
for label, group in df_grouped:
```

The result is shown in Figure 9.6. The second object returned in each iterable is a dataframe object, where we can apply the standard methods we have introduced in previous chapters.

Similar to multi-layered index, we can also use multiple columns to create a layered GroupBy object for more complex classification. We can pass the columns as a list when we call the **groupby** method, such as:

```
df_ex1.groupby(["Gender", "Language"])
```

```
In [193]: df_grouped["Salary"].agg(sum)

Out[193]: Gender
          F    10000
          M     7000
          Name: Salary, dtype: int64
```

Figure 9.7: Using multiple columns for grouping

The classification result is identical to what we have done by using the **set_index** method with the sample results shown in Figure 9.7.

9.2 Aggregating data

After we have grouped the data accordingly, the next thing we may do is to extract the features in each of the subgroup. We may use the method **aggregate** (or its short form **agg**) to compute some of the mathematical features for some columns. This method take any existing mathematical methods available in Python, Numpy and Pandas, including:

- sum
- max

- min
- mean
- std

Notice that the method **agg** can be used to a GroupBy objects, as well as a standard dataframe. For the latter case, it will compute the quantities based on the whole dataframe, instead of a subset of a dataframe. Notice that in the argument we can pass **axis=0** for doing aggregation along the column and **axis=1** along the row.

We can use this method to compute the sum of all salaries in each subgroup by:

```
df_grouped["Salary"].agg(sum)
```

The result is listed in Figure 9.8. There are equivalent ways to achieve the same results by using method in other libraries such as NumPy by

```
import numpy as np
df_grouped["Salary"].agg(np.sum)
```

or by defining our own function for process the column "Salary"

```
def my_sum(column):
return sum(column)

df_grouped["Salary"].agg(my_sum)
```

```
In [193]: df_grouped["Salary"].agg(sum)

Out[193]: Gender
          F    10000
          M     7000
          Name: Salary, dtype: int64
```

Figure 9.8: Using aggregate to compute mathematical features of data

In fact, in the coding level, **aggregate** relies on the **apply** method. Therefore, method that can be passed using the **apply** can be equally processed by the method **agg**. Notice that when we pass our self-defined methods, the name does not need the quotation marks, we pass as if we pass the Python method into the **agg** method.

It is also possible to use the whole Groupby object and compute the aggregated quantities such as:

```
df_grouped.agg(sum)
```

In that case, Pandas will pick columns where the datatypes are integers or floating numbers, and return the results for these columns. In Figure 9.9 we show how it works on the **df_grouped**.

When multiple functions are passed, we only need to store the functions in a list when we call the **agg** method, such as:

```
In [196]: df_grouped.agg(sum)
```

Out[196]:

Gender	Salary
F	10000
M	7000

Figure 9.9: Passing the whole GroupBy object for aggregation

```
df_grouped.agg([sum, mean])
```

when we want to compute both summation and average of the columns with only numerical data. Finally, we can also use the **agg** function to process data with row as the entry. To do so, we need to pass the argument **axis=1** when we call **agg** method.

9.3 Summary

In this chapter, we have studied how to group the data based on features of the attributes. The use of **groupby** generates a GroupBy object which contains the classification label and their corresponding data. The groupby

object allows us to access each subgroup easily so that we can process all subgroups by iterations.

To compute statistical features along columns in a dataframe or a subset of dataframe (known as the GroupBy object), we can use **agg** to pass the necessary functions for later processing. Pandas automatically identifies columns which store only numerical data, and apply the methods one by one to each dataframe subset. This method is very efficient in process data with similar features to extract their collective properties.

9.4 Exercises

This time, let us assume another scenario where your friend becomes a survey staff in the demography bureau. Your friend brings forth to you a dataframe of the number of residents in villages and cities. In Figure 9.10 we show the dataframe from your friend.

You have already reminded your friend that it takes a lot of Pandas skill to process the survey data. However, your friend does not read the textbook "Learning Pandas from Zero" well and is now stuck at how to compute some aggregate data of these survey data. Can you help your friend out by providing the necessary script?

1. What is the maximum population achieved in the city and when did it happen?

```
df_ex1
```

	Name	Type	Year2000	Year2001	Year2002	Year2003	Year2004
0	Los Angelas	City	100000	105000	108000	110000	112000
1	San Francisco	City	230000	240000	250000	255000	290000
2	Lahan	Village	500	510	530	100	100
3	Aveh	City	20000	24000	25000	29000	30000
4	Cedric	Village	4000	4200	3500	3600	3000

Figure 9.10: Dataframe for Exercise 1 which contains the fiducial population number in different places

2. What is the total population at each years?
3. What is the mean population in the city and in the village?
4. Which place has the largest drop rate in a year? (Hint: It will need multiple steps to arrive this answer.)

Again, the table here we use is small, so it is possible to compute by hand. But don't do that unless you want to check your answer! Instead, try to rely on writing script to help you find the answer. It will give you more insight when you work with a much larger database while being confident that you are working with the data correctly.

Chapter 10

Pivot Table

In the last chapter we have a diversified study in how to use grouping and aggregating to efficiently collect entries of the same group and compute their statistical features. In fact, Pandas has another option in doing grouping, which is called the Pivot Table. Pandas identifies from the dataframe entries by specified columns, and group the entries where the entries of these columns share the same values. Then Pandas draws a pivot table by relocating the associated data based on the values of the specified columns. In this chapter, we will go through by a mini-case study to demonstrate how to build a pivot table and its applications. We will proceed step-by-step how the data analysis is taken place to extract useful measurement from the dataframe.

10.1 Mini Case-Study: Sales of a Fruit Store

First, we have a friend who sells fruit. He makes a chart to collect how many fruits are sold in his store. Without loss of generality, in Figure 10.1 we show the snapshot of the table we will work on. It is a simple dataframe which contains the date, and the corresponding sales of apples, oranges and kiwis of that day.

First we need to check the datatype of the dataframe by the **dtypes** attribute. (Do you know what it means? If no, let's revise Chapter 5 again!) We show the result in Figure 10.2. Clearly, the columns for the three types of fruits are int-64, meaning that they contain integer numbers only. The column *date* is classified as object as it is a string in general (with hyphen in this case).

However, the datatype for object is not ideal for processing date because the temporal meaning of date is not considered by Pandas. It does not care about which day it is, or the relation of that date to other dates in other rows. To make Pandas understand this column contains the temporal data, we need to convert the string into *datetime* objects by using the method:

```
pd.to_datetime(dataframe, format=...)
```

Here, dataframe is the object we want to convert, and format is the standard **strptime** format for changing a

```
df1
```

	date	apple	orange	kiwi
0	2021-05-17	14	20	20
1	2021-05-18	18	19	19
2	2021-05-19	20	12	20
3	2021-05-20	21	13	30
4	2021-05-21	24	15	36
5	2021-05-24	21	16	35
6	2021-05-25	18	17	24
7	2021-05-26	20	13	13
8	2021-05-27	26	15	25
9	2021-05-28	27	12	26

Figure 10.1: Reference dataframe of this Chapter

string into a datetime object in standard Python. In particular, we have %Y, %m and %d corresponding to the year, month and day in a calendar. In Figure 10.3 we show how we change the column "date" from its object attribute to a datetime attribute. In the format we passed "%Y-%m-%d" which matches the datetime format in the original dataframe. We can see that after

```
In [224]: df1.dtypes

Out[224]: date      object
          apple      int64
          orange     int64
          kiwi       int64
          dtype: object
```

Figure 10.2: Checking the datatype of the dataframe

the conversion, indeed the "date" column contains only datetime64 data.

After that, we want to prepare the data for the pivot table. I want to build a calendar-like pivot table. This means, each row corresponds to a week and each column corresponds to a weekday (Monday, Tuesday, ...). To do so, we need to transform the "date" column into the week number and the weekday number. The module **datetime** of Python has all the handy methods to take care these tasks. We first prepare the methods which will be passed the the method **apply** to the dataframe:

```
import datetime
def get_week(date):
    return date.isocalendar()[1]

def get_weekday(date):
```

```
In [19]: df1["date"] = pd.to_datetime(
             df1["date"],
             format="%Y-%m-%d")

         df1.dtypes

Out[19]: date        datetime64[ns]
         apple                int64
         orange               int64
         kiwi                 int64
         week                 int64
         weekday              int64
         dtype: object
```

Figure 10.3: Changing the datatype of the dataframe

```
    return date.isoweekday()
```

The meaning of the methods is obvious. The method **get_week** returns which week the date is in, while the method **get_weekday** returns the weekday the date is. In Python, Monday is represented by 1, Tuesday by 2 and so on.

Then we pass the methods to the dataframe:

```
df1["week"] = df1["date"].apply(get_week)
df1["weekday"] = df1["date"].apply(get_weekday)
```

```
df1
```

	date	apple	orange	kiwi	week	weekday
0	2021-05-17	14	20	20	20	1
1	2021-05-18	18	19	19	20	2
2	2021-05-19	20	12	20	20	3
3	2021-05-20	21	13	30	20	4
4	2021-05-21	24	15	36	20	5
5	2021-05-24	21	16	35	21	1
6	2021-05-25	18	17	24	21	2
7	2021-05-26	20	13	13	21	3
8	2021-05-27	26	15	25	21	4
9	2021-05-28	27	12	26	21	5

Figure 10.4: Adding columns to the dataframe

Notice that we use new columns *week* and *weekday* to store the results because the column *date* is used twice. The final results are shown in Figure 10.4. I deliberately designed the set to be two full weeks of weekday which can be compared easily.

Now we have the necessary data for making a pivot

table. To do so, we use the pandas method:

```
pd.pivot_table(dataframe,
    values=...,
    index=...,
    columns=...)
```

where **values** are the data in the pivot table, and we pass the target columns into the **index** and **columns** arguments. In Figure 10.5 we show one realization of the pivot table by using *week* as the **row**, *weekday* as the **column**, and *apple* to be the **value**. We say it is one realization because we can choose other columns for the **values**, **index** and **columns**. They are chosen according to what we want to extract from the dataframe. This method returns the pivot table so we use the variable **pt_apple** to store it. Notice that there are more than one ways in constructing your pivot table. The exact structure will depend how you want to organize and visualize the data.

If we check the datatype of the pivot table, it is actually a dataframe (See Figure 10.6).

Finally, we can analyze the pivot table by the standard aggregation method or other standard statistical methods. For example, we can study the sum of apple sold in a week, and the maximum and minimum number sold in each week by the **agg** method. We show the results in Figure 10.7. If you are not familiar with the **agg** method, please review Chapter 9. We use **axis=1** to remind Pandas that we want to calculate the aggregation

```
pt_apple = pd.pivot_table(df1,
    values="apple",
    index="week",
    columns="weekday")
pt_apple
```

weekday	1	2	3	4	5
week					
20	14	18	20	21	24
21	21	18	20	26	27

Figure 10.5: Converting dataframe into a pivot table

```
In [219]: type(pt_apple)
Out[219]: pandas.core.frame.DataFrame
```

Figure 10.6: Datatype of a pivot table

```
In [221]: pt_apple.agg([sum,max,min], axis=1)
```

Out[221]:

	sum	max	min
20	97	24	14
21	112	27	18

Figure 10.7: Analyzing the pivot table

along the **row**, not by the **column**. If we set **axis=1** in this example, Pandas will return the sum of apple sales in every weekday, as well as their maximum and minimum number of sales in each weekday.

If you are familiar with the **groupby** method, you may see that we are actually grouping the entries by the *week* column and then collect their statistical features. Indeed, the whole process here can be reproduced by the following script:

```
df1.groupby("week")["apple"].agg([sum,max,min])
```

The command looks very compact at first sight, but it is in fact the virtue of Pandas which allows us to carry out multiple steps within a line of script. For example, in this code, we mean that

- we want to group the data by the *week* value,

- we choose the *apple* column from the GroupBy object,
- we ask Pandas to compute the sum, maximum and minimum in each subgroup.

Here, we do not set **axis=1** because the dataframe store each row as the sales of apple in one day. Doing aggregation along the column means we are summing the sales data of the same week. In fact aggregating along the row does not have meaning because there is only one data: the sales of apple at that particular day.

Whether or not you use a pivot table depends on the features and the processing procedure you need to do to the dataframe. Therefore there is no exact rules in which method (**groupby** or **pivot_table**) must be used for the analysis.

10.2 Summary

In this short chapter we have done a quick excursion in how to use the pivot table to collect data according to some of the column features. We have considered the case where we analyze the sales report of a fruit store. We have also touched some basic data processing related to datetime objects. Then we have demonstrated how to use the pivot table to extract some useful quantities such as the weekly sales.

10.3 Exercises

1. Repeat the exercise but for the column *orange* or *kiwi*. Try not to copy the code but to use what you have learned in this and previous chapters.

2. Consider the following dataframe in Figure 10.8, which is the closing price of a fiducial stock for 15 days:

   - Build a pivot table which stores the closing price in a calendar form
   - Compare the percentage of daily change in every two consecutive trading days (notice that you might need to start from the original dataframe first)
   - Find out which weekday has the highest average gain or loss on average
   - (extra) Repeat the same analysis by using only the dataframe. You will need to use the method **shift**

     ```
     df2["close"].shift(n)
     ```

     to create a duplicate of column but the data is shifted down by n rows. For your information, we often use the method **rolling** to create a window where the data within the nearest n

rows is grouped. This allows us to do a dynamical grouping such as moving average. For example,

```
df2["close"].rolling(3).mean()
```

will correspond to a moving average of 3 days. Notice that the rolling starts from the top to the bottom, which means that the first 2 rows (row 0 and 1) will have no results because there is no data about row -2 and -1 in the dataframe. The first result appears in row 2, which is the average of the close prices of row 0, 1 and 2

```
df2
```

Out[225]:

	date	close
0	2021-05-17	100.0
1	2021-05-18	102.3
2	2021-05-19	103.6
3	2021-05-20	104.8
4	2021-05-21	106.5
5	2021-05-24	107.8
6	2021-05-25	104.5
7	2021-05-26	105.5
8	2021-05-27	105.4
9	2021-05-28	104.3
10	2021-05-31	102.3
11	2021-06-01	104.1
12	2021-06-02	105.5
13	2021-06-03	106.6
14	2021-06-04	105.2

Figure 10.8: The closing price of a fiducial stock for Exercise 2

Chapter 11

Visualization

In this chapter we shall examine how to use the default plotting methods in Pandas for visualizing a dataframe. Visualization serves as an important function to efficiently grasp an overall structure of the data, which will guide us to do a more precise processing and statistical analysis. Pandas has the default plotting methods developed from the Matplotlib library. Many of the functions have overlap with the standard Matplotlib which are well discussed in the literature. Therefore, here we will focus on the principle use of these commands for generating graphs. We will illustrate the ideas based on a mini case study.

```
df1
```

Out[226]:

	Year	Apple	Orange
0	1999	20	30
1	2000	25	32
2	2001	27	28
3	2002	29	25

Figure 11.1: Reference dataframe for this chapter

11.1 Plotting Options in Pandas Dataframe

Let us assume our Pandas friend comes to us again with the dataframe shown in Figure 11.1. This is again a fictitious sales report about the shop's annual sales of apples and oranges. To visualize the data, we can choose to plot the bar chart or line chart to show their sales variation in time. We will need the command

```
df1.plot(kind="bar", x="Year", y="Apple")
```

for making a bar chart about the sales of apple (Figure 11.2) and

```
df1.plot(kind="line", x="Year", y="Apple")
```

for making a line chart (See Figure 11.3, with optional arguments included). This method returns us a Matplotlib **axis** object. If no column data is provided, Pandas will plot by default all the columns. There are many optional argument we can pass during the creation of the figure. We will introduce a few important ones, including:

- **title**: Set the title of the plot
- **xticks**, **ytick**: the numerical labels along the x-axis and y-axis
- **sharex**, **sharey**: allow figures to stack horizontally and vertically **color**: the color of the line/bar **xlim**, **ylim**: the range of x and y

We refer the readers to the Matplotlib documentation for the detailed description of the arguments. Some of the customizations, such as setting the x-axis or y-axis label, are not directly tunable as arguments in the Pandas method. In that case, we will need to tune the Matplotlib **axis** object which is returned by this method. The way to set up features in the *axis* object is the same as that in Matplotlib.

In Figure 11.3, for illustration purpose we set some optional arguments to increase the diversity of the figure. We have added **style** for the style of lines. We also set **marker** for the symbol used for each data point. A

```
df1.plot(kind="bar",
         x="Year",
         y="Apple",
         title="Apple sales")
```

```
<AxesSubplot:title={'center':'Apple sales'}, xlabel='Year'>
```

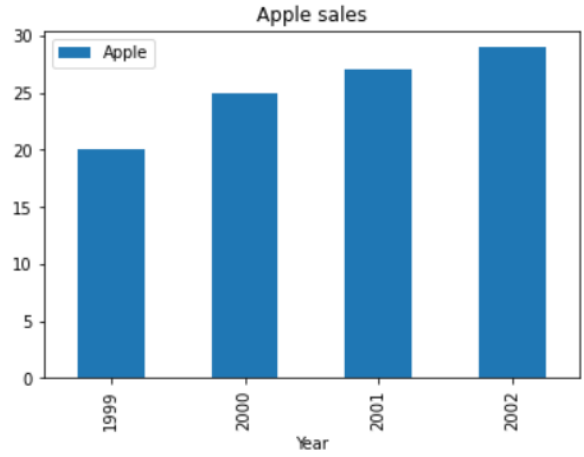


Figure 11.2: Bar chart for plotting the sales of apples in the 4 years with default settings

number of options are available so we refer the reader to this link for further information.

Pandas also provides other types of charts such as pie chart by using the **kind** parameter (See Figure **??** for its demonstration)

```
ax = df1.plot(kind="line",
              x="Year",
              y="Apple",
              title="Sales of Apple",
              color="red",
              style="--",
              xticks=[1999,2000,2001,2002],
              marker="s")

ax
```

```
<AxesSubplot:title={'center':'Sales of Apple'}, x
```

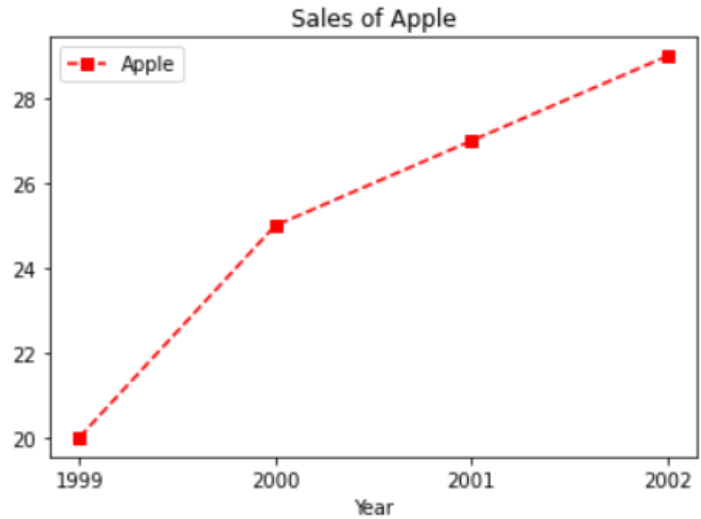


Figure 11.3: Line chart for plotting the sales of apples in the 4 years with optional arguments

```
df1.plot(kind="pie", y="Apple")
```

where histogram (**kind="hist"**), box chart (**kind="box"**) and area chart (**kind="area"**) can be obtained similarly. In the pie chart example, we further choose some useful arguments to pass. They include the **figsize** which controls the dimension of the figure, **fontsize** which controls the font size of the label and the **legend** option. We set the **legend=False** to remove the automatic legend. The pie chart option by default normalize the data automatically. We may immediately extract the part with the largest portion.

Finally, we can also generate a scatter plot to investigate the correlation of any pair of variables, by setting

```
df1.plot(kind="scatter", x="Apple", y="Orange")
```

Scatter plot is very useful to identify potential trends among data without the bias of default lines. In Figure 11.5 we plot the orange sales again the apple sales as an example. It is useful from the points to determine if there is (anti-)correlation existing among two columns. We also set some common arguments for demonstration, which include **alpha** for the opaqueness of the datapoint, **marker** as the same argument in Figure 11.3, and **s** for the size of the marker.

```
plot = df1.set_index("Year").plot(
    kind="pie",
    label="Apple sales",
  y="Apple",
  figsize=(5,5),
  fontsize=16,
  legend=False)
```

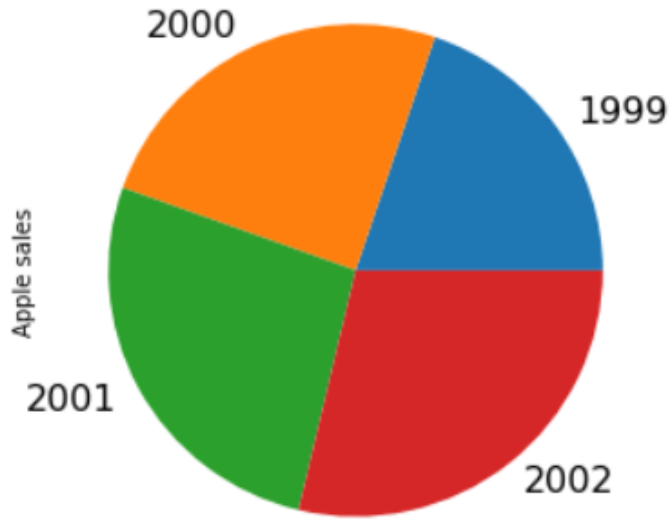


Figure 11.4: Pie chart for the sales of apples

```
ax = df1.plot(kind="scatter",
         x="Apple",
         y="Orange",
         c="green",
         alpha=0.5,
         marker="o",
         s=100,
         title="Fruit sales")

ax
```

```
<AxesSubplot:title={'center':'Fruit sales'}, xlabel
```

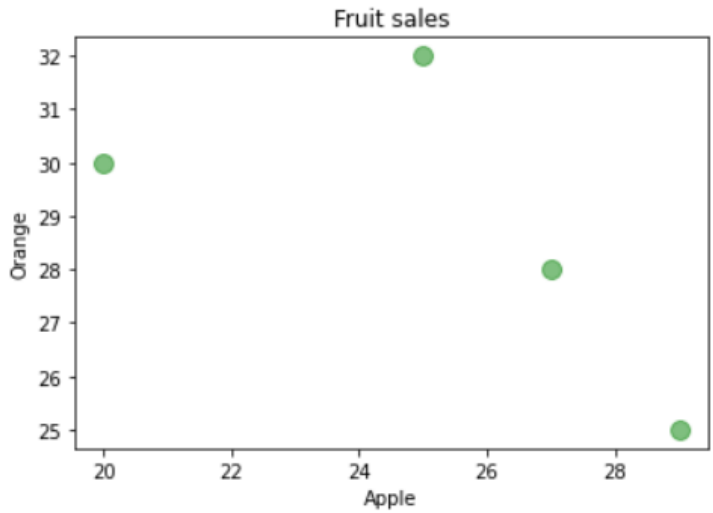


Figure 11.5: Scatter chart for the sales of apples and oranges

```
df_ex1
```

	Name	ID	Location	Email	Language	Salary	Position	Gender
0	Abel	A01	UK	abel@gmail.com	Python	2000	Staff	M
1	Belinda	B01	AU	belinda@yahoo.com	C	3000	Staff	F
2	Carla	A02	UK	carla@yahoo.com	C	1000	Helper	F
3	Don	A03	US	don@hotmail.com	Python	1000	Staff	M
4	Enson	B02	CA	enson@gmail.com	C++	2000	Assistant	M
5	Felipe	B03	NZ	felipe@yahoo.com	Java	3000	Helper	F
6	Georg	A04	CA	georg@hotmail.com	C++	3000	Assistant	F
7	Humprey	B04	US	humprey@yahoo.com	Python	2000	Staff	M

Figure 11.6: Reference dataframe for the study of Groupby object

11.2 Plotting Options in GroupBy Object

In this part we will use the same dataframe in Chapter 9 when we discuss the methods **groupby**. We show in Figure 11.6 the dataframe we will be using, which is the identical dataframe we have been using in Chapter 9. In general we use the **groupby** method to partition the dataframe into many parts. Then we can make the derived dataframe by using the aggregate method or we can plot each subgroup as an individual dataframe.

For example, we may group the entries by the column *Location*, and count how many rows in each subgroup by:

```
In [295]: df_grouped = df_ex1.groupby("Location").size()
          df_grouped.hist

Out[295]: Location
          AU    1
          CA    2
          NZ    1
          UK    2
          US    2
          dtype: int64
```

Figure 11.7: Grouping data by its "Location" and find its size

```
df_ex1_group = df_ex1.groupby["Location"].size()
```

The result in shown in Figure 11.7 where we can see each number corresponds to the number of staff in that particular location. Other aggregate feature, such as **sum**, **mean** and so on, can be done similarly.

Then we may use the derived dataframe (or series) to generate the plot by the standard method

```
df_grouped.plot(kind="bar")
```

where we show the result in Figure 11.8.

```
df_grouped.plot(kind="bar",
                yticks=[0,1,2],
                title="Member location")
```

```
<AxesSubplot:title={'center':'Member location'},
```

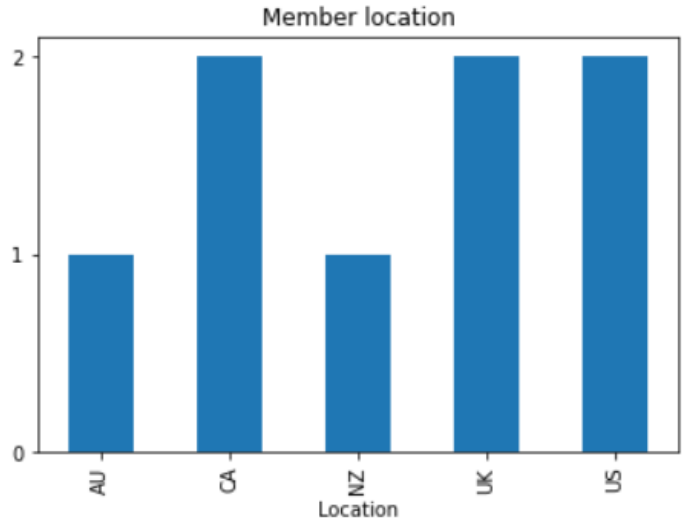


Figure 11.8: Plotting the derived dataframe from a GroupBy object

11.3 Summary

In this chapter we have discussed how to carry out visualization of a dataframe by native Pandas API. We have examined how the methods of bar chart, line and scattering plots, pie chart and so on, can be readily called from a dataframe object for a rapid making of a high quality graph. We have examined how this can be done for a dataframe object and a GroupBy object. We also include examples of how we can fine-tune the figure by passing a number of arguments.

The use of Matplotlib can also naturally visualize a dataframe without the call of native Pandas API. However, because the plotting library of Pandas has the backend of Matplotlib, we will not discuss in details the usage of Matplotlib in this note.

11.4 Exercises

1. Your friend comes to you again with the same stock data again and want to ask for your help to visualize the data (See Figure 11.9 for a quick review). Your friend wants to achieve the following tasks. Can you help your friend?

    - A line chart to show the price
    - A 3-day moving average of the price

- A historgram showing how many days the price is rising and how many days the price is falling

Hint: in case if you have not heard about this term, it means an average of price taken from the last three days. To do so, we need to rely on Pandas **rolling** method, which is available as Pandas Series or DataFrame method. You need to pass the window size as the argument. This methods set up a window for Pandas that in each row, Pandas can only consider data inside the window instead of the full column. It is very similar to the **agg** method we have discussed in Chapter 9. By definition, there is no 3-day average in the first two days.

```
df2
```

Out[225]:

	date	close
0	2021-05-17	100.0
1	2021-05-18	102.3
2	2021-05-19	103.6
3	2021-05-20	104.8
4	2021-05-21	106.5
5	2021-05-24	107.8
6	2021-05-25	104.5
7	2021-05-26	105.5
8	2021-05-27	105.4
9	2021-05-28	104.3
10	2021-05-31	102.3
11	2021-06-01	104.1
12	2021-06-02	105.5
13	2021-06-03	106.6
14	2021-06-04	105.2

Figure 11.9: The dataframe of fiducial stock data for exercise 1

Chapter 12

What can you do next?

Congratulations! If you follow from Chapter 1 up to now, I appreciate very much your motivation and persistence in learning Pandas! We have indeed studied a lot what we can do in Pandas.

In this chapter we will briefly review the important concepts covered in this book. Then we will discuss what we can do to further strengthen our skills in Pandas. We will also touch the essential procedure or skills in how to efficiently process a database by Pandas.

12.1 Brief Review

- In Chapter 2 we focused on how to set up a dataframe from scratch using a list, a dictionary, a csv file

and an excel file. We introduce the ideas of row and column names (index and column header). We presented how to use the command **DataFrame**, **read_csv** and **read_excel** to pass these data into a Pandas DataFrame.

- In Chapter 3 we discussed how to call individual or collective entries of data by three types of notations, namely index notation, colon notation and list notation. We described extensively how the **loc** and **iloc** methods are used to call the data.

- In Chapter 4 we presented how to add, modify and remove entries in a Dataframe.

- In Chapter 5 we talked about how to describe a dataframe from its structure and its numerical data features by using the methods/attributes **index**, **columns**, **nunique**, **describe** and so on.

- In Chapter 6 and 7 we elaborated in details how to process numerical and textual data collectively in the column level. We demonstrate how this can be done by the powerful method **apply**. We also surveyed some other methods such as **mask** and **where**.

- In Chapter 8 we examined the indexing procedure and the merging procedure of multiple dataframes.

We discussed how the index can be used for classification and how index can be used for merging multiple dataframe into one by the method **merge**.

- In Chapter 9 we surveyed the methods **groupby** and **agg** for building a groupby object and computing the aggregated quantity in individual subdataframe.

- In Chapter 10 we discussed about the construction and applications of a pivot table by the method **pivot_table**. We learned how to rearrange the data to gain extra insight along specific column features.

- Finally in Chapter 11 we have surveyed numerous ways to visualize the data by using the **plot** module inherent to the Dataframe class.

12.2 What to do next?

One frequent problem when learning a programming language is that we assume we know the skills after reading the tutorial somewhere online or following the assignment in an online course. This is usually not the case. It is not because the tutorial or the online course is less well designed. But it is because the skills we have read from

these medium do not directly go into our long term memory. That means, once we start learning something else, very likely these new skills from previous courses will be gradually lost.

Moreover, in this booklets we have focused only the essential methods and attributes of a dataset. There are still many useful ones which I do not have space to pack in. Some of these can be time-series related such as correlation (**cor**), exponential weighted mean (**ewm**), shifting (**shift**) and so on. Some of these can be handling missing data such as interpolation (**interpolate**), filling missing entries (**fillna**), or removing missing entries (**dropna**). And there are sophisticated Pandas method such as **pipe** which lets you do a chain of transformation. So, if we really want to make full use of Pandas to benefit our data processing skills, there are still many places for us to explore and to enjoy!

If we want to keep these new skills in my mind, the most important we need to do is practice! We really need to apply these methods or applications to some datasets so that we can recall them casually as our convenient toolbox. Different from other branches, the data science is a rapidly growing branch. Not only there are numerous resources available everywhere, but also the data! Once we have the data, we can actually try to process the data as if we need to use the data for machine learning. And these data will be a good starting point to practice our skills!

One of the resources I frequently visit is Kaggle. It is a database of datasets. It is the directory to store a lot of open-source datasets contributed by users around the world. Some of these can be the datasets from their assignments. Some of these can be the source data for their publications of academic articles. The domain of the data is very wide. Data related to medical studies, business, finance, social network, and scientific data is available. It is very likely that there is at least one or two datasets which may fit your expertise so that you may understand the actual meaning of the data while processing the data. However, I will also encourage you to have a taste of data which is outside your domain. This will help you to test if you know what to do with a new set of data.

Kaggle itself is also a platform. You can view some of the data analysis results of many other users who kindly share their prescription in handling these datasets. Certainly, one way to benefit from their experience is to read their code and try to repeat on your own computer. The shortcut of Ctrl+C and Ctrl+V will not do much to benefit your skills. By typing the code by our hands, our brain also develops muscle memory about the important commands and the procedure in how data is generally handled. It will further solidify all the new skills we have done in these tutorials. I can't stress enough that it takes time to practice to internalize our mastery in data processing.

Moreover, there are also competitions which are held by the organizers or companies for you to test your skills in processing the data. Some of these have monetary rewards and some let you gain the experience in the process. These experiences will provide you solid test in how well you have mastered data processing and data analysis. Certainly, this book only talks about the hard skills in data processing. We have not yet touched any about data analysis. This will be the second step to move one in our journey of data science.

12.3 Procedure in handling dataset

Let us assume we have a dataset taken from Kaggle. Then what should we do in order to benefit from this? Here I will outline a schematic path how I usually deal with a new dataset I have never seen.

1. **Check the basic structure of the data**

   I will first check how the data looks like. It will include inspecting some sample entries in a dataset to see what each column contains. This will help us to visualize what each column stands for and what the potential data is. I will also take a look at the datatype of each columns and their possible data

range. (I hope at this stage you will already know which methods or attributes we need to call in order to do these operations.) They will help us to see if we need to fill in missing entries or remove missing entries so that the data can be later aggregated meaningfully later.

2. **Orient yourself**

   A usual dataset is far more complicated than the examples we have used in this book. The number of rows and columns can be orders of magnitudes larger than those used in this book. Thus, it could be overwhelming if we want to gather all the information in one place at once. Usually on Kaggle there are some tasks available in the datasets. We can treat those tasks as our objective. And we can design the data processing pipeline according to the task. When there is no task available, then we need to imagine one at the first place. We need to do so early one because it will determine how we will transform the data. And most of the time in deriving the useful data, we are actually removing the irrelevant ones. And whether or not that part of data being irrelevant depends solely on our objective.

   For example, assume we have a transaction record of a shop which contains all the customers and their

shopping record in a year. Things we can ask ourselves about the dataset could be (1) how much does each customer spend? (2) How frequent does each customer buy in that shop? (3) What are the most popular items in that shop? (4) Is there any pattern in the collective behaviours of all customers?

3. **Process pipeline**

   After we know what our data is and what we want, we can then proceed to build our data transformation pipeline. Many of the frequently used methods are already covered in this book. The process can be building new columns by aggregating existing columns. We can also filter the dataset by querying or grouping to reduce the complexity and variation of the data. In the columns which contain text, we need to consider which keywords are relevant to be extracted.

   Using the transaction record example in the previous point, we can do groupby and aggregation to collect the total expense of each customer. We can do sampling to partition the dataset into regular intervals for obtaining the weekly or monthly sales record. We can also select customers based on the total expenses they have spent.

4. **Visualization**

After we have made our data into pretty columns, we can use the plotting library to visualize the data. Notice that to draw a graph is not always for presentation use. It can be used for ourselves to catch the hidden structure hidden in the data by inspection. This will later guide us how we want to use the data for data analysis. How to choose the plotting style is also important. A good plotting style can quickly let us appreciate if there is a trend or clustering in the dataset. However, as this book is not about data analysis I will not go into details at this moment.

Using the same transaction record as an example, we can plot a bar chart to describe the sales of specific items. We can also plot a line chart to describe the sales record in each day, week or month. We can also make a scatter plot to see how the customer shopping habit depends on the demographics.

5. **Data Analysis**

   Finally, we can proceed from data process to data analysis. As this book is not about data analysis, which will need a much larger extent to discuss the methodology, mechanism and procedure, we will only go through it schematically. Once we know the pattern of the data, we can use standard supervised or non-supervised learning to do clas-

sification and prediction. This actually contains two parts: (1) Whether we can use a model to describe the behaviour in the dataset and (2) How the model can generate new prediction for future use. Both of them will rely on extensive machine learning language so we will not attempt to make a hasty sketch.

6. Presentation

   After all these steps, I believe it will bring you a complete analysis of the dataset you have in hand. We will need to wrap up all the calculations in a carefully documented code so that future readers can actually learn from your discovery. A good report will not only include the finely tuned figures or machine learning results, but also a description what can be told from these figures and from these numerical results, and what we can do about these results. The actual context will depend on the dataset and the objective we have described in Points 1 and 2.

12.4 Advertisement

Finally let me briefly promote about what I will do next after this project. There are a lot of things I want to cover in Python and many of them are exciting projects!

It needs not to say that the very versatile functionality of Python and its libraries will cover almost everything you have in mind. Some of the topics I think interesting will be:

- Learning Data Science from Projects

  We have already done in this book up to Step 4 described in our Procedure. However, most of these are described as hard skills as there is no specific context or applications accompanied. In the new projects, I hope to make reference to some real datasets and demonstrate what we can do with them. We will cover all the essential skills deployed for data analysis. In the meantime, we will discuss what should be noticed or cautioned, and what are the good ways to gain insight from the data.

- Design and make your game by PyGame

  Making game is always my hobby. Playing game is fun of course, but making game is a lot of fun. It is actually a robust programming experience to do the frontend and backend programming. Frontend here means to build the algorithm to convert your game to something visible on the screen where users can interact with the game. Backend here means to design the game algorithm and logic which handle users' input and interaction of their input. In the project I hope to demonstrate from some of my

game examples to diagnose how to make a game, its procedure and how to implement it, in the context of PyGame. I show one sample in Figure 12.1.

- Make your own simulations by Python

  Because of my expertise, I have been doing simulations in physics for a long time. And there are a lot of interesting simulations we can do on our own laptop or desktop computers as if we are a scientist! These simulations (or formally known as numerical experiments) will help us learning a lot how our universe works. We will touch about how stars orbits around each other, how Earthquake or avalanche can be modeled, how chemical reactions are taking place. There are literally infinite ways to use Python to create experiment to understand different physical processes in the world. I will select some cool topics and discuss how simulations can help us to find the truth behind these phenomena. Furthermore I will pay attention on how the visualization and interaction with the experiments.

Certainly there are many other things we can talk about about Python. Feel free to let me know and send me a message if you happen to have a cool idea. For now, let me farewell and I wish you good luck in learning Python and Pandas.

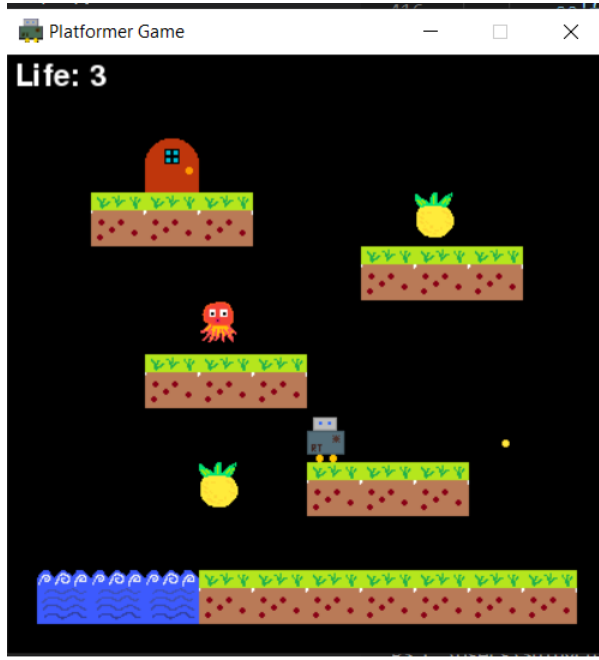


Figure 12.1: A screen shot of the platformer game built by PyGame. See the lovely robot shoots a bullet to the right! The robots needs to find its way home and avoid the monster.

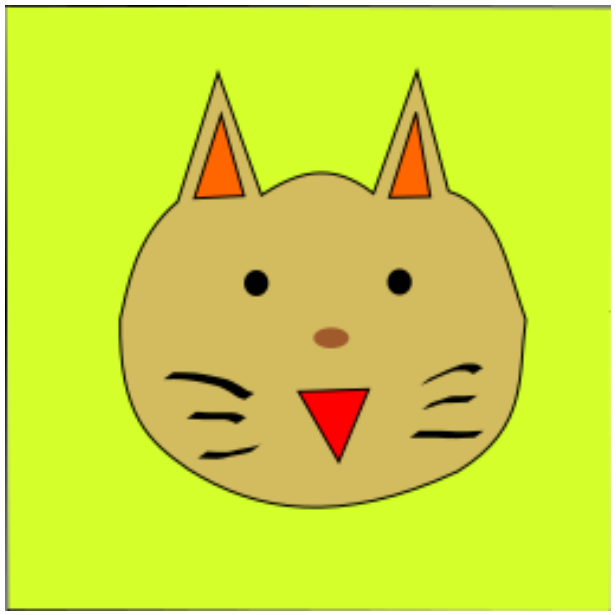


Figure 12.2: *"See you soon!"* said by the cat Maumau.