Chapter 3 Data analytics

3.1 Introduction to Python data analytics

We have introduced descriptive statistics for describing the characteristics of a set of data. However, the computation can be tedious when the data size is large. Computer programming can be used as a tool for making such computations. In this chapter, we will introduce useful techniques for data analytics using Python with **Pandas**.

Pandas stand for Python Data Analysis. It is a library that contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python. Pandas adopts array-based computing, which allows data processing without for loops. It is designed for working with tabular or heterogeneous data. Before using the functions in Pandas library, we have to import it first by. When we call functions in Pandas, we can start with "`pd.`".

```
import pandas as pd
```

In Pandas, data can be stored into two forms, **Series** and **DataFrame**. Notice that the letters 'S', 'D' and 'F' are capital. A Series is a one-dimensional array-like object containing a sequence of values and an associated array of data labels, called its index. We can simply convert a list of numerical values into a Series by applying `pd.Series()`. For example, we can convert the list of first 5 prime numbers into a Series. The index by default is 0 to 4.

```
x = pd.Series([2,3,5,7,11])
```

```
x
```

```
0     2
1     3
2     5
3     7
4    11
dtype: int64
```

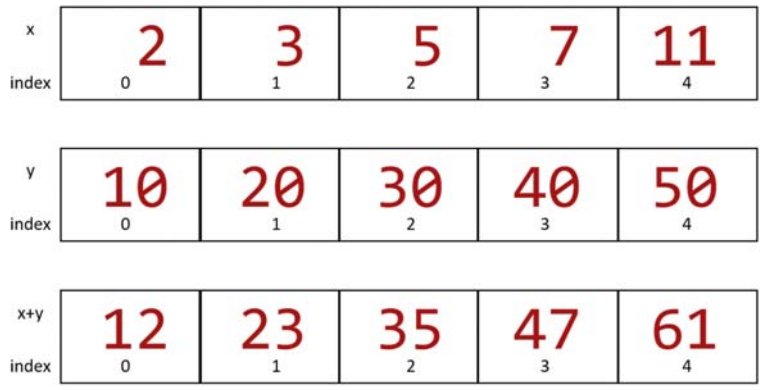
However, we may also rename the index as another list of string or numbers.

```
y = pd.Series([2,3,5,7,11],index=['a','b','c','d','e'])
```

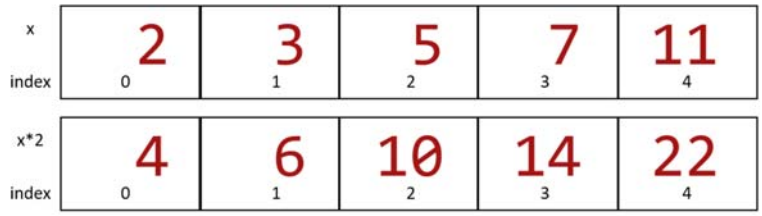
```
y
```

```
a     2
b     3
c     5
d     7
e    11
dtype: int64
```

One useful feature of Series is **vectorized computation**. If we apply an arithmetic operation between two Series, the entries of the corresponding indexes will be calculated, resulting a new Series.



If we apply an arithmetic operation between a Series and a number, the operation will be applied to each entry of the Series.



We can verify this result in Python.

```
x = pd.Series([2,3,5,7,11])
y = pd.Series([10,20,30,40,50])
```

```
x+y
```

```
0    12
1    23
2    35
3    47
4    61
dtype: int64
```

```
x*2
```

```
0     4
1     6
2    10
3    14
4    22
dtype: int64
```

With vectorized computation, we can apply standardization to the whole Series easily.

```
A = pd.Series([28,29,30,31,32])
Z = (A-30)/2**0.5
```

```
Z
```

```
0   -1.414214
1   -0.707107
2    0.000000
3    0.707107
4    1.414214
```

3.2 Descriptive statistics and arithmetic in Pandas

Pandas Series are equipped with a set of common mathematical and statistical methods. The table below shows some common descriptive statistics methods in Pandas. To apply these methods, put a dot " `.` " after a Series and then call the functions.

Statistical meaning	Function
sum	`sum()`
arithmetic mean	`mean()`
median	`median()`
maximum	`max()`
minimum	`min()`
mode	`mode()`

Statistical meaning	Function
sample size	`count()`
mean absolute deviation	`mad()`
population variance	`var(ddof=0)`
sample variance	`var(ddof=1)`
population standard deviation	`std(ddof=0)`
sample standard deviation	`std(ddof=1)`

Using these functions, we can verify the results in the previous chapter.

```
A = pd.Series([28,29,30,31,32])
B = pd.Series([10,10,30,50,50])
```

```
A.mean()
```

```
30.0
```

```
B.mean()
```

```
30.0
```

```
A.median()
```

```
30.0
```

```
B.median()
```

```
30.0
```

```
A.mad()
```

```
1.2
```

```
B.mad()
```

```
16.0
```

```
A.var(ddof=0)
```

```
2.0
```

```
B.var(ddof=0)
```

```
320.0
```

```
A.std(ddof=0)
```

```
1.4142135623730951
```

```
B.std(ddof=0)
```

```
17.88854381999832
```

Instead of showing each particular item, we can also generate a summarized result of the descriptive statistics of a Series using the method `describe()`. Notice that the `std` here refers to sample standard deviation.

```
A.describe()
```

```
count     5.000000
mean     30.000000
std       1.581139
min      28.000000
25%      29.000000
50%      30.000000
75%      31.000000
max      32.000000
dtype: float64
```

```
B.describe()
```

```
count     5.0
mean     30.0
std      20.0
min      10.0
25%      10.0
50%      30.0
75%      50.0
max      50.0
dtype: float64
```

The numbers resulted from the describe method are calculated based on the data values. However, some methods return the indirect statistics. For example, we know the maximum value in Series A is 32, but we might not know where this value is. The functions `idxmax()` and `idxmin()` returns the index of the maximum/minimum entry in a Series.

```
A.idxmax()
```

```
4
```

```
A.idxmin()
```

```
0
```

In Series B, we find that there are multiple entries with the same value. If we want to check the unique values and the frequency of each unique value in a Series, we can use the function `value_counts()` which returns a new Series with its index being the unique values and its entries being the frequency of each unique values. This is what we call a **frequency table**.

```
B.value_counts()
```

```
10    2
50    2
30    1
dtype: int64
```

Series is only a one-dimensional array-type of data structure, which is only suitable for storing data set of a single variable. However, in real-life the data file might contain multiple variables. For example, the health record of a class of students might contain their gender (string), height (float) and weight (float). Each of these three variables can stored as a Series, and we can combine these Series together into a two-dimensional tabular form called DataFrame. Each of the Series is regarded as one column in the DataFrame. In order distinguish, we can also give names to these columns.

In the example below, we first define the three Series with values and names. Then we combine the Series together into a DataFrame using the `concat` method. The syntax is:

```
df_name = pd.concat([Series1, Series2, ...], axis = 1)
```

```
x1 = pd.Series(['M','F','M','F','F'],name='sex')
x2 = pd.Series([1.73,1.61,1.80,1.56,1.69],name='height')
x3 = pd.Series([65,54,72,63,58],name='weight')
df = pd.concat([x1,x2,x3],axis=1)
```

The DataFrame is displayed in tabular form tidily as shown below.

```
df
```

	sex	height	weight
0	M	1.73	65
1	F	1.61	54
2	M	1.80	72
3	F	1.56	63
4	F	1.69	58

Since DataFrame is a two-dimensional data structure, we can call out either a row, a column or a single entry from a DataFrame. To call a single column, directly use the column name:

```
df['height']
```

```
0    1.73
1    1.61
2    1.80
3    1.56
4    1.69
Name: height, dtype: float64
```

To call a single row, apply the `loc` method and use the index of the row:

```
df.loc[2]
```

```
sex         M
height    1.8
weight     72
Name: 2, dtype: object
```

To call a single entry, we can apply the `loc` method and include both row index and column name of the target entry.

```
df.loc[2,'height']
```

```
1.8
```

For a DataFrame, we might want to add new columns based on some arithmetic of the existing columns. Recall that since a column of a DataFrame is a Series, it also supports vectorized computation. If we make arithmetic operations between two columns, the result is a Series with the same dimension. We can create a new column in a DataFrame with such result. The syntax is:

```
DataFrame_name["new_col_name"] = Series_name
```

For example, we would like to create a new column of weight in pounds, which equals to weight in kilograms multiplied by 2.2. We would like to create another column of BMI (body mass index) which is the weight in kg over square of height in m. Refer to the coding below:

```
df["weight(pound)"] = df["weight"]*2.2
df["BMI"] = df["weight"]/df["height"]**2
```

```
df
```

	sex	height	weight	weight(pound)	BMI
0	M	1.73	65	143.0	21.718066
1	F	1.61	54	118.8	20.832530
2	M	1.80	72	158.4	22.222222
3	F	1.56	63	138.6	25.887574
4	F	1.69	58	127.6	20.307412

3.3 Data loading

In the previous section, the data no matter in Series or DataFrame form are input one-by-one on our own. In reality, this would be impossible due to the volume of big-data. Pandas provides various method to read data from various file formats or sources into a DataFrame for analytics.

One common type of data file is **comma-separated values (csv)** file. It is a delimited text file that uses a comma to separate values. Each line is regarded as a data record. However, the first line is usually used as column titles, indicating the meaning of the values stored in this column. When opened on Excel, the values are automatically arranged by rows and columns without showing the commas. When opened on Notepad, each record is stored on a line with its values separated by commas.

In the example below, we would like to read data from an excel file "health1.csv" which already preserve the first row as the header, providing information of each column. The header row will be converted to the column names of the DataFrame and not regarded as data values. The syntax is:

```
df_name = pd.read_csv("file_path")
```

```
df1 = pd.read_csv("health1.csv")
```

	A	B	C
1	sex	height	weight
2	M	1.73	65
3	F	1.61	54
4	M	1.8	72
5	F	1.56	63
6	F	1.69	58

	sex	height	weight
0	M	1.73	65
1	F	1.61	54
2	M	1.80	72
3	F	1.56	63
4	F	1.69	58

If the data file doesn't contain a header as in "health2.csv", we need to specify by putting `header=None`. The column names in the DataFrame created will be by default 0,1,2,...

```
df2 = pd.read_csv("health2.csv",header=None)
```

	A	B	C
1	M	1.73	65
2	F	1.61	54
3	M	1.8	72
4	F	1.56	63
5	F	1.69	58

	0	1	2
0	M	1.73	65
1	F	1.61	54
2	M	1.80	72
3	F	1.56	63
4	F	1.69	58

In case we would like to add column names to a file without header, we can use `names`:

```
df2 = pd.read_csv("health2.csv",header=None, names=['sex','height','weight'])
```

By default, the DataFrame created from reading a data file will be assigned with index 0,1,2,... and so on. In some data file, one of the column might contain the index of this set of data. If you wish to set a particular column from the data file to be the index column, put the parameter `index_col` within the brackets of `read_csv`. In the file "health3.csv", a column called "id" contains the student identity number. We can set it to be the index column as this value can uniquely distinguish different rows (students).

```
df3 = pd.read_csv("health3.csv",index_col="id")
```

	A	B	C	D
1	id	sex	height	weight
2	20345678	M	1.73	65
3	20999999	F	1.61	54
4	21000001	M	1.8	72
5	21000456	F	1.56	63
6	22010101	F	1.69	58

id	sex	height	weight
20345678	M	1.73	65
20999999	F	1.61	54
21000001	M	1.80	72
21000456	F	1.56	63
22010101	F	1.69	58

In case the data file is not a csv file, we can still read it using `read_table` method. But we need to specify the delimiting character, which separate between values. In the example below, the file "health.txt" use whitespace as the delimiting character. We can read it by:

```
df4 = pd.read_table("health.txt", sep=' ')
```

Notice that when you read a data file, you need to ensure that the file is under the same location of your python file.

Suppose we have performed data processing and updated a DataFrame. We can export it into a data file for storage and sharing using to_`csv method`. The newly created csv file will be put under the same location of your python file by default.

```
df4.to_csv("health4.csv")
```

3.4 Data preparation and data cleaning

We have learnt data loading and simple data analytics. But in reality, you will find a gap between these two steps. Due to the process of data collection, the original data file might contain problematic entries and hence not ready for carrying out data analytics. To fill in this gap, we need a process called **data preparation**. In data preparation, the most important process is **data cleaning**. It is a process to fix or remove incorrect, corrupted or missing data.

In the example below, two entries in the csv file are replaced by a word "unknown" and an empty cell. When this file is read as a DataFrame, they are not regarded as numerical values. The "unknown" is read as a string and not valid for statistical measures such as `mean()` or `sum()`. An error message will occur. On the other hand, the empty cell is read as `NaN` which means Not-a-Number. The statistical measures can still be evaluated but this entry will be ignored. Oppositely, if you want to empty the value in a cell, you can enter `None`.

	A	B	C
1	sex	height	weight
2	M	1.73	
3	F	1.61	54
4	M	1.8	72
5	F	unknown	63
6	F	1.69	58

```
df5 = pd.read_csv("health5.csv")
```

	sex	height	weight
0	M	1.73	NaN
1	F	1.61	54.0
2	M	1.8	72.0
3	F	unknown	63.0
4	F	1.69	58.0

To remedy these problematic entries, we might not want to edit it one-by-one (as there might be thousands of such in a set of big-data!). Instead, we can use some existing methods in Pandas. Applying `fillna(0)` to a Series, DataFrame or a particular column of a DataFrame replaces all the `NaN` by 0. This 0 can be changed to other values. As a more general way, the `replace()` method allow you to replace any old value in the Series/DataFrame to a new value. The old and new values have to be specified inside the brackets separated by comma.

```
df5.fillna(0)
```

	sex	height	weight
0	M	1.73	0.0
1	F	1.61	54.0
2	M	1.8	72.0
3	F	unknown	63.0
4	F	1.69	58.0

```
df5.replace("unknown",0)
```

	sex	height	weight
0	M	1.73	NaN
1	F	1.61	54.0
2	M	1.8	72.0
3	F	0	63.0
4	F	1.69	58.0

Notice that for either these two methods, the result is another object (Series or DataFrame) without changing the original one. To update the original object, assign it to the new object.

```
df5 = df5.fillna(0)
df5 = df5.replace("unknown",0)
df5
```

	sex	height	weight
0	M	1.73	0.0
1	F	1.61	54.0
2	M	1.8	72.0
3	F	0	63.0
4	F	1.69	58.0

In data preparation, another useful technique is **data filtering** which refers to selecting desirable samples from the dataset under some certain criteria. In a pandas DataFrame, such criteria can be based on the values of its columns. The syntax is as follows:

```
new_df = old_df[old_df["col_name"](relation)(number)]
```

For example, let's consider the original DataFrame `df1` containing the health data of 5 students in the previous session. Suppose we would like to create two new DataFrames by separating `df` into the two gender groups. We can check if the value in `df["sex"]` is equal to `"M"` or `"F"`. Notice that for equality we use double equal signs `==`.

```
df1_m = df1[df1["sex"]=="M"]
df1_f = df1[df1["sex"]=="F"]
```

```
df1_m
```

	sex	height	weight
0	M	1.73	65
2	M	1.80	72

```
df1_f
```

	sex	height	weight
1	F	1.61	54
3	F	1.56	63
4	F	1.69	58

The criteria can also be an inequality. For example, we can filter the data based on the height being greater than or equal to 1.8 m, or the weight lower than 60 kg.

```
df1_tall = df1[df1["height"]>=1.8]
df1_light = df1[df1["weight"]<60]
```

```
df1_tall
```

	sex	height	weight
2	M	1.8	72

```
df1_light
```

	sex	height	weight
1	F	1.61	54
4	F	1.69	58