Working with Data

Short Introduction to Machine Learning

In the previous weeks we have discussed how various computation can be done. We begin by discussing computational complexity and divide and conquer strategy through recursion. We then discusses how data can have computation associated with it through object oriented programming. We discussed several data structures and algorithms associated with various problems. We then end up with offering a different perspective by looking at computation as a state machine.

In this second half of the course, we will look into how computation can learn from data in order to make a new computation. This new computation is often called a **prediction**. In these lessons, we focus on what is called as **supervised machine learning**. The word supervised machine learning indicates that the computer learns from some existing data on how to compute the prediction. An example of this would be given some images labelled as "cat" and "not a cat", the computer can learn to predict (or to compute) whether any a new image given to it is a cat or not a cat.

cat	not a cat
	

Another example would be given a some data of housing prices in Singapore with the year of sale, area, number of rooms, and its floor hight, the computer can predict the prize of another house. One other example would be given data of breast cancel cell and its measurements, one can predict whether the cell is malignant or benight. Supervised machine learning assumes that we have some existing data **labelled** with this category "malignant" and "benign". Using this labelled data (supervised), the computer can predict the category given some new data.

Reading Data

The first step in machine learning would be to understand the data itself. In order to do that we need to be able to read data from some source. One common source is a text file in the form of CSV format (comma separated value). Another common format is Excel spreadsheet. The data can be from some databases or some server. Different data sources will require different ways of handling it. But in all those cases we will need to know how to read those data.

For this purpose we will use Pandas library to read our data. We import the data into our Python code by typing the following code.

```
import pandas as pd
```

Now we can use Pandas functions to read the data. For example, if we want to read a CSV file, we can simply type:

```
df = pd.read_csv('mydata.csv')
```

Let's take an example of Singapore housing prices. We can get some of these data from Data.gov.sg. We have downloaded the CSV file so that you can access it simply from the following dropbox link. We can use the url to the raw file to open the CSV in our Python code.

In [1]:

```
import pandas as pd

file_url = 'https://www.dropbox.com/s/jz8ck0obu9u1rng/resale-flat-prices-based-on-registration-date-from-jan-2017-onwards.csv?raw=1'
df = pd.read_csv(file_url)
df
```

Out[1]:

	month	town	flat_type	block	street_name	storey_range	floor_area_sqm	flat_model	lease_commence_date	remaining_le
0	2017-01	ANG MO KIO	2 ROOM	406	ANG MO KIO AVE 10	10 TO 12	44.0	Improved	1979	61 year mo
1	2017-01	ANG MO KIO	3 ROOM	108	ANG MO KIO AVE 4	01 TO 03	67.0	New Generation	1978	60 year mo
2	2017-01	ANG MO KIO	3 ROOM	602	ANG MO KIO AVE 5	01 TO 03	67.0	New Generation	1980	62 year mo
3	2017-01	ANG MO KIO	3 ROOM	465	ANG MO KIO AVE 10	04 TO 06	68.0	New Generation	1980	62 year mo
4	2017-01	ANG MO KIO	3 ROOM	601	ANG MO KIO AVE 5	01 TO 03	67.0	New Generation	1980	62 year mo
...	...	...	...	...	...	...	...	...	...	
95853	2021-04	YISHUN	EXECUTIVE	326	YISHUN RING RD	10 TO 12	146.0	Maisonette	1988	66 year mo
95854	2021-04	YISHUN	EXECUTIVE	360	YISHUN RING RD	04 TO 06	146.0	Maisonette	1988	66 year mo
95855	2021-04	YISHUN	EXECUTIVE	326	YISHUN RING RD	10 TO 12	146.0	Maisonette	1988	66 year mo
95856	2021-04	YISHUN	EXECUTIVE	355	YISHUN RING RD	10 TO 12	146.0	Maisonette	1988	66 year mo
95857	2021-04	YISHUN	EXECUTIVE	277	YISHUN ST 22	04 TO 06	146.0	Maisonette	1985	63 year mo

95858 rows × 11 columns

The output of `read_csv()` function is in Pandas' `DataFrame` type.

In [2]:

```
type(df)
```

Out[2]:

```
pandas.core.frame.DataFrame
```

`DataFrame` is Pandas' class that contains attributes and methods to work with a tabular data as shown above. Recall that we can create our own custom data type using the keyword `class` and define its attributes and methods. We can even override some of Python operators to work with our new data type. This is what Pandas library does with `DataFrame`. This `DataFrame` class provides some properties and methods that allows us to work with tabular data.

For example, we can get all the name of the columns in our data frame using `df.columns` properties.

In [3]:

```
df.columns
```

Out[3]:

```
Index(['month', 'town', 'flat_type', 'block', 'street_name', 'storey_range',
       'floor_area_sqm', 'flat_model', 'lease_commence_date',
       'remaining_lease', 'resale_price'],
      dtype='object')
```

We can also get the index (the names on the rows) using `df.index`.

In [4]:

```
df.index
```

Out[4]:

```
RangeIndex(start=0, stop=95858, step=1)
```

We can also treat this data frame as a kind of matrix to find its shape using `df.shape`.

In [5]:

```
df.shape
```

Out[5]:

```
(95858, 11)
```

As we can see, the data contains 95858 rows and 11 columns. One of the column names is called `resale_price`. Since our aim is to predict the house price, this column is usually called the **target**. The rest of the columns is called the **features**. This means that we have about 10 feature columns.

The idea supervised machine learning is that using the given data such as shown above, the computer would like to predict what is the *target* give a new set of *features*. The computer does this by *learning* the existing labelled data. The label in this case is the resale price from the historical sales data.

In order to understand the data, it is important to be able to manipulate and work on the data frame.

Data Frame Operations

It is important to know how to manipulate the data. Pandas has two data structures:

- Series
- DataFrame

You can consider `Series` data as one-dimensional labelled array while `DataFrame` data as two-dimensional labelled data structure. For example, the table that we saw previously, which is the output of `read_csv()` function, is a `DataFrame` because it has both rows and columns and, therefore, two dimensional. On the other hand, we can access just one of the columns from the data frame to get a `Series` data.

Getting a Column or a Row as a Series

You can access the column data as series using the square bracket operator.

```
df[column_name]
```

In [6]:

```
print(df['resale_price'])
print(type(df['resale_price']))
```

```
0         232000.0
1         250000.0
2         262000.0
3         265000.0
4         265000.0
            ...
95853     650000.0
95854     645000.0
95855     585000.0
95856     675000.0
95857     625000.0
Name: resale_price, Length: 95858, dtype: float64
<class 'pandas.core.series.Series'>
```

The code above prints the column `resale_price` and its type. As can be seen the type of the output is a `Series` data type.

You can also get some particular row by specifying its `index`.

You can also access the column using the `.loc[index, column]` method. In this method, you need to specify the labels of the index. For example, to access all the rows for a particular column called `resale_price`, we can do as follows. Notice that we use `:` to access all the rows. Moreover, we specify the name of the columns in the code below.

use `:` to access all the rows. Moreover, we specify the name of the columns in the code below.

In [7]:

```
print(df.loc[:, 'resale_price'])
print(type(df.loc[:, 'resale_price']))
```

```
0         232000.0
1         250000.0
2         262000.0
3         265000.0
4         265000.0
            ...
95853     650000.0
95854     645000.0
95855     585000.0
95856     675000.0
95857     625000.0
Name: resale_price, Length: 95858, dtype: float64
<class 'pandas.core.series.Series'>
```

In the above code, we set the index to access all rows by using `:`. Recall that in Python's list slicing, we also use `:` to access all the element. Similarly here, we use `:` to access all the rows. In a similar way, we can use `:` to access all the columns, e.g. `df.loc[:, :]` will copy the whole data frame.

This also gives you a hint how to access a particular row. Let's say, you only want to acces the first row, you can type the following.

In [8]:

```
print(df.loc[0, :])
print(type(df.loc[0, :]))
```

```
month                             2017-01
town                           ANG MO KIO
flat_type                          2 ROOM
block                                 406
street_name              ANG MO KIO AVE 10
storey_range                     10 TO 12
floor_area_sqm                         44
flat_model                       Improved
lease_commence_date                  1979
remaining_lease         61 years 04 months
resale_price                       232000
Name: 0, dtype: object
<class 'pandas.core.series.Series'>
```

In the above code, we access the first row, which is at index 0, and all the columns.

Recall that all these data are of the type `Series`. You can create a Data Frame from an existing series just like when you create any other object by instantiating a `DataFrame` object and passing on an argument as shown below.

In [9]:

```
df_row0 = pd.DataFrame(df.loc[0, :])
df_row0
```

Out[9]:

	0
month	2017-01
town	ANG MO KIO
flat_type	2 ROOM
block	406
street_name	ANG MO KIO AVE 10
storey_range	10 TO 12
floor_area_sqm	44

	0
flat_model	Improved
lease_commence_date	1979
remaining_lease	61 years 04 months
resale_price	232000

Getting Rows and Columns as DataFrame

The operator `:` works similar to Python's slicing. This means that you can get some rows by slicing them. For example, you can access the first 10 rows as follows.

In [10]:

```
print(df.loc[0:10, :])
print(type(df.loc[0:10, :]))
```

```
      month        town flat_type block        street_name storey_range  \
0   2017-01  ANG MO KIO    2 ROOM   406  ANG MO KIO AVE 10     10 TO 12
1   2017-01  ANG MO KIO    3 ROOM   108   ANG MO KIO AVE 4     01 TO 03
2   2017-01  ANG MO KIO    3 ROOM   602   ANG MO KIO AVE 5     01 TO 03
3   2017-01  ANG MO KIO    3 ROOM   465  ANG MO KIO AVE 10     04 TO 06
4   2017-01  ANG MO KIO    3 ROOM   601   ANG MO KIO AVE 5     01 TO 03
5   2017-01  ANG MO KIO    3 ROOM   150   ANG MO KIO AVE 5     01 TO 03
6   2017-01  ANG MO KIO    3 ROOM   447  ANG MO KIO AVE 10     04 TO 06
7   2017-01  ANG MO KIO    3 ROOM   218   ANG MO KIO AVE 1     04 TO 06
8   2017-01  ANG MO KIO    3 ROOM   447  ANG MO KIO AVE 10     04 TO 06
9   2017-01  ANG MO KIO    3 ROOM   571   ANG MO KIO AVE 3     01 TO 03
10  2017-01  ANG MO KIO    3 ROOM   534  ANG MO KIO AVE 10     01 TO 03

    floor_area_sqm      flat_model  lease_commence_date     remaining_lease  \
0             44.0        Improved                 1979  61 years 04 months
1             67.0  New Generation                 1978  60 years 07 months
2             67.0  New Generation                 1980  62 years 05 months
3             68.0  New Generation                 1980   62 years 01 month
4             67.0  New Generation                 1980  62 years 05 months
5             68.0  New Generation                 1981           63 years
6             68.0  New Generation                 1979  61 years 06 months
7             67.0  New Generation                 1976  58 years 04 months
8             68.0  New Generation                 1979  61 years 06 months
9             67.0  New Generation                 1979  61 years 04 months
10            68.0  New Generation                 1980   62 years 01 month

    resale_price
0       232000.0
1       250000.0
2       262000.0
3       265000.0
4       265000.0
5       275000.0
6       280000.0
7       285000.0
8       285000.0
9       285000.0
10      288500.0
<class 'pandas.core.frame.DataFrame'>
```

Notice, however, that the slicing in Pandas' data frame is **inclusive** of the ending index unlike Python's slicing. The other thing to note about is that the output data type is no longer a series but rather a `DataFrame`. The reason is that now the data is two-dimensionsional.

You can specify both the rows and the columns you want as shown below.

In [11]:

```
df.loc[0:10,'month':'remaining_lease']
```

Out[11]:

month	town	flat_type	block	street_name	storey_range	floor_area_sqm	flat_model	lease_commence_date	remaining_lease

	month	town	flat_type	block	street_name	storey_range	floor_area_sqm	flat_model	lease_commence_date	remaining_lease
0	2017-01	ANG MO KIO	2 ROOM	406	ANG MO KIO AVE 10	10 TO 12	44.0	Improved	1979	61 years 04 months
1	2017-01	ANG MO KIO	3 ROOM	108	ANG MO KIO AVE 4	01 TO 03	67.0	New Generation	1978	60 years 07 months
2	2017-01	ANG MO KIO	3 ROOM	602	ANG MO KIO AVE 5	01 TO 03	67.0	New Generation	1980	62 years 05 months
3	2017-01	ANG MO KIO	3 ROOM	465	ANG MO KIO AVE 10	04 TO 06	68.0	New Generation	1980	62 years 01 month
4	2017-01	ANG MO KIO	3 ROOM	601	ANG MO KIO AVE 5	01 TO 03	67.0	New Generation	1980	62 years 05 months
5	2017-01	ANG MO KIO	3 ROOM	150	ANG MO KIO AVE 5	01 TO 03	68.0	New Generation	1981	63 years
6	2017-01	ANG MO KIO	3 ROOM	447	ANG MO KIO AVE 10	04 TO 06	68.0	New Generation	1979	61 years 06 months
7	2017-01	ANG MO KIO	3 ROOM	218	ANG MO KIO AVE 1	04 TO 06	67.0	New Generation	1976	58 years 04 months
8	2017-01	ANG MO KIO	3 ROOM	447	ANG MO KIO AVE 10	04 TO 06	68.0	New Generation	1979	61 years 06 months
9	2017-01	ANG MO KIO	3 ROOM	571	ANG MO KIO AVE 3	01 TO 03	67.0	New Generation	1979	61 years 04 months
10	2017-01	ANG MO KIO	3 ROOM	534	ANG MO KIO AVE 10	01 TO 03	68.0	New Generation	1980	62 years 01 month

If you want to select the column, you can pass on a list of columns as shown in the example below.

In [12]:

```
columns = ['town', 'block', 'resale_price']
df.loc[:, columns]
```

Out[12]:

	town	block	resale_price
0	ANG MO KIO	406	232000.0
1	ANG MO KIO	108	250000.0
2	ANG MO KIO	602	262000.0
3	ANG MO KIO	465	265000.0
4	ANG MO KIO	601	265000.0
...	...	...	...
95853	YISHUN	326	650000.0
95854	YISHUN	360	645000.0
95855	YISHUN	326	585000.0
95856	YISHUN	355	675000.0
95857	YISHUN	277	625000.0

95858 rows × 3 columns

A similar output can be obtained without `.loc`

In [13]:

```
df[columns]
```

Out[13]:

	town	block	resale_price
0	ANG MO KIO	406	232000.0
1	ANG MO KIO	108	250000.0
2	ANG MO KIO	602	262000.0
3	ANG MO KIO	465	265000.0
4	ANG MO KIO	601	265000.0
...	...	...	...
95853	YISHUN	326	650000.0
95854	YISHUN	360	645000.0
95855	YISHUN	326	585000.0
95856	YISHUN	355	675000.0
95857	YISHUN	277	625000.0

95858 rows × 3 columns

If you can combine specifying the rows and the columns as usual using `.loc`.

In [14]:

```
df.loc[0:10, columns]
```

Out[14]:

	town	block	resale_price
0	ANG MO KIO	406	232000.0
1	ANG MO KIO	108	250000.0
2	ANG MO KIO	602	262000.0
3	ANG MO KIO	465	265000.0
4	ANG MO KIO	601	265000.0
5	ANG MO KIO	150	275000.0
6	ANG MO KIO	447	280000.0
7	ANG MO KIO	218	285000.0
8	ANG MO KIO	447	285000.0
9	ANG MO KIO	571	285000.0
10	ANG MO KIO	534	288500.0

The index is not always necessarily be an integer. Pandas can take strings as the index of a data frame. But there are times, even when the index is not an integer, we still prefer to locate using the position of the rows to select. In this case, we can use `.iloc[position index, position column]`.

In [15]:

```
columns = [1, 3, -1]
df.iloc[0:10, columns]
```

Out[15]:

	town	block	resale_price
0	ANG MO KIO	406	232000.0
1	ANG MO KIO	108	250000.0
2	ANG MO KIO	602	262000.0
3	ANG MO KIO	465	265000.0
4	ANG MO KIO	601	265000.0
5	ANG MO KIO	150	275000.0
6	ANG MO KIO	447	280000.0
7	ANG MO KIO	218	285000.0
8	ANG MO KIO	447	285000.0
9	ANG MO KIO	571	285000.0

The above code gives the same data frame but it uses different input to specifies. By using `.iloc[]`, we specify the position of the index and the columns instead of the label of the index and the columns. It happens that for the index, the position numbering is exactly the same as the label.

Selecting Data Using Conditions

We can use conditions with Pandas' data frame to select particular rows and columns using either `.loc[]` or `.iloc[]`. The reason is that these methods can take in boolean arrays.

Let's see some examples below. First, let's list down the resale price by focusing on the block at a given town.

In [16]:

```
columns = ['town', 'block', 'resale_price']
df.loc[:, columns]
```

Out[16]:

	town	block	resale_price
0	ANG MO KIO	406	232000.0
1	ANG MO KIO	108	250000.0
2	ANG MO KIO	602	262000.0
3	ANG MO KIO	465	265000.0
4	ANG MO KIO	601	265000.0
...	...	...	...
95853	YISHUN	326	650000.0
95854	YISHUN	360	645000.0
95855	YISHUN	326	585000.0

	town	block	resale_price
95856	YISHUN	355	675000.0
95857	YISHUN	277	625000.0

95858 rows × 3 columns

Let's say we want to see those sales where the price is greater than \$500k. We can put in this condition in filtering the rows.

In [17]:

```
df.loc[df['resale_price'] > 500_000, columns]
```

Out[17]:

	town	block	resale_price
43	ANG MO KIO	304	518000.0
44	ANG MO KIO	646	518000.0
45	ANG MO KIO	328	560000.0
46	ANG MO KIO	588C	688000.0
47	ANG MO KIO	588D	730000.0
...	...	...	...
95853	YISHUN	326	650000.0
95854	YISHUN	360	645000.0
95855	YISHUN	326	585000.0
95856	YISHUN	355	675000.0
95857	YISHUN	277	625000.0

27233 rows × 3 columns

Note: Python ignores the underscores in between numeric literals and you can use it to make it easier to read.

Let's say if we want to find all those sales between \$500k and \\$600k only, we can use the AND operator `&` to have more than one conditions.

In [18]:

```
df.loc[(df['resale_price'] >= 500_000) & (df['resale_price'] <= 600_000), columns]
```

Out[18]:

	town	block	resale_price
43	ANG MO KIO	304	518000.0
44	ANG MO KIO	646	518000.0
45	ANG MO KIO	328	560000.0
49	ANG MO KIO	101	500000.0
110	BEDOK	185	580000.0
...	...	...	...
95849	YISHUN	504C	550000.0
95850	YISHUN	511B	600000.0
95851	YISHUN	504C	590000.0
95852	YISHUN	838	571888.0

	town	block	resale_price
95855	YISHUN	326	585000.0

13478 rows × 3 columns

Note: the parenthesis separating the two AND conditions are compulsory.

We can also specify more conditions. For example, we are only interested in ANG MO KIO area. We can have the following code.

In [19]:

```
df.loc[(df['resale_price'] >= 500_000) & (df['resale_price'] <= 600_000) &
       (df['town'] == 'ANG MO KIO'), columns]
```

Out[19]:

	town	block	resale_price
43	ANG MO KIO	304	518000.0
44	ANG MO KIO	646	518000.0
45	ANG MO KIO	328	560000.0
49	ANG MO KIO	101	500000.0
1219	ANG MO KIO	351	530000.0
...	...	...	...
94741	ANG MO KIO	545	590000.0
94742	ANG MO KIO	545	600000.0
94743	ANG MO KIO	551	520000.0
94746	ANG MO KIO	642	545000.0
94749	ANG MO KIO	353	588000.0

329 rows × 3 columns

If you are interested only in blocks 300s and 400s, you can add this conditions further.

In [20]:

```
df.loc[(df['resale_price'] >= 500_000) & (df['resale_price'] <= 600_000) &
       (df['town'] == 'ANG MO KIO') &
       (df['block'] >= '300') & (df['block'] < '500'), columns]
```

Out[20]:

	town	block	resale_price
43	ANG MO KIO	304	518000.0
45	ANG MO KIO	328	560000.0
1219	ANG MO KIO	351	530000.0
2337	ANG MO KIO	344	538000.0
2338	ANG MO KIO	329	548000.0
...	...	...	...
92327	ANG MO	353	520000.0

	town	block	resale_price
	KIO		
92332	ANG MO KIO	459	600000.0
92335	ANG MO KIO	459	500000.0
94740	ANG MO KIO	305	537000.0
94749	ANG MO KIO	353	588000.0

174 rows × 3 columns

Series and DataFrame Functions

Pandas also provides several functions that can be useful in understanding the data. In this section, we will explore some of these.

Creating DataFrame and Series

We can create a new DataFrame from other data type such as dictionary, list-like objects, or Series. For example, given a `Series`, you can convert into a `DataFrame` as shown below.

In [21]:

```
price = df['resale_price']
print(isinstance(price, pd.Series))
price_df = pd.DataFrame(price)
print(isinstance(price_df, pd.DataFrame))
price_df
```

```
True
True
```

Out[21]:

	resale_price
0	232000.0
1	250000.0
2	262000.0
3	265000.0
4	265000.0
...	...
95853	650000.0
95854	645000.0
95855	585000.0
95856	675000.0
95857	625000.0

95858 rows × 1 columns

Similarly, you can convert other data to a `Series` by using its contructor. In the example below, we create a new series from a list of integers from 2 to 100.

In [22]:

```
new_series = pd.Series(list(range(2,101)))
print(isinstance(new_series, pd.Series))
new_series
```

```
True
```

Out[22]:

```
0        2
1        3
2        4
3        5
4        6
      ...
94      96
95      97
96      98
97      99
98     100
Length: 99, dtype: int64
```

Copying

One useful function is to copy a data frame to another dataframe. We can use `df.copy()`. This function has an argument `deep` which by default is `True`. If it is true, it will do a deep copy of the Data Frame. Otherwise, it will just do a shallow copy. See documention.

In [23]:

```
df2 = df.copy()
df2
```

Out[23]:

	month	town	flat_type	block	street_name	storey_range	floor_area_sqm	flat_model	lease_commence_date	remaining_le
0	2017-01	ANG MO KIO	2 ROOM	406	ANG MO KIO AVE 10	10 TO 12	44.0	Improved	1979	61 year mo
1	2017-01	ANG MO KIO	3 ROOM	108	ANG MO KIO AVE 4	01 TO 03	67.0	New Generation	1978	60 year mo
2	2017-01	ANG MO KIO	3 ROOM	602	ANG MO KIO AVE 5	01 TO 03	67.0	New Generation	1980	62 year mo
3	2017-01	ANG MO KIO	3 ROOM	465	ANG MO KIO AVE 10	04 TO 06	68.0	New Generation	1980	62 year mo
4	2017-01	ANG MO KIO	3 ROOM	601	ANG MO KIO AVE 5	01 TO 03	67.0	New Generation	1980	62 year mo
...	...	...	...	...	...	...	...	...	...	
95853	2021-04	YISHUN	EXECUTIVE	326	YISHUN RING RD	10 TO 12	146.0	Maisonette	1988	66 year mo
95854	2021-04	YISHUN	EXECUTIVE	360	YISHUN RING RD	04 TO 06	146.0	Maisonette	1988	66 year mo
95855	2021-04	YISHUN	EXECUTIVE	326	YISHUN RING RD	10 TO 12	146.0	Maisonette	1988	66 year mo
95856	2021-04	YISHUN	EXECUTIVE	355	YISHUN RING RD	10 TO 12	146.0	Maisonette	1988	66 year mo
95857	2021-04	YISHUN	EXECUTIVE	277	YISHUN ST 22	04 TO 06	146.0	Maisonette	1985	63 year mo

95858 rows × 11 columns

Statistical Functions

We can get some descriptive statistics about the data using some of Pandas functions. For example, we can get the five point summary using `.describe()` method.

In [24]:

```
df.describe()
```

Out[24]:

	floor_area_sqm	lease_commence_date	resale_price

	floor_area_sqm	lease_commence_date	resale_price
count	95854.000000	95854.000000	9.585400e+04
mean	97.772234	1994.553934	4.467242e+05
std	24.238799	13.128913	1.552974e+05
min	31.000000	1966.000000	1.400000e+05
25%	82.000000	1984.000000	3.350000e+05
50%	95.000000	1995.000000	4.160000e+05
75%	113.000000	2004.000000	5.250000e+05
max	249.000000	2019.000000	1.258000e+06

The above code only shows a few columns because the other columns are not numbers. Pandas will only try to get the statistics of the columns that contain numeric numbers. We can also get the individual statistical functions as shown below.

In [25]:

```
print(df['resale_price'].mean())
```

```
446724.22886801313
```

In [26]:

```
print(df['resale_price'].std())
```

```
155297.43748684428
```

In [27]:

```
print(df['resale_price'].min())
```

```
140000.0
```

In [28]:

```
print(df['resale_price'].max())
```

```
1258000.0
```

In [29]:

```
print(df['resale_price'].quantile(q=0.75))
```

```
525000.0
```

You can change the way the statistics is computed. Currently, the statistics is calculated over all the rows in the vertical dimension. This is what is considered as `axis=0` in Pandas. You can change it to compute over all the columns by specifying `axis=1`.

In [30]:

```
df.mean(axis=1)
```

Out[30]:

```
0          78007.666667
1          84015.000000
2          88015.666667
3          89016.000000
4          89015.666667
              ...
95853     217378.000000
95854     215711.333333
95855     195711.333333
95856     225711.333333
95857     209043.666667
```

```
Length: 95858, dtype: float64
```

Again, Pandas only computes the mean from the numeric data across the columns.

Transposing Data Frame

You can also change the rows into the column and the column into the rows. For example, previously we have this data frame we created from a `Series` when extracting row 0.

In [31]:

```
df_row0
```

Out[31]:

	0
month	2017-01
town	ANG MO KIO
flat_type	2 ROOM
block	406
street_name	ANG MO KIO AVE 10
storey_range	10 TO 12
floor_area_sqm	44
flat_model	Improved
lease_commence_date	1979
remaining_lease	61 years 04 months
resale_price	232000

In the above code, the column is row 0 and the rows are the different column names. You can transpose the data using the `.T` property.

In [32]:

```
df_row0_transposed = df_row0.T
df_row0_transposed
```

Out[32]:

	month	town	flat_type	block	street_name	storey_range	floor_area_sqm	flat_model	lease_commence_date	remaining_lease	resa
0	2017-01	ANG MO KIO	2 ROOM	406	ANG MO KIO AVE 10	10 TO 12	44	Improved	1979	61 years 04 months	

Vector Operations

One useful function in Pandas is `.apply()` (see documentation) where we can apply some function to all the data in the column or row or Series in a vectorized manner. In this way, we need not iterate or loop the data one at a time to apply this computation.

For example, if we want to create a column for resale price in terms of $1000, we can use the `.apply()` method by dividing the `resale_price` column with 1000.

In [33]:

```
def divide_by_1000(data):
    return data / 1000

df['resale_price_in1000'] = df['resale_price'].apply(divide_by_1000)
df['resale_price_in1000']
```

Out[33]:

```
0         232.0
1         250.0
2         262.0
3         265.0
4         265.0
          ...
95853     650.0
95854     645.0
95855     585.0
95856     675.0
95857     625.0
Name: resale_price_in1000, Length: 95858, dtype: float64
```

The method `.apply()` takes in a function that will be processed for every data in that Series. Instead of creating a named function, we can make use of Python's lambda function to do the same.

In [34]:

```
df['resale_price_in1000'] = df['resale_price'].apply(lambda data: data/1000)
df['resale_price_in1000']
```

Out[34]:

```
0         232.0
1         250.0
2         262.0
3         265.0
4         265.0
          ...
95853     650.0
95854     645.0
95855     585.0
95856     675.0
95857     625.0
Name: resale_price_in1000, Length: 95858, dtype: float64
```

Notice that the argument in `divide_by_1000()` becomes the first token after the keyword `lambda`. The return value of the function is provided after the colon, i.e. `:`.

You can use this to process and create any other kind of data. For example, we can create a new categorical column called "Pricey" and set any sales above \$500k is considered as pricey otherwise is not. If it is pricey, we will label it as 1, otherwise, as 0.

In [35]:

```
df['pricey'] = df['resale_price_in1000'].apply(lambda price: 1 if price > 500 else 0 )
df[['resale_price_in1000', 'pricey']]
```

Out[35]:

	resale_price_in1000	pricey
0	232.0	0
1	250.0	0
2	262.0	0
3	265.0	0
4	265.0	0
...	...	...
95853	650.0	1
95854	645.0	1
95855	585.0	1
95856	675.0	1
95857	625.0	1

95858 rows × 2 columns

In the above function, we use the if *expression* to specify the return value for the lambda function. It follows the following format:

```
expression_if_true if condition else expression_if_false
```

There are many other Pandas functions and methods. It is recommended that you look into the documentation for further references.

Reference

- Pandas User Guide
- Pandas API Reference

Normalization

Many times, we will need to normalize the data, both the features and the target. The reason is that each column in the dataset may have different scales. For example, the column `floor_area_sqm` is in between 33 to 249 while `lease_commense_date` is actually in a range between 1966 and 2019. See below statistics for these two columns.

In [36]:

```
display(df['floor_area_sqm'].describe())
display(df['lease_commence_date'].describe())
```

```
count    95858.000000
mean        97.772234
std         24.238799
min         31.000000
25%         82.000000
50%         95.000000
75%        113.000000
max        249.000000
Name: floor_area_sqm, dtype: float64

count    95858.000000
mean      1994.553934
std         13.128913
min       1966.000000
25%       1984.000000
50%       1995.000000
75%       2004.000000
max       2019.000000
Name: lease_commence_date, dtype: float64
```

As we will see later in subsequent weeks, we usually need to normalize the data before doing any training for our machine learning model. There are two common normalization:

- z normalization
- minmax normalization

You will work on the functions to calculate these normalization in your problem sets.

Z Normalization

This is also called as standardization. In this tranformation, we move the mean of the data distribution to 0 and its standard deviation to 1. The equation is given as follows.
$$normalized = \frac{data - \mu}{\sigma}$$

Min-Max Normalization

In this transformation, we scale the data in such a way that the maximum value in the distribution is 1 and its minimum value is 0. We can scale it using the following equation.

$$normalized = \frac{data - min}{max - min}$$

Splitting Dataset

One common pre-processing operations that we normally do in machine learning is to split the data into:

- **training** dataset
- **test** dataset

The idea of splitting the dataset is simply because we should **NOT** use the same data to verify the model which we train. Let's illustrate this using our HDB resale price dataset. We have this HDB resale price dataset with 95858 entries. In our machine learning process, we would like to use this data to do the following:

1. Train the model using the dataset
2. Verify the accuracy of the model

If we only have one dataset, we cannot use the same data to verify the accuracy with the ones we use to train the model. This bias would obviously create high accuracy. The analogy is like when a teacher giving exactly the same question during the exam as the ones during the practice session.

To overcome this, we should split the data into two. One set is used to train the model while the other one is used to verify the model. Coming back to the analogy of the teacher giving the exam, if the teacher has a bank of questions, he or she should separate the questions into two. Some questions can be used for practice before the exam, while the rest can be used as exam questions.

If we illustrate this using our HDB resale price dataset, this means that we have to split the table into two. Out of 95868 entries, we will take some entries as out training dataset and leave the rest for out test dataset. The common proportion is either 70% or 80% for the training dataset and leave the other 30% or 20% for the test dataset.

One important note is that the split must be done **randomly**. This is to avoid systematic bias in the split of the dataset. For example, one may enter the data according to the flat type and so flat with smaller rooms and smaller floor area will be on the top rows and those with the larger flats will be somewhere at the end of the rows. If we do not split the data randomly, we may not have any larger flats in our training set and only use the smaller flats. Similarly it can happen with any other column such as the block or the town area.

There are times in machine learning, we need to experiment with different parameters and find the optimum parameters. In these cases, the dataset is usually split into three:

- **training** dataset, which is used to build the model
- **validation** dataset, which is used to evaluate the model for various parameters and to choose the optimum parameter
- **test** dataset, which is used to evaluate the model built with the optimum parameter found previously

You will see some of these application in the subsequent weeks.