Visualization with Matplotlib

Introduction

- Making plots and static or interactive visualizations is one of the most important tasks in data analysis. It may be a part of the exploratory process; for example, helping identify outliers, needed data transformations, or coming up with ideas for models.
- Matplotlib is the most extensively used library of python for data visualization due to it's high flexibility and extensive functionality that it provides.

Table of Contents

1. Setting up
    - Importing matplotlib
    - Matplotlib for Jupyter notebook
    - Dataset
    - Documentation
2. Matplotlib basics
    - Make a simple plot
    - Labels, and Legends
    - Size, Colors, Markers, and Line Styles
    - Figures and subplots
3. Line Chart
4. Bar Chart
5. Histogram
6. Box plot
7. Violin plot
8. Scatter plot
9. Bubble plot

1. Setting up

Importing matplotlib

Just as we use the `np` shorthand for NumPy and the `pd` shorthand for Pandas, we will use standard shorthands for Matplotlib import:

```
import matplotlib.pyplot as plt
```

We import the **pyplot** interface of matplotlib with a shorthand of `plt` and we will be using it like this in the entire notebook.

Matplotlib for Jupyter notebook

You can directly use matplotlib with this notebook to create different visualizations in the notebook itself. In order to do that, the following command is used:

```
%matplotlib inline
```

Documentation

All the functions covered in this notebook and their detail description can be found in the official matplotlib documentation.

In [1]:

```
# importing required libraries
import numpy as np
import pandas as pd

# importing matplotlib
import matplotlib.pyplot as plt

# display plots in the notebook itself
%matplotlib inline
```

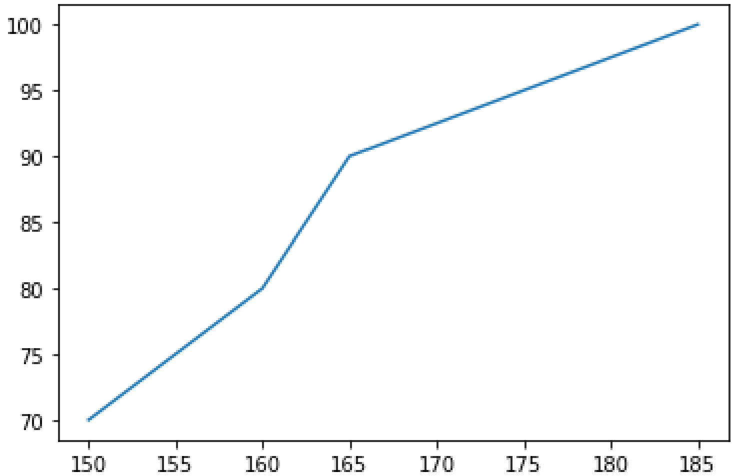
2. `Matplotlib basics`

Make a simple plot

Let's create a basic plot to start working with!

In [2]:

```
# list of height
height = [150,160,165,185]
# list of weight
weight = [70, 80, 90, 100]

# draw the plot
plt.plot(height, weight);
```



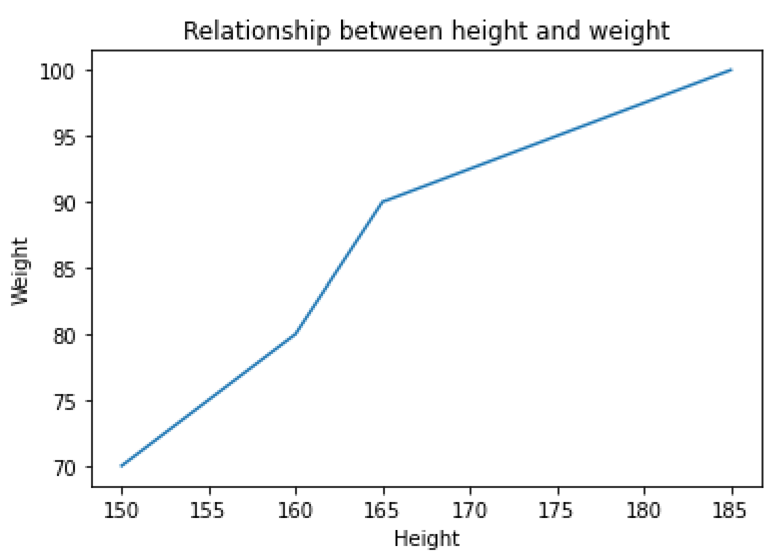
We pass two lists as our input arguments to **plot()** method and invoke the required plot. Here note that the first array appears on the x-axis and second array appears on the y-axis of the plot.

Title, Labels, and Legends

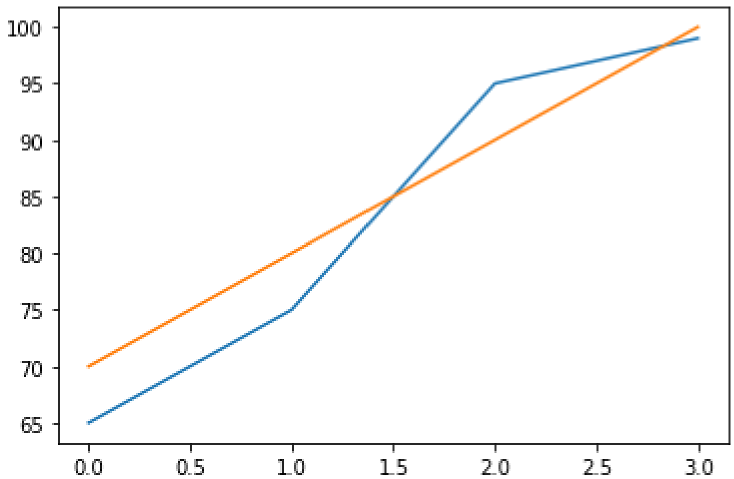
- Now that our first plot is ready, let us add the title, and name x-axis and y-axis using methods title(), xlabel() and ylabel() respectively.

In [3]:
```
# draw the plot
plt.plot(height,weight)
# add title
plt.title("Relationship between height and weight")
# label x axis
plt.xlabel("Height")
# label y axis
plt.ylabel("Weight")
```

Out[3]:
```
Text(0, 0.5, 'Weight')
```

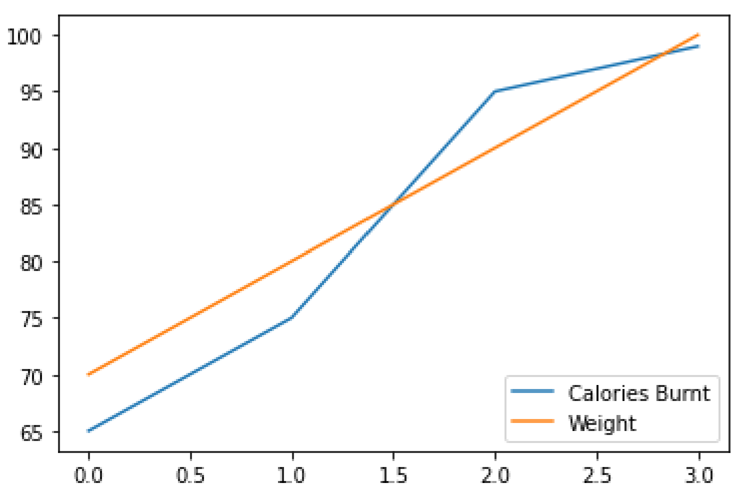


In [4]:
```python
# list of calories_burnt
calories_burnt = [65, 75, 95, 99]

# draw the plot for calories burnt
plt.plot(calories_burnt)
# draw the plot for weight
plt.plot(weight);
```



- Adding **legends** is also simple in matplotlib, you can use the `legend()` which takes **labels** and **loc** as label names and location of legend in the figure as paremeters.

In [5]:
```python
# draw the plot for calories burnt
plt.plot(calories_burnt)
# draw the plot for weight
plt.plot(weight)

# add legend in the lower right part of the figure
plt.legend(labels=['Calories Burnt', 'Weight'], loc='lower right');
```

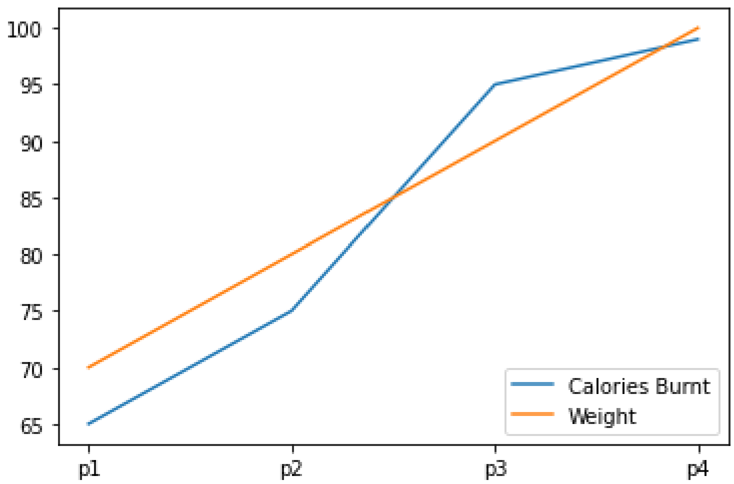


- Notice that in the previous plot, we are not able to understand that each of these values belong to different persons.
- Look at the X axis, can we add labels to show that each belong to different persons?

- The labeled values on any axis is known as a **tick**.
- You can use the `xticks` to change both the location of each tick and it's label. Let's see this in an example

---

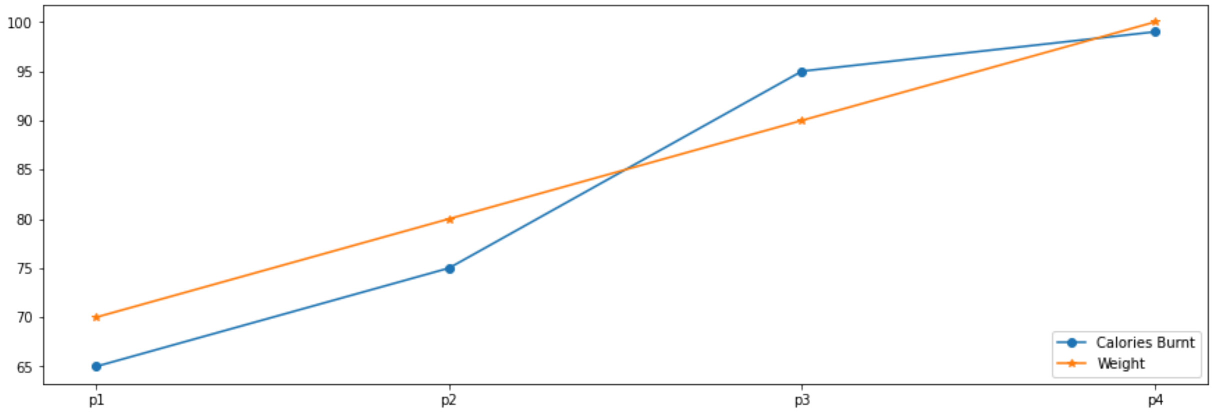
In [6]:
```
# draw the plot
plt.plot(calories_burnt)
plt.plot(weight)

# add legend in the lower right part of the figure
plt.legend(labels=['Calories Burnt', 'Weight'], loc='lower right')

# set labels for each of these persons
plt.xticks(ticks=[0,1,2,3], labels=['p1', 'p2', 'p3', 'p4']);
```



Size, Colors, Markers and Line styles

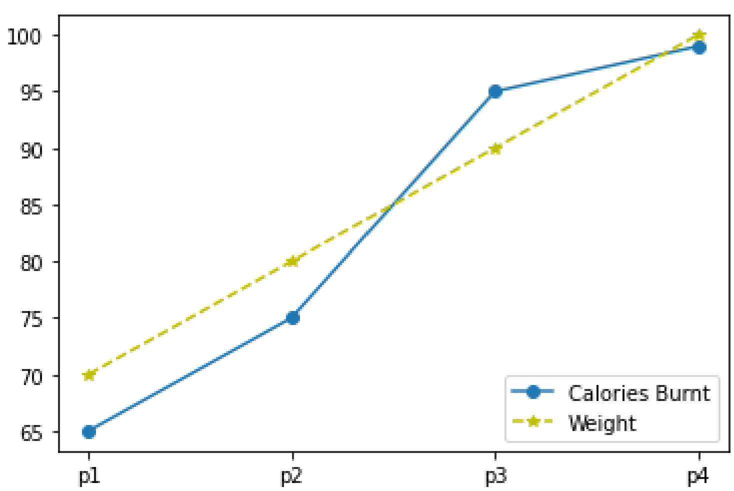
- You can also specify the size of the figure using method `figure()` and passing the values as a tuple of the length of rows and columns to the argument figsize.
- The values of length are considered to be in **inches**.

In [7]:
```
# figure size in inches
plt.figure(figsize=(15,5))

# draw the plot
plt.plot(calories_burnt, marker= 'o')
plt.plot(weight, marker = '*')

# add legend in the lower right part of the figure
plt.legend(labels=['Calories Burnt', 'Weight'], loc='lower right')

# set labels for each of these persons
plt.xticks(ticks=[0,1,2,3], labels=['p1', 'p2', 'p3', 'p4']);
```



- With every X and Y argument, you can also pass an optional third argument in the form of a string which indicates the colour and line type of the plot.
- The default format is `b-` which means a **solid blue line**. In the figure below we use `go` which means **green circles**. Likewise, we can make many such combinations to format our plot.

In [8]:
```
# draw the plot
plt.plot(calories_burnt,marker= 'o')
plt.plot(weight,'y--', marker='*')

# add legend in the lower right part of the figure
plt.legend(labels=['Calories Burnt', 'Weight'], loc='lower right')

# set labels for each of these persons
plt.xticks(ticks=[0,1,2,3], labels=['p1', 'p2', 'p3', 'p4']);
```



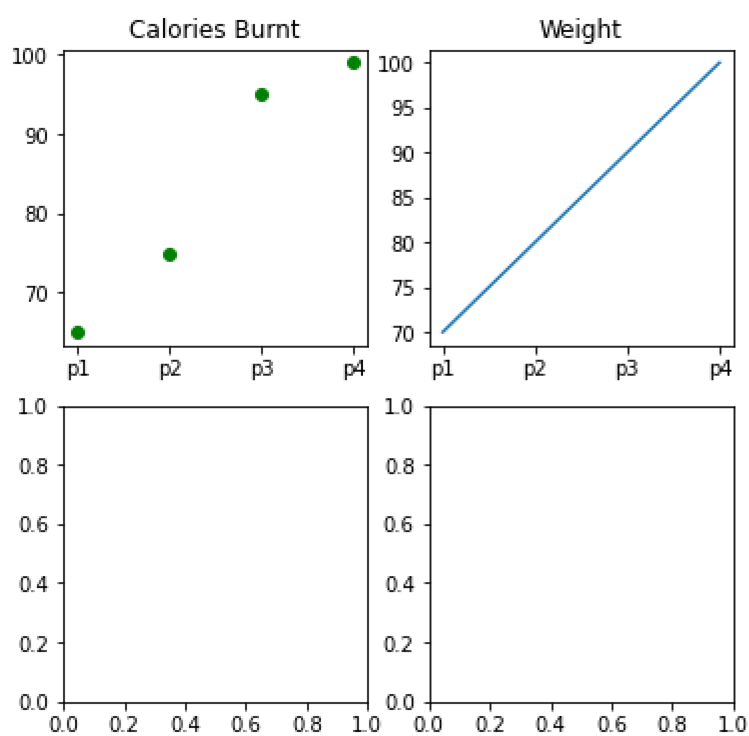
- We can also plot multiple sets of data by passing in multiple sets of arguments of X and Y axis in the `plot()` method as shown.

Figure and subplots

- We can use `subplots()` method to add more than one plots in one figure.
- The `subplots()` method takes two arguments: they are **nrows, ncols**. They indicate the number of rows, number of columns respectively.
- This method creates two objects: **figure** and **axes** which we store in variables `fig` and `ax`.
- You plot each figure by specifying its position using row index and column index. Let's have a look at the below example:

In [9]:

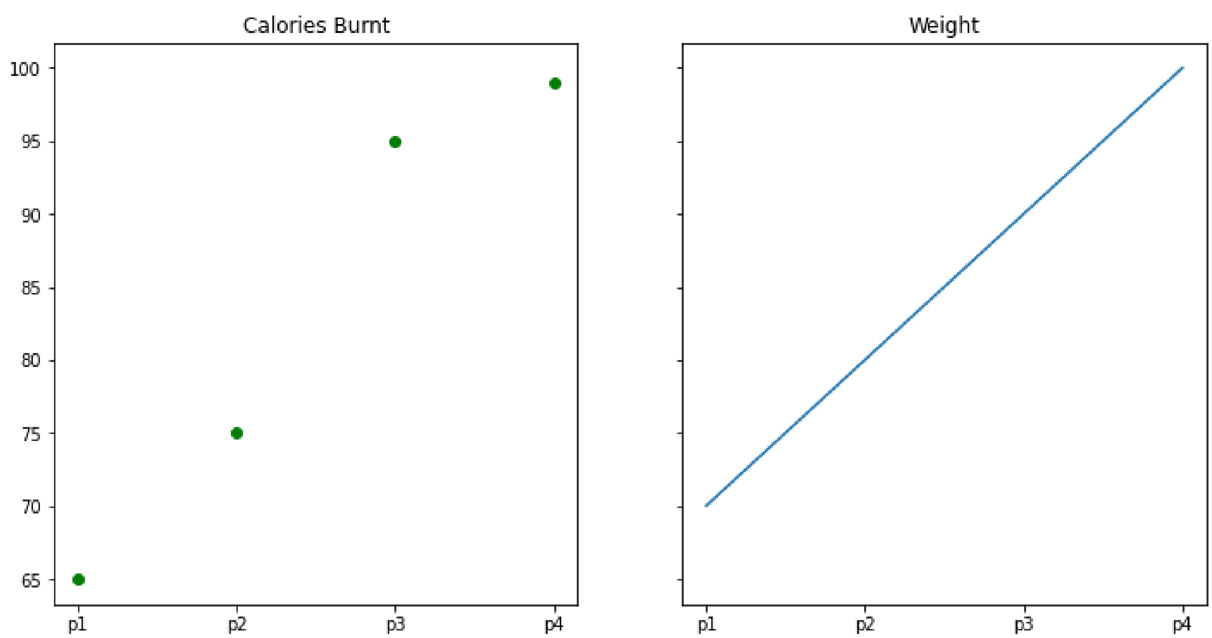
```
# create 2 plots
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(6,6))

# plot on 0 row and 0 column
ax[0,0].plot(calories_burnt,'go')

# plot on 0 row and 1 column
ax[0,1].plot(weight)

# set titles for subplots
ax[0,0].set_title("Calories Burnt")
ax[0,1].set_title("Weight")

# set ticks for each of these persons
ax[0,0].set_xticks(ticks=[0,1,2,3]);
ax[0,1].set_xticks(ticks=[0,1,2,3]);

# set labels for each of these persons
ax[0,0].set_xticklabels(labels=['p1', 'p2', 'p3', 'p4']);
ax[0,1].set_xticklabels(labels=['p1', 'p2', 'p3', 'p4']);
```



- Notice that in the above figure we have two empty plots, that is because we created 4 subplots (2 rows and 2 columns).
- As a data scientist, there will be times when you need to have a common axis for all your subplots. You can do this by using the **sharex** and **sharey** paremeters of `subplot()`.

In [10]:

```
# create 2 plots
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,6), sharex=True, shar


# plot on 0 row and 0 column
ax[0].plot(calories_burnt,'go')

# plot on 0 row and 1 column
ax[1].plot(weight)

# set titles for subplots
ax[0].set_title("Calories Burnt")
ax[1].set_title("Weight")

# set ticks for each of these persons
ax[0].set_xticks(ticks=[0,1,2,3]);
ax[1].set_xticks(ticks=[0,1,2,3]);

# set labels for each of these persons
ax[0].set_xticklabels(labels=['p1', 'p2', 'p3', 'p4']);
ax[1].set_xticklabels(labels=['p1', 'p2', 'p3', 'p4']);
```



- Notice in the above plot, now both x and y axes are only labelled once for each of the outer plots. This is because the inner plots "share" both the axes.

- Also, there are only **two plots** since we decreased the number of rows to 1 and columns to 2 in the `subplot()`.
- You can learn more about subplots here.

Load dataset

Let's load a dataset and have a look at first 5 rows.

In [11]:
```
# read the dataset
data_BM = pd.read_csv('../input/datase/bigmart_data.csv')
# drop the null values
data_BM = data_BM.dropna(how="any")
# view the top results
data_BM.head()
```

Out[11]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MR
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.809
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.269
2	FDN15	17.500	Low Fat	0.016760	Meat	141.618
4	NCD19	8.930	Low Fat	0.000000	Household	53.861
5	FDP36	10.395	Regular	0.000000	Baking Goods	51.400

---

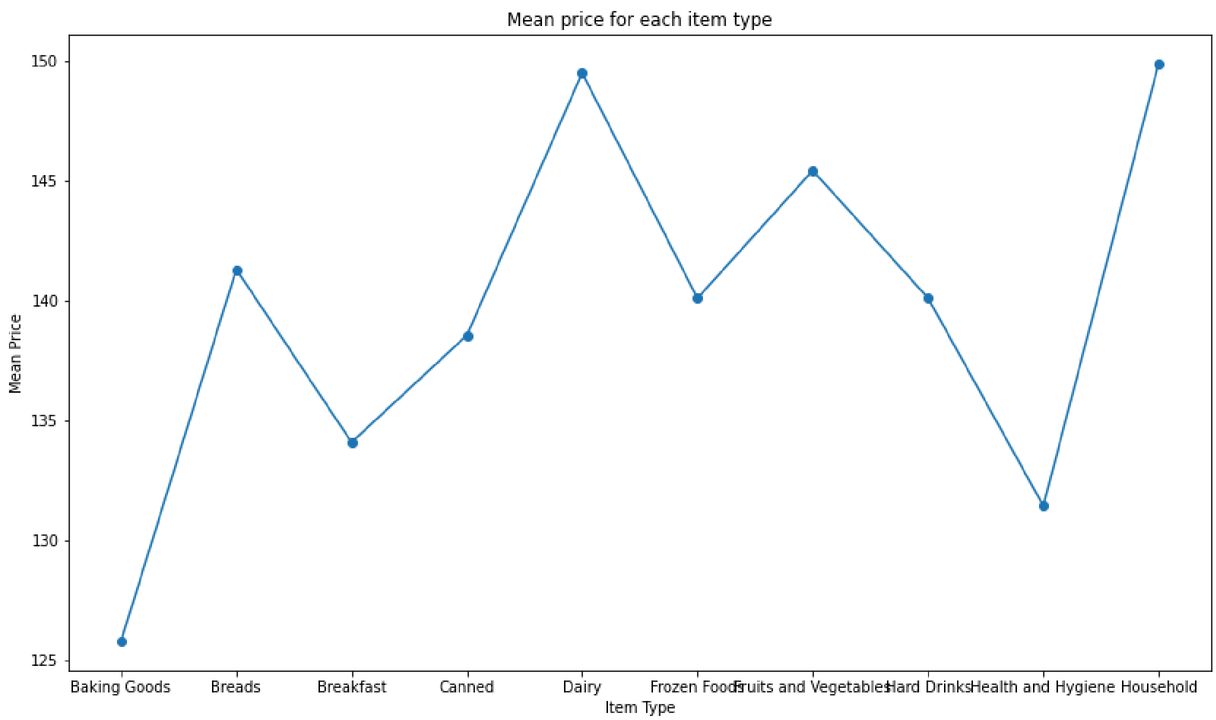
3. Line Chart

- We will create a line chart to denote the **mean price per item**. Let's have a look at the code.
- With some datasets, you may want to understand changes in one variable as a function of time, or a similarly continuous variable.
- In matplotlib, **line chart** is the default plot when using the `plot()`.

In [12]:
```
price_by_item = data_BM.groupby('Outlet_Establishment_Year').Item_Outlet_S
price_by_item
```

Out[12]:
```
Outlet_Establishment_Year
1987    2298.995256
1997    2277.844267
1999    2348.354635
2004    2438.841866
2009    1995.498739
Name: Item_Outlet_Sales, dtype: float64
```

In [13]:
```python
# mean price based on item type
price_by_item = data_BM.groupby('Item_Type').Item_MRP.mean()[:10]

x = price_by_item.index.tolist()
y = price_by_item.values.tolist()

# set figure size
plt.figure(figsize=(14, 8))

# set title
plt.title('Mean price for each item type')

# set axis labels
plt.xlabel('Item Type')
plt.ylabel('Mean Price')

# set xticks
plt.xticks(labels=x, ticks=np.arange(len(x)))

plt.plot(x, y, marker = 'o');
```



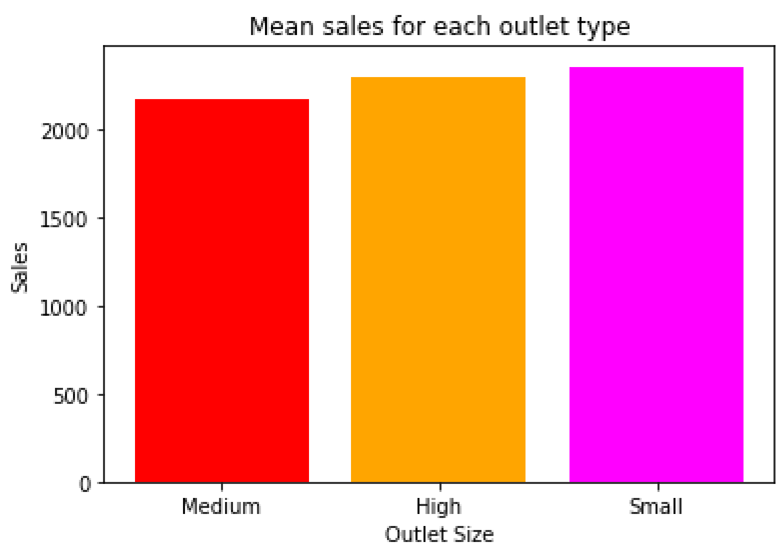
---

4. Bar Chart

- Suppose we want to have a look at **what is the mean sales for each outlet type?**
- A bar chart is another simple type of visualization that is used for categorical variables.
- You can use `plt.bar()` instead of `plt.plot()` to create a bar chart.

In [14]:

```
# sales by outlet size
sales_by_outlet_size = data_BM.groupby('Outlet_Size').Item_Outlet_Sales.me

# sort by sales
sales_by_outlet_size.sort_values(inplace=True)

x = sales_by_outlet_size.index.tolist()
y = sales_by_outlet_size.values.tolist()

# set axis labels
plt.xlabel('Outlet Size')
plt.ylabel('Sales')

# set title
plt.title('Mean sales for each outlet type')

# set xticks
plt.xticks(labels=x, ticks=np.arange(len(x)))

plt.bar(x, y, color=['red', 'orange', 'magenta']);
```



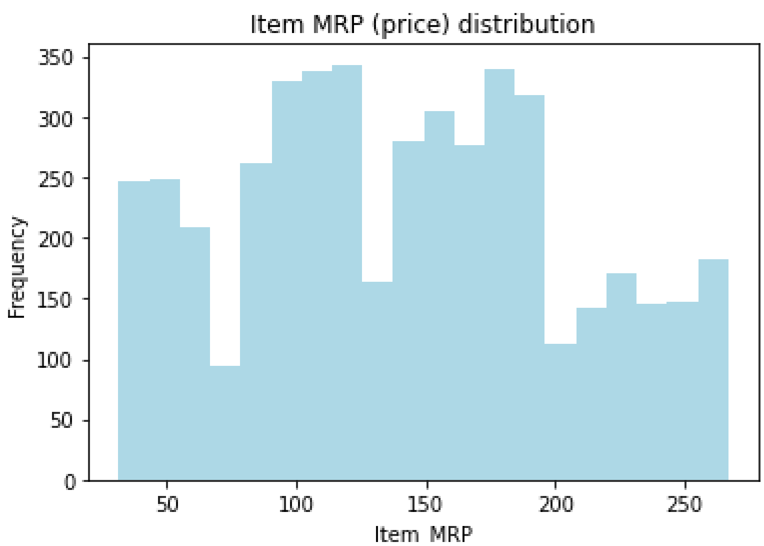
---

5. Histogram

- **Distribution of Item price**
- Histograms are a very common type of plots when we are looking at data like height and weight, stock prices, waiting time for a customer, etc which are continuous in nature.

- Histogram's data is plotted within a range against its frequency.
- Histograms are very commonly occurring graphs in probability and statistics and form the basis for various distributions like the normal -distribution, t-distribution, etc.
- You can use `plt.hist()` to draw a histogram. It provides many parameters to adjust the plot, you can explore more here.

In [15]:
```
# title
plt.title('Item MRP (price) distribution')

# xlabel
plt.xlabel('Item_MRP')

# ylabel
plt.ylabel('Frequency')

# plot histogram
plt.hist(data_BM['Item_MRP'], bins=20, color='lightblue');
```



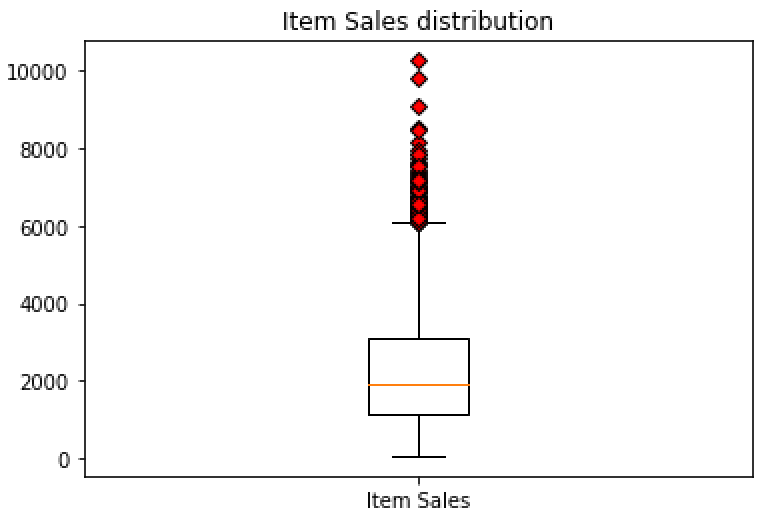
---

6. Box Plots

- **Distribution of sales**
- Box plot shows the three quartile values of the distribution along with extreme values.
- The "whiskers" extend to points that lie within 1.5 IQRs of the lower and upper quartile, and then observations that fall outside this range are displayed independently.
- This means that each value in the boxplot corresponds to an actual observation in the data.
- Let's try to visualize the distributio of `Item_Outlet_Sales` of items.

In [16]:

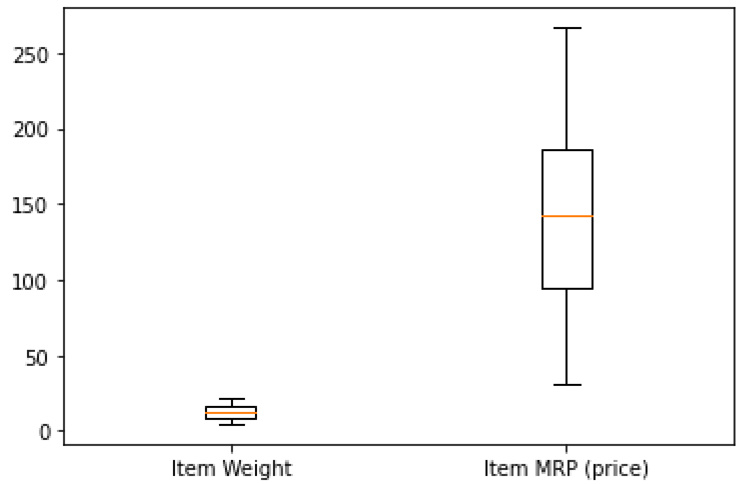
```
data = data_BM[['Item_Outlet_Sales']]

# create outlier point shape
red_diamond = dict(markerfacecolor='r', marker='D')

# set title
plt.title('Item Sales distribution')

# make the boxplot
plt.boxplot(data.values, labels=['Item Sales'], flierprops=red_diamond);
```



- You can also create multiple boxplots for different columns of your dataset.
- In order to plot multiple boxplots, you can use the same `subplots()` that we saw earlier.
- Let's see Item_Weight, Item_MRP distribution together

In [17]:

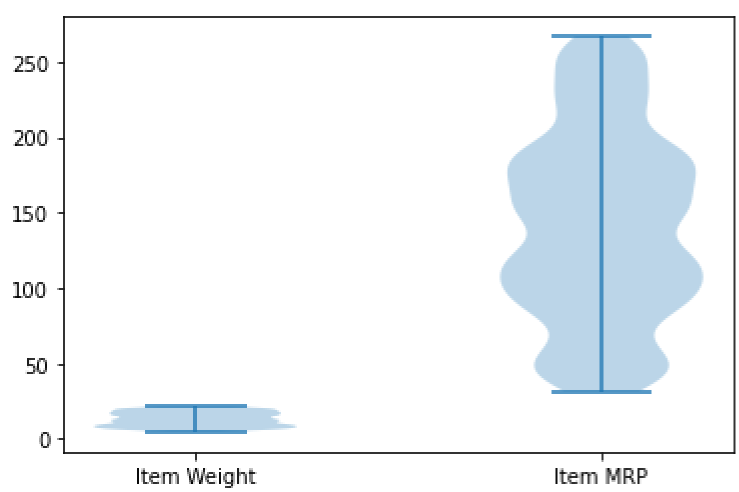
```
data = data_BM[['Item_Weight', 'Item_MRP']]

# create outlier point shape
red_diamond = dict(markerfacecolor='r', marker='D')

# generate subplots
fig, ax = plt.subplots()

# make the boxplot
plt.boxplot(data.values, labels=['Item Weight', 'Item MRP (price)'], flier
```



7. Violin Plots

- **Density distribution of Item weights and Item price**

In [18]:
```
data = data_BM[['Item_Weight', 'Item_MRP']]

# generate subplots
fig, ax = plt.subplots()

# add labels to x axis
plt.xticks(ticks=[1,2], labels=['Item Weight', 'Item MRP'])

# make the violinplot
plt.violinplot(data.values);
```



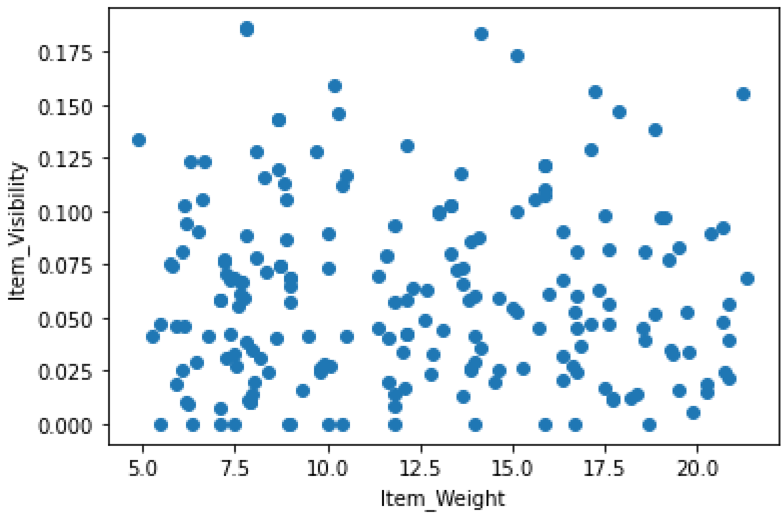
8. Scatter Plots

- **Relative distribution of item weight and it's visibility**

- It depicts the distribution of two variables using a cloud of points, where each point represents an observation in the dataset.
- This depiction allows the eye to infer a substantial amount of information about whether there is any meaningful relationship between them.

NOTE : Here, we are going to use only a subset of the data for the plots.

In [19]:
```
# set label of axes
plt.xlabel('Item_Weight')
plt.ylabel('Item_Visibility')

# plot
plt.scatter(data_BM["Item_Weight"][:200], data_BM["Item_Visibility"][:200]
```



---

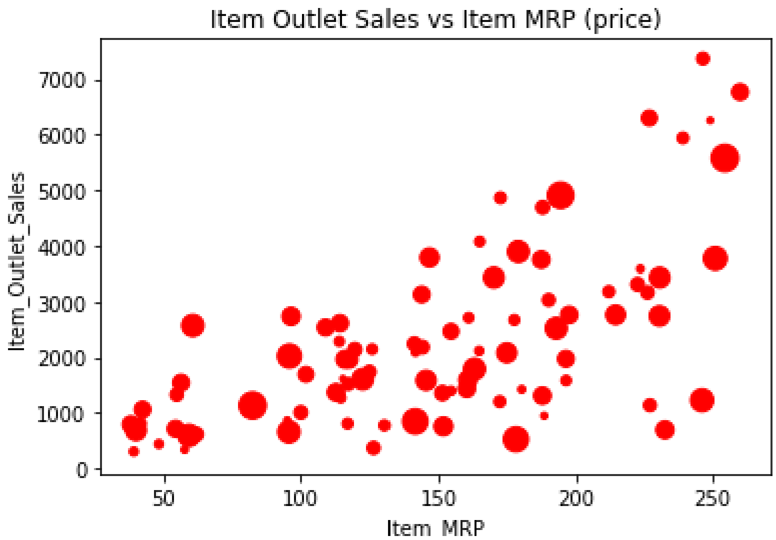
9. Bubble Plots

- **Relative distribution of sales, item price and item visibility**
- Let's make a scatter plot of Item_Outlet_Sales and Item_MRP and make the **size** of bubbles by the column Item_Visibility.
- Bubble plots let you understand the interdependent relations among 3 variables.

Note that we are only using a subset of data for the plots.

In [20]:
```
# set label of axes
plt.xlabel('Item_MRP')
plt.ylabel('Item_Outlet_Sales')

# set title
plt.title('Item Outlet Sales vs Item MRP (price)')
```

```
# plot
plt.scatter(data_BM["Item_MRP"][:100], data_BM["Item_Outlet_Sales"][:100],
```



In [ ]:

In [ ]: