AZEEZ TRYING TO LEARN AND TEACH PYTHON

CHAPTER 1: Modules, Comment, and Pip

A module is a file containing code written by a programmer for others to import and/or used in their programs.

Pip: is the package manager for Python. You can use Pip to install a module on your system.

There are 2 types of modules in Python,namely: Built-in and the External modules

Comments: are used to write something which the program is expected to omit during execution. There 2 types of comments: Single line comment, written with #, and Multi-line comments, written with """-"""

CHAPTER 2: Variables and DataTypes

A variable is the name given to a memory location in a program. Variable is otherwise defined "container" used to store a value.

For example: a = 11 b = 22.5 c = "Azeez

Data Types

Primarilly, we have 5 different data types in python.

1. Integers
2. Floating point number
3. Strings
4. Booleans
5. None Python automatically identifies the type of the data.

Defining variable

> A variables must not have alphabets, underscores, and digits A variable name cannot begin with anything other than alphabet or underscore A variable name must not have white space in-between. Examples: Azeez, Nigeria, Orator_1, _Toko_Koutogi etc.

Operators in Python

The common ones are;

1. Arithmetic operators such as +, -, *, / etc.
2. Assignment operators such as =,+=, -=, *=, /= etc.
3. Comparison operators such as ==, >, <, >=, <=, != etc.
4. Logical operators; and, or, not

type() function and Typecasting

type function is used to find the data type of a given variable in Python.

In [4]:

```
N = 100
type(N)
```

Out[4]:

```
int
```

In [35]:

```
N = str(N)
type(N)
```

Out[35]:

```
str
```

In [5]:

```
M = "12"
type(M)
```

Out[5]:

```
str
```

In [36]:

```
M = int(M)
type(M)
```

Out[36]:

```
int
```

In [9]:

```
L = 12.1
type(L)
```

Out[9]:

```
float
```

A number can be converted into a string and vice versa (whenever possible). There are many functions to convert one datatype into another.

In [ ]:

```
str(12) means "12" #integer to string
```

In [ ]:

```
int("12") means 12 #string to integer
```

In [8]:

```
float(32) means to 32.0 #integer to float
etc.
```

In [84]:

```
value = "10"
type(value)
```

Out[84]:

```
str
```

In [33]:

```
number = int(value)
type(number)
```

Out[33]:

```
int
```

input() function

This function allows the user to take input from the keyboard as a string. name = input("Please enter your name ") It is essential to note that the output of input function is always a string, even if the number is entered.

In [44]:

```
name = input('What is your name?\n')
print('Hello ' + name)
```

```
What is your name?
Azeez
Hello Azeez
```

CHAPTER 3:

Strings

String is a text or sequence of characters. We write strings in quotes. It of 3 types: Single quoted string Double quoted string Triple quoted string

In [45]:

```
a = 'Yahya'
```

In [11]:

```
b = "Bouchra"
b
```

Out[11]:

```
'Bouchra'
```

In [37]:

```
c = '''Khawla'''
c
```

Out[37]:

```
'Khawla'
```

In [38]:

```
D ="""Azeez"""
D
```

Out[38]:

```
'"Azeez'
```

String slicing

A string in Python can be sliced for getting a part of the string. e.g. country 0123456 length = 7 The index in a string starts from 0 to (length -1) in Python. In order to slice a sting, we use the following syntx: sl = name[ind_start : ind_end] first index is include while the last is excluded.

In [18]:

```
country = 'Nigeria'
country[0:3] # country[0:3] returns 'Nig'
```

Out[18]:

```
'Nig'
```

In [265]:

```
country[1:3] # returns 'ig'
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
<ipython-input-265-386e057e6f56> in <module>
----> 1 country[1:3] # returns 'ig'

NameError: name 'country' is not defined
```

Negative indices

negative indices can also be used. -1 coresponds to the (length -1) index, -2 to (length -2)

In [20]:

```
country[-1] #the last letter in 'Nigeria'
```

Out[20]:

```
'a'
```

In [21]:

```
country[-2] #the second to the last letter
```

Out[21]:

```
'i'
```

Slicing with skip value

We can provide a skip vlue as a part of our slice like this:

In [29]:

```
st = 'Khawla'
st[:2:2]
```

Out[29]:

```
'K'
```

In [26]:

```
nation = "Morocco"
nation[1:2:2]
```

Out[26]:

```
'o'
```

In [30]:

```
wow = "amazing"
nation[1:6:2]
```

Out[30]:

```
'ooc'
```

In [85]:

```
text = "Hello World!"
print (text[: 5 ])
print (text[ 6 : 11 ]) #The first slice we print is “Hello” and the second one is “Worl
d”.
```

```
Hello
World
```

In [51]:

```
# Another thing we can do is to iterate over strings with for loops.
text = "Hello World!"
for x in text:
    print(x) #In this example, we print the individual characters one after the other.
```

```
H
e
l
l
o

W
o
r
l
d
!
```

Escape characters

In strings we can use a lot of different escape characters . Escape character comprises of more than one characters but represents one character when used within the strings. They are non-printable characters like tab or new line . They are all initiated by a backslash, which is the reason why we need to use double backslashes for file paths.

ESCAPE CHARATCERS \b Backspace \n New Line \s Space \t Tab

In [100]:

```
life = "\tLife is a combination of up and down.\nTo survive in life, \b you must be pre
pared for trials.There are times of trials, but there are many times of happiness, if y
ou can hold on!"
print(life)
```

```
        Life is a combination of up and down.
To survive in life,  you must be prepared for trials.There are times of t
rials, but there are many times of happiness, if you can hold on!
```

String formating

When we have a text which shall include the values of variables, we can use the % operator and placeholders, in order to insert our values.

The following shows which placeholders are needed for which data types.

Place holder

%c Character %s String %d or %i Integer %f Float %e Exponential Notation

In [218]:

```
name, age = "Adam" , 25
print ( "%s is my name!" % name)
print ( "I am %d years old!" % age)
#Notice that we used different placeholders for different data types. We use %s for str
ings and %d for integers.
```

```
Adam is my name!
I am 25 years old!
```

If you want to do it more general without specifying data types, you can use the format function.

In [55]:

```
name, age = "John" , 25
print ( "My name is {} and I am {} years old" .format(name, age))
#Here we use curly brackets as placeholders and insert the values afterwards using the
 format function.
```

```
My name is John and I am 25 years old
```

PYTHON String functions

Even though strings are just texts or sequences of characters, we can apply a lot of functions and operations on them. We will focus on the most essential, most interesting and most important of these functions. The ones that you might need in the near future.

len() function

In [1]:

```
my_str = 'This is just a string'
len(my_str) # This returns the lenght of the string
```

Out[1]:

```
21
```

In [ ]:

```
help(set) # to get info about any function e.g set
```

endswith function

In [89]:

```
my_str.endswith('ng') # This verifies whether the string ends with 'ng' or not
```

Out[89]:

```
True
```

Case Manipulating Functions

We have five different case manipulating string functions in Python. Let's have a look at them.

string.lower() string.upper() string.title() string.capitalize() string.swapcase()

In [59]:

```
string = 'We are proudly the students of UM6P'
string.lower() #Converts all letters to lowercase
```

Out[59]:

```
'we are proudly the students of um6p'
```

In [ ]:

```

```

In [60]:

```
string.upper() # Converts all letters to uppercase
```

Out[60]:

```
'WE ARE PROUDLY THE STUDENTS OF UM6P'
```

In [61]:

```
string.title() # Converts all letters to titlecase
```

Out[61]:

```
'We Are Proudly The Students Of Um6P'
```

In [62]:

```
string.capitalize() # Converts first letter to uppercase
```

Out[62]:

```
'We are proudly the students of um6p'
```

In [64]:

```
string.swapcase() # Swaps the case of all letters
```

Out[64]:

```
'wE ARE PROUDLY THE STUDENTS OF um6p'
```

Count Function

If you want to count how many times a specific string occurs in another string, you can use the count function.

In [65]:

```
text = "I like you and you like me!"
print (text.count( "you" ))
#In this case, the number two will get printed, since the string “you” occurs two time
s.
```

```
2
```

Find function

In order to find the first occurrence of a certain string in another string, we use the find function.

In [67]:

```
text = "I like you and you like me!"
print (text.find( "you" ))
# Here the result is 7 because the first occurrence of “you” is at the index 7 .
```

```
7
```

Join Function

With the join function we can join a sequence to a string and separate each element by this particular string.

In [16]:

```
names = [ "Mike" , "John" , "Anna" ]
sep = "-"
print(sep.join(names))
# you can use separator as _,-,+,&,* or whatever you like.
```

```
Mike-John-Anna
```

Replace Function

The replace function replaces one string within a text by another one. In the following example, we replace the name John by the name Anna .

In [83]:

```
text = "I like John, and Toko is my friend!"
text = text.replace( "John" , "Abdullah" )
text
```

Out[83]:

```
'I like Abdullah, and Toko is my friend!'
```

Split function

If we want to split specific parts of a string and put them into a list, we use the split function.

In [80]:

```
names = "John,Max,Bob,Anna" # Here we have a string of names separated by commas.
name_list = names.split( "," ) #We then use the split function and define the comma as
 the separator in order to save the individual names into a list.
name_list
```

Out[80]:

```
['John', 'Max', 'Bob', 'Anna']
```

Triple Quotes

The last topic of this chapter is triple quotes . They are just a way to write multi-line strings without the need of escape characters.

In [70]:

```
print ( '''Hello World!
This is a multi-line comment!
And we don't need to use escape characters
in order to write new empty lines!''' )
```

```
Hello World!
This is a multi-line comment!
And we don't need to use escape characters
in order to write new empty lines!
```

CHAPTER 4: LISTS AND TUPLES

Python lists are containers to store a set of values of any data type.

In [1]:

```
a = 'is'
b = 'nice'
my_list = ['my', "list", a, b]
my_list
```

Out[1]:

```
['my', 'list', 'is', 'nice']
```

In [12]:

```
numbers = [ 10 , 22 , 6 , 1 , 29 ]
names = [ "John" , "Alex" , "Bob" ]
mixed = [ "Anna" , 20 , 28.12 , True ]
```

In [101]:

```
My_friends = ['Abdullah', 'Toko', 'Yahya', 100, 20.4, 'R2 Girls', True]
My_friends
```

Out[101]:

```
['Abdullah', 'Toko', 'Yahya', 100, 20.4, 'R2 Girls', True]
```

In [3]:

```
hall = 11.25
kit = 18.0
liv = 20.0
bed = 10.75 #creating a list of lists
bath = 9.50
house =[['hallway', hall],['kittchen', kit], ['living room', liv], ['bedroom', bed], [
'bathroom', bath]]
house
```

Out[3]:

```
[['hallway', 11.25],
 ['kittchen', 18.0],
 ['living room', 20.0],
 ['bedroom', 10.75],
 ['bathroom', 9.5]]
```

Subsetting list / List indexing

A list can be indexed just like a string.

In [4]:

```
print(house[2])
```

```
['living room', 20.0]
```

In [6]:

```
print(house[-2]) #second element/list from the end
```

```
['bedroom', 10.75]
```

In [9]:

```
list = ['Khawla',1,'Bouchra', 'Safi', 3]
list[0]
```

Out[9]:

```
'Khawla'
```

In [104]:

```
list[1]
```

Out[104]:

```
1
```

In [10]:

```
list[-1]
```

Out[10]:

```
3
```

In [105]:

```
list[2]
```

Out[105]:

```
'Bouchra'
```

In [106]:

```
list[5]
```

```
---------------------------------------------------------------------------
-
IndexError                                Traceback (most recent call las
t)
<ipython-input-106-60b58d4d9246> in <module>
----> 1 list[5]

IndexError: list index out of range
```

In [14]:

```
numbers = [ 10 , 22 , 6 , 1 , 29 ]
numbers[1] + numbers[3]
```

Out[14]:

```
23
```

In [ ]:

```

```

List Methods

Consider the foolowing methods on a list

In [56]:

```
list1 = [1, 2, 3, 4, 7, 8, 15, 6, 2, 3]
list1
```

Out[56]:

```
[1, 2, 3, 4, 7, 8, 15, 6, 2, 3]
```

In [60]:

```
list1.sort() # rearranges the list in ascending order
print(list1)
```

```
[1, 2, 2, 3, 3, 4, 6, 7, 8, 15]
```

In [111]:

```
list1.reverse() # rearranges the list in descending order
list1
```

Out[111]:

```
[15, 8, 7, 6, 4, 3, 3, 2, 2, 1]
```

In [116]:

```
list1.append(10) # adds 10 at the end of list1
list1
```

Out[116]:

```
[15, 8, 7, 6, 3, 4, 3, 3, 2, 2, 1, 10, ['Bola', 'Shade'], 10]
```

In [42]:

```
list1.insert(4, 3) # adds 3 at index 4
list1
```

Out[42]:

```
[1, 2, 3, 4, 3, 7, 8, 15, 6, 2, 3, 'Ola', 'Fatimah', 'Bola', 13]
```

In [44]:

```
list1 = [1, 2, 3, 4, 7, 8, 15, 6, 2, 3]
list1.append(['Bola', "Ola", "Fatimah"]) # adds a new list to list1
list1
```

Out[44]:

```
[1, 2, 3, 4, 7, 8, 15, 6, 2, 3, ['Bola', 'Ola', 'Fatimah']]
```

In [45]:

```
list1.extend(['Bola'])
list1
```

Out[45]:

```
[1, 2, 3, 4, 7, 8, 15, 6, 2, 3, ['Bola', 'Ola', 'Fatimah'], 'Bola']
```

In [31]:

```
del list1[10]
list1
```

Out[31]:

```
[1, 2, 3, 4, 7, 8, 15, 6, 2, 3, 'Ola', 'Fatimah', 'Bola']
```

In [47]:

```
list1.extend([13, 14])
list1
```

Out[47]:

```
[1, 2, 3, 4, 7, 8, 15, 6, 2, 3, ['Bola', 'Ola', 'Fatimah'], 'Bola', 13, 1
4]
```

In [48]:

```
del(list1[-1])
list1
```

Out[48]:

```
[1, 2, 3, 4, 7, 8, 15, 6, 2, 3, ['Bola', 'Ola', 'Fatimah'], 'Bola', 13]
```

In [49]:

```
list3 = [1, 2, 3, 4, 7, 8, 15, 6, 2, 3]
list3.pop(2) # deletes the element at index 2 and returns its value
list3
```

Out[49]:

```
[1, 2, 4, 7, 8, 15, 6, 2, 3]
```

In [ ]:

```

```

In [50]:

```
list3.remove(15) # removes 15 from list2
list3
```

Out[50]:

```
[1, 2, 4, 7, 8, 6, 2, 3]
```

Slicing List

Instead of only accessing one single element, we can also define a range that we want to access. By using the colon, we can slice our lists and access multiple elements at once.

In [37]:

```
list3
```

Out[37]:

```
[1, 2, 4, 7, 8, 6, 2, 3]
```

In [38]:

```
print(list3[ 1 : 3 ]) # 2 and 4
print(list3[: 3 ]) # 1, 2 and 4
print(list3[ 1 :]) # 2, 4, 7, 8, 2, and 3
print(list3[:])
```

```
[2, 4]
[1, 2, 4]
[2, 4, 7, 8, 6, 2, 3]
[1, 2, 4, 7, 8, 6, 2, 3]
```

In [51]:

```
hall = 12.1
kit = 10.0
liv = 14.3
bed = 11
bath = 10.1
house =[['hallway', hall],['kittchen', kit], ['living room', liv], ['bedroom', bed], [
'bathroom', bath]]
house
```

Out[51]:

```
[['hallway', 12.1],
 ['kittchen', 10.0],
 ['living room', 14.3],
 ['bedroom', 11],
 ['bathroom', 10.1]]
```

In [52]:

```
house[2][1] #second list, at the index 1. This is called subsetting
```

Out[52]:

```
14.3
```

Modifying list elements (replacement)

In a list, we can also modify the values. For this, we index the elements in the same way.

In [33]:

```
numbers = [3, 4, 7, 22, 'Hassan']
numbers
```

Out[33]:

```
[3, 4, 7, 22, 'Hassan']
```

In [29]:

```
numbers[ 1 ] = 10                    #The second element of the numbers list is now 10
numbers[ 2 ] = "Sulaiman"            # instead of 7 and the third element of the names
                                       # list is now Sulaiman instead of 7.
numbers
```

Out[29]:

```
[3, 10, 'Sulaiman', 22, 'Hassan']
```

In [34]:

```
numbers
```

Out[34]:

```
[3, 4, 7, 22, 'Hassan']
```

In [35]:

```
numbers[2:] = ['s', 't']   # replaces the second element (7) with the new list. Because the second part of the slice is not stated, it sets it as as end
numbers
```

Out[35]:

```
[3, 4, 's', 't']
```

List operations

Some of the operators we already know can be used when working with lists – addition and multiplication.

In [182]:

```
print(numbers + ["soup"])
numbers * 2
```

```
[3, 10, 'Sulaiman', 22, 'Hassan', 'soup']
```

Out[182]:

```
[3, 10, 'Sulaiman', 22, 'Hassan', 3, 10, 'Sulaiman', 22, 'Hassan']
```

Membership Operarors

We use them to check if an element is a member of a sequence, but also to iterate over sequences. With the in or not in operators, we check if a sequence contains a certain element. If the element is in the list, it returns True . Otherwise it returns False .

In [215]:

```
lis = [ 10 , 20 , 30 , 40 , 50 ]
print ( 20 in list1) # True
print ( 60 in list1) # False
print ( 60 not in list1) # True
```

```
False
False
True
```

In [217]:

```
#But we also use membership operators, when we iterate over sequences with for loops.
for x in lis:
    print (x)
#For every element in the sequence, x becomes the value of the next element and gets pr
inted.
```

```
10
20
30
40
50
```

TUPLES

A tuple is an immutable data type in Python. Basically, all the reading and accessing functions like len, min and max stay the same and can be used with tuples. But of course it is not possible to use any modifying or appending functions.

In [138]:

```
a = () #empty tuple returns empple parentheses
a
```

Out[138]:

```
()
```

In [145]:

```
a = (2,) # tuple with only one element needs a comma to specify
# to the interpreterthat it's a tuple
a
```

Out[145]:

```
(2,)
```

In [142]:

```
a = (2,4,5)
a
```

Out[142]:

```
(2, 4, 5)
```

Once defined, a tuple cannot be altered or manipulated.

Tuple methods

In [146]:

```
a = (2, 5,  4, 5)
a.count(5) #will return the no of times that 5 occurs
```

Out[146]:

```
2
```

In [148]:

```
a.index(4) # will return the index of the first occurence of 4 in a
```

Out[148]:

```
2
```

CHAPTER 5: Dictionary and Sets

Dictionary is a collection of key - value pairs. A dictionary works a bit like a lexicon. One element in this data structure points to another. We are talking about key-value pairs. Every entry in this sequence has a key and a respective value. In other programming languages this structure is called hash map. Since the key now replaces the index, it has to be unique. This is not the case for the values. We can have many keys with the same value but when we address a certain key, it has to be the only one with that particular name. Also keys can’t be changed.

In [53]:

```
# Syntax :
a = {"key" : "value", "name" : "Azeez", "age": 33, 'sex' : "male", 'list5' :[1, 3, 5]}
a # if there were multiple keys with the same name, we couldn’t get a result because we
wouldn’t know
# which value we are talking about.
```

Out[53]:

```
{'key': 'value', 'name': 'Azeez', 'age': 33, 'sex': 'male', 'list5': [1,
3, 5]}
```

In [154]:

```
a['key']
```

Out[154]:

```
'value'
```

In [156]:

```
a['list5']
```

Out[156]:

```
[1, 3, 5]
```

In [157]:

```
a['name']
```

Out[157]:

```
'Azeez'
```

In [54]:

```
a['list5'] = [11,23]
a
```

Out[54]:

```
{'key': 'value', 'name': 'Azeez', 'age': 33, 'sex': 'male', 'list5': [11,
23]}
```

Properties of a dictionary

It is unorderd, It is mutable, It is indexed, Cannot have duplicate keys

Dictionary functions

Similar to lists, dictionaries also have a lot of functions and methods. But since they work a bit differently and they don't have indices, their functions are not the same.

In [60]:

```
a = { 'name' : 'Wale', 'from' : 'Nig', 'scores':[99,98,98.5]}
a
len(a) #Returns the length of a dictionary
str(a) #Returns the dictionary displayed as a string
```

Out[60]:

```
"{'name': 'Wale', 'from': 'Nig', 'scores': [99, 98, 98.5]}"
```

Dictionary Methods

In [61]:

```
a.items() #returns a list of (key-value) tuples
```

Out[61]:

```
dict_items([('name', 'Wale'), ('from', 'Nig'), ('scores', [99, 98, 98.
5])])
```

In [62]:

```
a.keys() #returns a list of key tuples
```

Out[62]:

```
dict_keys(['name', 'from', 'scores'])
```

In [63]:

```
a.values() #returns a list of values tuples
```

Out[63]:

```
dict_values(['Wale', 'Nig', [99, 98, 98.5]])
```

In [64]:

```
a.update({'name' : 'Zamba', 'from':'Moroco'})
print(a) #updates the dictionary with  the supplied key-value pairs
```

```
{'name': 'Zamba', 'from': 'Moroco', 'scores': [99, 98, 98.5]}
```

In [65]:

```
a.get('name') #returns the value of the specified key.
```

Out[65]:

```
'Zamba'
```

In [66]:

```
a.copy()   #Returns a copy of the dictionary
a
```

Out[66]:

```
{'name': 'Zamba', 'from': 'Moroco', 'scores': [99, 98, 98.5]}
```

In [67]:

```
del a['scores']
a
```

Out[67]:

```
{'name': 'Zamba', 'from': 'Moroco'}
```

In [68]:

```
a.fromkeys('from')  #Returns a new dictionary with the same keys but empty values
```

Out[68]:

```
{'f': None, 'r': None, 'o': None, 'm': None}
```

In [69]:

```
a
```

Out[69]:

```
{'name': 'Zamba', 'from': 'Moroco'}
```

In [70]:

```
b = {'age' : 22, "sex" : 'male'} #creating a second dictionary
a.update(b) #Adds the content of another dictionary (b) to an existing one (a)

a
```

Out[70]:

```
{'name': 'Zamba', 'from': 'Moroco', 'age': 22, 'sex': 'male'}
```

In [71]:

```
a.clear()   #Removes all elements from a dictionary
a
```

Out[71]:

```
{}
```

CONDITIONAL EXPRESSIONS

In Python, we must be able to execute instructions on a condition(s) being met. This is what Conditions are for!

if, elif, and else

These statements are multiway decision taken by our program due to certain conditions in our code.

In [222]:

```
# Syntax :
'''if(condition 1):
    print("yes")
elif (condition 2):
    print("no")
else:
    print("maybe")
'''
```

Out[222]:

```
'if(condition 1):\n    print("yes")\nelif (condition 2):\n    print("no")
\nelse:\n    print("maybe")\n'
```

In [223]:

```
x = 12
if (x > 11):
    print("Bigger number")
else :
    print("Lesser number")
```

```
Bigger number
```

EX : Take a variable x and print "Even" if the number is divisible by 2, otherwise print "Odd".

In [14]:

```
x = 7
if x%2 == 0 :
    print('x is even')
else :
    print('x is odd')
```

```
x is odd
```

elif

elif means else if. An if statement can be chained together with a lot of these elif statements followed by an else statement.

In [ ]:

```
# Syntax:
if (condition 1):
    #code
elif (condition 2): #This ladder will stop once a condition in if or elif is met.
    #code
elif (condition 3):
    #code
else :
    #code
```

EX : Take a variable y and print "Grade A" if y is greater than 90, "Grade B" if y is greater than 60 but less than or equal to 90 and "Grade F" Otherwise.

In [16]:

```
y=90

if(y>90):
    print("Grade A")
elif(y>60):
    print("Grade B")
else:
    print("Grade F")
```

```
Grade B
```

In [229]:

```
y = 0
if x < y:
    print('x is less than y')
elif x > y: #There is no limit on the number of elif statements.
    print('x is greater than y')
else: # If there is an else clause, it has to be at the end, but there doesn't have to
 be one.
    print('x and y are equal')
```

```
x is greater than y
```

In [234]:

```
option = input('select an option: ')
if option == 'a':
    print('Bad guess')
elif option == 'b':
    print('Good guess')
elif option == 'c':
    print('Close, but not correct')
```

```
select an option: c
Close, but not correct
```

In [236]:

```
number = input ( "Enter a number: " )
number = int (number)
if number < 10 :
    print ( "Your number is less than 10" )
elif number > 10 :
    print ( "Your number is greater than 10" )
else :
    print ( "Your number is 10" )
```

```
Enter a number: 10
Your number is 10
```

NESTED IF-STATEMENTS

embedding another if statement in an if statement.

In [241]:

```
n = 2
if n > 10:
    print('above')
if n < 10:
    print('below')
else:
    print('equal')
```

```
below
```

In [239]:

```
number = 4
if number % 2 == 0 :
    if number == 0 :
        print ( "Your number is even but zero" )
    else:
        print ( "Your number is even" )
else :
        print ( "Your number is odd" )
```

```
Your number is even
```

Logical Operators

They are of 3 types: and # True if both operands are true, else False or # True, if at least one operand is true, else False not # inverts True to False and vice versa.

In [242]:

```
(2 < 3) and (4 >= 3)
```

Out[242]:

```
True
```

In [243]:

```
3 < 2 and 4 <= 3
```

Out[243]:

```
False
```

Loops

Sometimes we want to repeat a set of statements in our program.For instance; print 0 to 10. Loops make it easy . Primarily, there ar 2 types: while loop and for loop

while loop

In [ ]:

```
#Syntax :
while condition :
    #body of the loop.
```

In [248]:

```
number = 0
while number < 10 :
    print(number)
    number += 1
```

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

In [249]:

```
number = 0
while number < 10 :
    number += 1
    print(number)
```

```
1
2
3
4
5
6
7
8
9
10
```

In [250]:

```
i = 0
while i < 4:
    print("Hello ")
    i = i + 1
```

```
Hello
Hello
Hello
Hello
```

For loop

is used to iterate through a sequence [iterables]like list,tuple or string.

In [252]:

```
# Syntax : FOR LOOP, we don't need to have a condition. This executes on sequences
list = [ 1,3,9]
for item in list:
    print(item)
```

```
1
3
9
```

In [254]:

```
numbers = [ 10 , 20 , 30 , 40 ]
for number in numbers:
    print (number)
```

```
10
20
30
40
```

Range function

The range function is used to generate a sequence of numbers. We can specify the start, stop,and step-size.

In [256]:

```
"""Syntax:
range(start, stop, step-size)"""
#step-size is not commonly used.
```

Out[256]:

```
'Syntax:\nrange(start, stop, step-size)'
```

In [270]:

```
for i in range(0, 5):
    print(i)
#With the range function, we can create lists that contain all numbers in between two n
umbers.
```

```
0
1
2
3
4
```

In [261]:

```
range(5) # we can as well use this
```

Out[261]:

```
range(0, 5)
```

EX: Write a for loop to print all the numbers between 10 and 50.

In [18]:

```
for i in range(11,15):
    print(i)
```

```
11
12
13
14
```

A for loop to print at 20 intervals

In [268]:

```
for x in range(1, 100, 20):
    print(x)
```

```
1
21
41
61
81
```

EX: Write a for loop to print all the odd numbers between 20 and 30.

In [273]:

```
for x in range ( 20 , 30, 2 ):
    print(x)
```

```
20
22
24
26
28
```

The break statement

'break' is used to exit a loop. it tells the interpreter to exit the loop.

In [274]:

```
for i in range(0,80):
    print(i)
    if i == 5:
        break
```

```
0
1
2
3
4
5
```

In [284]:

```
number = int(input("enter a number "))
while number < 10 :
    number += 1
    if number == 5 :
        break
    print(number)
```

```
enter a number 0
1
2
3
4
```

In [289]:

```
number = int(input("enter a number "))
while number < 10 :
    number += 1
    if number == 5 :
        break
    print(number)
```

```
enter a number 5
6
7
8
9
10
```

The continue statement

It is used to stop the current iteration of the loop and continue with the next. It instructs the program to skip this iteration. If we don't want to break the full loop, but to skip only one iteration, we can use the continue statement.

In [296]:

```
for i in range(5):
    print('iterating')
    if i == 3:
        continue
    print(i)
```

```
iterating
0
iterating
1
iterating
2
iterating
iterating
4
```

In [304]:

```
number = 0
while number < 6 :
    number += 1
    if number == 5 :
        continue
    print(number)
```

```
1
2
3
4
6
```

Pass statement

Pass is a null statement in Python. It instructs to 'do nothing'. You use it as placeholder in case you are not ready to run the code.

In [305]:

```
l = [3, 4, 2]
for item in l:
    pass
print('Not yet ready')
```

```
Not yet ready
```

In [ ]:

```
'''The pass statement is a very special statement, since it does absolutely nothing.
Actually, it is not really a loop control statement, but a placeholder for code.
'''
if number == 10 :
    pass
else :
    pass
while number < 10 :
    pass
```

CHAPTER 8: Functions and recursion

A function is a group of statements performing a specific task. It can be reused over and over time.

In [ ]:

```
# Syntax:
    def function():
    print('Hey, friend!')
#This function can be called any number of times and anywhere in the program.
```

Calling a function

To call a function, we put the name of the function followed by parentheses

In [ ]:

```
function()
```

In [ ]:

```
def hello():
    print( "Hello" )
#Here we have a function hello that prints the text “Hello” . It’s quite simple. Now we
can call the function by using its name.
hello()
```

Parameters

If we want to make our functions more dynamic, we can define parameters. These parameters can then be processed in the function code.

In [ ]:

```
def print_sum(number1, number2):
    print(number1 + number2)

print_sum(20 , 30 ) #call print_sum
```

Return values

The two functions we wrote were just executing statements. What we can also do is return a certain value. This value can then be saved in a variable or it can be processed. For this, use the keyword return.

In [ ]:

```
def add(number1, number2):
    return number1 + number2
#Here we return the sum of the two parameters instead of printing it. But we can then u
se this result in our code.
```

In [ ]:

```
number = add( 10 , 20 )
print(number)
```

In [ ]:

```
def greet(name):
    gr = 'Hello ' + name
    return gr

a = greet('Azeez')
print(a)
```

EX : Create a function that takes two numbers as argument and returns the greater of the two.

In [19]:

```
def compare(a,b):
    if(a>b):
        greater=a
    else:
        greater=b

    return greater
```

In [23]:

```
compare(1.002, 1.0002)
```

Out[23]:

```
1.002
```

Default parameters

Sometimes we want our parameters to have default values in case we don't specify anything else. We can do that by assigning values in the function definition.

In [ ]:

```
def say(text= "Default Text" ):
print (text)
#In this case, our function say prints the text that we pass as a parameter.
#But if we don't pass anything, it prints the default text.
```

In [ ]:

```
def greet(name ='stranger'):
    #function body
greet() #name will be 'stranger' by default
greet('Azeez') # name will be 'Azeez in function body'
```

Variable parameters

Sometimes we want our functions to have a variable amount of parameters. For that, we use the asterisk symbol (*) in our parameters. We then treat the parameter as a sequence.

In [1]:

```
def print_sum(*numbers):
    result = 0
    for x in numbers:
        result += x
        print(result)
'''Here we pass the parameter numbers . That may be five, ten or a hundred numbers. We
 then iterate over
this parameter, add every value to our sum and print it.'''
```

Out[1]:

```
'Here we pass the parameter numbers . That may be five, ten or a hundred n
umbers. We then iterate over \nthis parameter, add every value to our sum
and print it.'
```

In [2]:

```
print_sum( 10 , 20 , 30 , 40 )
```

```
10
30
60
100
```

In [19]:

```
def fun(*m):
    b = 1
    for s in m:
        b += 1
        print(b)
```

In [20]:

```
fun(1,2,3,4,5)
```

```
2
3
4
5
6
```

RECURSION

Recursion is a function wwhich calls itself. It is used to directly use a mathematical formula as a function.

In [10]:

```
def factorial(n):
    if i == 0 or i==1: #Base condition
        return 1
    else :
        return n*factorial(n-1) #function calling itself
```

In [14]:

```
i = 0
```

In [15]:

```
factorial(4)
```

Out[15]:

```
1
```

In [33]:

```
def fact(x):
    if x==1 :
        return 1
    else :
        return fact(x-1)
```

In [32]:

```
fact(2)
```

Out[32]:

```
1
```

In [ ]:

```
fact
```

CHAPTER 9:

Reading csv and excel file

In [73]:

```
import pandas as pd #importing pandas library
```

In [102]:

```
pwd #checking directory
```

Out[102]:

```
'C:\\Users\\Hamzat'
```

In [128]:

```
Data = r'C:\Users\Hamzat\Downloads\ANALYTICS VIDH\data1.xlsx'  #importing excel file.
data = pd.read_excel(Data)
data.head()
```

Out[128]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500

In [129]:

```
#Reading the dataset
Data1 = r'C:\Users\Hamzat\Downloads\ANALYTICS VIDH\data2.csv'  #importing excel file.
data1 = pd.read_csv(Data1)
data1.head() # checking the first 5 rows
```

Out[129]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500

In [111]:

```
data1.shape #Seeing the dimensions of the df dataframe
```

Out[111]:

```
(891, 12)
```

In [144]:

```
# get the number of missing data points per column
missing_values_count = data1.isnull().sum()
missing_values_count
```

Out[144]:

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

In [113]:

```
# bottom 5 rows
data1.tail()
```

Out[113]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	C
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.00	
887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.00	
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.45	
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.00	
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.75	

In [115]:

```
# Seeing the names of the columns in the dataframe
data1.columns
```

Out[115]:

```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

In [141]:

```
bigmart = pd.read_csv(r'C:\Users\Hamzat\Downloads\ANALYTICS VIDH\data\bigmart_data.csv'
)
data.head()
```

Out[141]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outl
0	FDA15	9.30	Low Fat	0.016047	Dairy	249.8092	
1	DRC01	5.92	Regular	0.019278	Soft Drinks	48.2692	
2	FDN15	17.50	Low Fat	0.016760	Meat	141.6180	
3	FDX07	19.20	Regular	0.000000	Fruits and Vegetables	182.0950	
4	NCD19	8.93	Low Fat	0.000000	Household	53.8614	

In [142]:

```
bigmart.shape
```

Out[142]:

```
(8523, 12)
```

In [143]:

```
# get the number of missing data points per column
missing_values_count = bigmart.isnull().sum()
missing_values_count
```

Out[143]:

```
Item_Identifier               0
Item_Weight                1463
Item_Fat_Content               0
Item_Visibility                0
Item_Type                      0
Item_MRP                       0
Outlet_Identifier              0
Outlet_Establishment_Year      0
Outlet_Size                 2410
Outlet_Location_Type           0
Outlet_Type                    0
Item_Outlet_Sales              0
dtype: int64
```

In [152]:

```
# Selecting a single column
bigmart['Item_Type']
```

Out[152]:

```
0                       Dairy
1                 Soft Drinks
2                        Meat
3       Fruits and Vegetables
4                   Household
                ...
8518              Snack Foods
8519             Baking Goods
8520       Health and Hygiene
8521              Snack Foods
8522              Soft Drinks
Name: Item_Type, Length: 8523, dtype: object
```

In [155]:

```
bigmart[["Item_Type", "Item_Fat_Content"]] # selecting multiple columns using names
```

Out[155]:

	Item_Type	Item_Fat_Content
0	Dairy	Low Fat
1	Soft Drinks	Regular
2	Meat	Low Fat
3	Fruits and Vegetables	Regular
4	Household	Low Fat
...	...	...
8518	Snack Foods	Low Fat
8519	Baking Goods	Regular
8520	Health and Hygiene	Low Fat
8521	Snack Foods	Regular
8522	Soft Drinks	Low Fat

8523 rows × 2 columns

In [158]:

```
# selecting rows by their positions
bigmart.iloc[:4] #returns the first 4 rows
```

Out[158]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outl
0	FDA15	9.30	Low Fat	0.016047	Dairy	249.8092	
1	DRC01	5.92	Regular	0.019278	Soft Drinks	48.2692	
2	FDN15	17.50	Low Fat	0.016760	Meat	141.6180	
3	FDX07	19.20	Regular	0.000000	Fruits and Vegetables	182.0950	

In [161]:

```
# selecting columns by their positions
bigmart.iloc[:,:2] #returns the first 2 columns
```

Out[161]:

	Item_Identifier	Item_Weight
0	FDA15	9.300
1	DRC01	5.920
2	FDN15	17.500
3	FDX07	19.200
4	NCD19	8.930
...	...	...
8518	FDF22	6.865
8519	FDS36	8.380
8520	NCJ29	10.600
8521	FDN46	7.210
8522	DRG01	14.800

8523 rows × 2 columns

In [167]:

```
# selecting column by their positions
bigmart.iloc[:,2] #third column
```

Out[167]:

```
0       Low Fat
1       Regular
2       Low Fat
3       Regular
4       Low Fat
         ...
8518    Low Fat
8519    Regular
8520    Low Fat
8521    Regular
8522    Low Fat
Name: Item_Fat_Content, Length: 8523, dtype: object
```

In [171]:

```
bigmart[bigmart['Item_Type']==2]
```

Out[171]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet

DATA MANIPULATION WITH PANDAS

NOW WE START THE REAL THING!!!

Pandas is the most widely used library of python for data science. It is incredibly helpful in manipulating the data so that you can derive better insights and build great machine learning models. In this notebook, we will have a look at some of the intermediate concepts of working with pandas.

Table of Contents

In this notebook we will use the Big Mart Sales Data. You can download the data from : https://datahack.analyticsvidhya.com/contest/practice-problem-big-mart-sales-iii/download/train-file (https://datahack.analyticsvidhya.com/contest/practice-problem-big-mart-sales-iii/download/train-file)

In [200]:

```
import pandas as pd
import numpy as np

# read the dataset
data_BM = pd.read_csv(r'C:\Users\Hamzat\Downloads\ANALYTICS VIDH\data\bigmart_data.csv'
)

data_BM = data_BM.dropna(how="any") # drop the null values

data_BM.head() # view the top results
```

Out[200]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outle
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	
4	NCD19	8.930	Low Fat	0.000000	Household	53.8614	
5	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	

1. Sorting dataframes

Pandas data frame has two useful functions

- **sort_values()**: to sort pandas data frame by one or more columns
- **sort_index()**: to sort pandas data frame by row index

Each of these functions come with numerous options, like sorting the data frame in specific order (ascending or descending), sorting in place, sorting with missing values, sorting by specific algorithm etc.

Suppose you want to sort the dataframe by "Outlet_Establishment_Year" then you will use **sort_values**

In [174]:

```
# sort by year
sorted_data = data_BM.sort_values(by='Outlet_Establishment_Year')
# print sorted data
sorted_data[:5]
```

Out[174]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	O
2812	FDR60	14.30	Low Fat	0.130307	Baking Goods	75.7328	
5938	NCJ06	20.10	Low Fat	0.034624	Household	118.9782	
3867	FDY38	13.60	Regular	0.119077	Dairy	231.2300	
1307	FDB37	20.25	Regular	0.022922	Baking Goods	240.7538	
5930	NCA18	10.10	Low Fat	0.056031	Household	115.1492	

- Now `sort_values` takes multiple options like:
  - `ascending` : The default sorting order is ascending, when you pass False here then it sorts in descending order.
  - `inplace` : whether to do inplace sorting or not

In [176]:

```
# sort in place and descending order
data_BM.sort_values(by='Outlet_Establishment_Year', ascending=False, inplace=True)
data_BM[:5]
```

Out[176]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	O
2825	FDL16	12.85	Low Fat	0.169139	Frozen Foods	46.4060	
7389	NCD42	16.50	Low Fat	0.012689	Health and Hygiene	39.7506	
2165	DRJ39	20.25	Low Fat	0.036474	Dairy	218.3482	
2162	FDR60	14.30	Low Fat	0.130946	Baking Goods	76.7328	
2158	FDM58	16.85	Regular	0.080015	Snack Foods	111.8544	

You might want to sort a data frame based on the values of multiple columns. We can specify the columns we want to sort by as a list in the argument for sort_values().

In [178]:

```
# read the dataset
data_BM = pd.read_csv(r'C:\Users\Hamzat\Downloads\ANALYTICS VIDH\data\bigmart_data.csv'
)
# drop the null values
data_BM = data_BM.dropna(how="any")

# sort by multiple columns
data_BM.sort_values(by=['Outlet_Establishment_Year', 'Item_Outlet_Sales'], ascending=Fa
lse)[:5]
```

Out[178]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	O
43	FDC02	21.35	Low Fat	0.069103	Canned	259.9278	
2803	FDU51	20.20	Regular	0.096907	Meat	175.5028	
641	FDY51	12.50	Low Fat	0.081465	Meat	220.7798	
2282	NCX30	16.70	Low Fat	0.026729	Household	248.4776	
2887	FDR25	17.00	Regular	0.140090	Canned	265.1884	

- Note that when sorting by multiple columns, pandas sort_value() uses the first variable first and second variable next.
- We can see the difference by switching the order of column names in the list.

In [179]:

```
# changed the order of columns
data_BM.sort_values(by=['Item_Outlet_Sales', 'Outlet_Establishment_Year'], ascending=Fa
lse, inplace=True)
data_BM[:5]
```

Out[179]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	O
4888	FDF39	14.850	Regular	0.019495	Dairy	261.2910	
4289	NCM05	6.825	Low Fat	0.059847	Health and Hygiene	262.5226	
6409	FDA21	13.650	Low Fat	0.035931	Snack Foods	184.4924	
4991	NCQ53	17.600	Low Fat	0.018905	Health and Hygiene	234.6590	
5752	FDI15	13.800	Low Fat	0.141326	Dairy	265.0884	

- We can use **sort_index()** to sort pandas dataframe to sort by row index or names.
- In this example, row index are numbers and in the earlier example we sorted data frame by 'Item_Outlet_Sales', 'Outlet_Establishment_Year' and therefore the row index are jumbled up.
- We can sort by row index (with inplace=True option) and retrieve the original dataframe.

In [180]:

```
# sort by index
data_BM.sort_index(inplace=True)
data_BM[:5]
```

Out[180]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outle
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	
4	NCD19	8.930	Low Fat	0.000000	Household	53.8614	
5	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	

2. Merging dataframes

- Joining and merging DataFrames is the core process to start with data analysis and machine learning tasks.
- It is one of the toolkits which every Data Analyst or Data Scientist should master because in almost all the cases data comes from multiple source and files.
- Pandas has two useful functions for merging dataframes:
  - **concat()**
  - **merge()**

Creating dummy data

In [181]:

```
# create dummy data
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                   index=[0, 1, 2, 3])


df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                   index=[4, 5, 6, 7])


df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                   index=[8, 9, 10, 11])
```

a. concat() for combining dataframes

- Suppose you have the following three dataframes: df1, df2 and df3 and you want to combine them **"row-wise"** so that they become a single dataframe like the given image:
- You can use **concat()** here. You will have to pass the names of the DataFrames in a list as the argument to the concat().

In [182]:

```
# combine dataframes
result = pd.concat([df1, df2, df3])
result
```

Out[182]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

- pandas also provides you with an option to label the DataFrames, after the concatenation, with a key so that you may know which data came from which DataFrame.
- You can achieve the same by passing additional argument **keys** specifying the label names of the DataFrames in a list.

In [183]:

```
# combine dataframes
result = pd.concat([df1, df2, df3], keys=['x', 'y', 'z'])
result
```

Out[183]:

		A	B	C	D
x	**0**	A0	B0	C0	D0
	1	A1	B1	C1	D1
	2	A2	B2	C2	D2
	3	A3	B3	C3	D3
y	**4**	A4	B4	C4	D4
	5	A5	B5	C5	D5
	6	A6	B6	C6	D6
	7	A7	B7	C7	D7
z	**8**	A8	B8	C8	D8
	9	A9	B9	C9	D9
	10	A10	B10	C10	D10
	11	A11	B11	C11	D11

- Mentioning the keys also makes it easy to retrieve data corresponding to a particular DataFrame.
- You can retrieve the data of DataFrame df2 which had the label `y` by using the `loc` method.

In [184]:

```
# get second dataframe
result.loc['y']
```

Out[184]:

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

- When gluing together multiple DataFrames, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in the following three ways:
  - Take the union of them all, `join='outer'` . This is the default option as it results in zero information loss.
  - Take the intersection, `join='inner'` .
  - Use a specific index, as passed to the `join_axes` argument.
- Here is an example of each of these methods. First, the default `join='outer'` behavior:

In [194]:

```
df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
                    'D': ['D2', 'D3', 'D6', 'D7'],
                    'F': ['F2', 'F3', 'F6', 'F7']},
                   index=[2, 3, 6, 7])


result = pd.concat([df1, df4], axis=1, sort=False)
result
```

Out[194]:

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

- Here is the same thing with `join='inner'` :

In [195]:

```
result = pd.concat([df1, df4], axis=1, join='inner')
result
```

Out[195]:

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

- Lastly, suppose we just wanted to `reuse the exact index` from the original DataFrame:

In [198]:

```
result = pd.concat([df1, df4], axis=1, join_axes=[df1.index])
result
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-198-fefe7d75ae95> in <module>
----> 1 result = pd.concat([df1, df4], axis=1, join_axes=[df1.index])
      2 result

TypeError: concat() got an unexpected keyword argument 'join_axes'
```

b. merge() for combining dataframes using SQL like joins

- Another ubiquitous operation related to DataFrames is the merging operation.
- Two DataFrames might hold different kinds of information about the same entity and linked by some common feature/column.
- We can use **merge()** to combine such dataframes in pandas.

Creating dummy data

In [188]:

```
# create dummy data
df_a = pd.DataFrame({
        'subject_id': ['1', '2', '3', '4', '5'],
        'first_name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
        'last_name': ['Anderson', 'Ackerman', 'Ali', 'Aoni', 'Atiches']})

df_b = pd.DataFrame({
        'subject_id': ['4', '5', '6', '7', '8'],
        'first_name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
        'last_name': ['Bonder', 'Black', 'Balwner', 'Brice', 'Btisan']})

df_c = pd.DataFrame({
        'subject_id': ['1', '2', '3', '4', '5', '7', '8', '9', '10', '11'],
        'test_id': [51, 15, 15, 61, 16, 14, 15, 1, 61, 16]})
```

Now these are our dataframes:

- Let's start with a basic join, we want to combine `df_a` with `df_c` based on the `subject_id` column.

In [189]:

```
pd.merge(df_a, df_c, on='subject_id')
```

Out[189]:

	subject_id	first_name	last_name	test_id
0	1	Alex	Anderson	51
1	2	Amy	Ackerman	15
2	3	Allen	Ali	15
3	4	Alice	Aoni	61
4	5	Ayoung	Atiches	16

- Now that we have done a basic join, let's get into **some commmon SQL joins.**

Merge with outer join

- “Full outer join produces the set of all records in Table A and Table B, with matching records from both sides where available. If there is no match, the missing side will contain null.”

In [190]:

```
pd.merge(df_a, df_b, on='subject_id', how='outer')
```

Out[190]:

	subject_id	first_name_x	last_name_x	first_name_y	last_name_y
0	1	Alex	Anderson	NaN	NaN
1	2	Amy	Ackerman	NaN	NaN
2	3	Allen	Ali	NaN	NaN
3	4	Alice	Aoni	Billy	Bonder
4	5	Ayoung	Atiches	Brian	Black
5	6	NaN	NaN	Bran	Balwner
6	7	NaN	NaN	Bryce	Brice
7	8	NaN	NaN	Betty	Btisan

Merge with inner join

- “Inner join produces only the set of records that match in both Table A and Table B.”

In [191]:

```
pd.merge(df_a, df_b, on='subject_id', how='inner')
```

Out[191]:

	subject_id	first_name_x	last_name_x	first_name_y	last_name_y
0	4	Alice	Aoni	Billy	Bonder
1	5	Ayoung	Atiches	Brian	Black

Merge with right join

- “Right outer join produces a complete set of records from Table B, with the matching records (where available) in Table A. If there is no match, the left side will contain null.”

In [192]:

```
pd.merge(df_a, df_b, on='subject_id', how='right')
```

Out[192]:

	subject_id	first_name_x	last_name_x	first_name_y	last_name_y
0	4	Alice	Aoni	Billy	Bonder
1	5	Ayoung	Atiches	Brian	Black
2	6	NaN	NaN	Bran	Balwner
3	7	NaN	NaN	Bryce	Brice
4	8	NaN	NaN	Betty	Btisan

Merge with left join

- “Left outer join produces a complete set of records from Table A, with the matching records (where available) in Table B. If there is no match, the right side will contain null.”

In [193]:

```
pd.merge(df_a, df_b, on='subject_id', how='left')
```

Out[193]:

	subject_id	first_name_x	last_name_x	first_name_y	last_name_y
0	1	Alex	Anderson	NaN	NaN
1	2	Amy	Ackerman	NaN	NaN
2	3	Allen	Ali	NaN	NaN
3	4	Alice	Aoni	Billy	Bonder
4	5	Ayoung	Atiches	Brian	Black

Merge OR Concat : Which to use when?

1. After learning both of the functions in detail, chances are that you might be confused which to use when.
2. One major difference is that `merge()` is used to combine dataframes on the basis of values of **common columns**. While `concat()` is used to **append dataframes** one below the other (or sideways, depending on whether the axis option is set to 0 or 1).
3. Exact usage depends upon the kind of data you have and analysis you want to perform.

Apply function

In [201]:

```
# read the dataset
data_BM = pd.read_csv(r'C:\Users\Hamzat\Downloads\ANALYTICS VIDH\data\bigmart_data.csv'
)

data_BM = data_BM.dropna(how="any")
# reset index after dropping
data_BM = data_BM.reset_index(drop=True)
# view the top results
data_BM.head()
```

Out[201]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outle
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	
3	NCD19	8.930	Low Fat	0.000000	Household	53.8614	
4	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	

Apply function

- Apply function can be used to perform pre-processing/data-manipulation on your data both row wise and column wise.
- It is a faster method than simply using a **for** loop over your dataframe.
- Almost every time I need to iterate over a dataframe or it's rows/columns, I will think of using the `apply`.
- Hence, it is widely used in feature engineering code.

In [203]:

```
# accessing row wise
data_BM.apply(lambda x: x)
```

Out[203]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	O
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	
3	NCD19	8.930	Low Fat	0.000000	Household	53.8614	
4	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	
...	...	...	...	...	...	...	
4645	FDF53	20.750	reg	0.083607	Frozen Foods	178.8318	
4646	FDF22	6.865	Low Fat	0.056783	Snack Foods	214.5218	
4647	NCJ29	10.600	Low Fat	0.035186	Health and Hygiene	85.1224	
4648	FDN46	7.210	Regular	0.145221	Snack Foods	103.1332	
4649	DRG01	14.800	Low Fat	0.044878	Soft Drinks	75.4670	

4650 rows × 12 columns

In [204]:

```
# access first row
data_BM.apply(lambda x: x[0])
```

Out[204]:

```
Item_Identifier                  FDA15
Item_Weight                        9.3
Item_Fat_Content               Low Fat
Item_Visibility              0.0160473
Item_Type                        Dairy
Item_MRP                       249.809
Outlet_Identifier               OUT049
Outlet_Establishment_Year         1999
Outlet_Size                     Medium
Outlet_Location_Type            Tier 1
Outlet_Type           Supermarket Type1
Item_Outlet_Sales               3735.14
dtype: object
```

In [205]:

```
# accessing column wise
data_BM.apply(lambda x: x, axis=1)
```

Out[205]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	O
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	
3	NCD19	8.930	Low Fat	0.000000	Household	53.8614	
4	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	
...	...	...	...	...	...	...	
4645	FDF53	20.750	reg	0.083607	Frozen Foods	178.8318	
4646	FDF22	6.865	Low Fat	0.056783	Snack Foods	214.5218	
4647	NCJ29	10.600	Low Fat	0.035186	Health and Hygiene	85.1224	
4648	FDN46	7.210	Regular	0.145221	Snack Foods	103.1332	
4649	DRG01	14.800	Low Fat	0.044878	Soft Drinks	75.4670	

4650 rows × 12 columns

In [206]:

```
# access first column by index
data_BM.apply(lambda x: x[0], axis=1)
```

Out[206]:

```
0       FDA15
1       DRC01
2       FDN15
3       NCD19
4       FDP36
        ...
4645    FDF53
4646    FDF22
4647    NCJ29
4648    FDN46
4649    DRG01
Length: 4650, dtype: object
```

In [207]:

```
# access by column name
data_BM.apply(lambda x: x["Item_Fat_Content"], axis=1)
```

Out[207]:

```
0       Low Fat
1       Regular
2       Low Fat
3       Low Fat
4       Regular
         ...
4645        reg
4646    Low Fat
4647    Low Fat
4648    Regular
4649    Low Fat
Length: 4650, dtype: object
```

- You can also use `apply` to implement a **condition** individually on every row/column of your dataframe.
- Suppose you want to clip Item_MRP to 200 and not consider any value greater than that.

```
def clip_price(price):
  if price > 200:
      price = 200
  return price
```

In [208]:

```
# before clipping
data_BM["Item_MRP"][:5]
```

Out[208]:

```
0    249.8092
1     48.2692
2    141.6180
3     53.8614
4     51.4008
Name: Item_MRP, dtype: float64
```

In [209]:

```
# clip price if it is greater than 200
def clip_price(price):
    if price > 200:
        price = 200
    return price

# after clipping
data_BM["Item_MRP"].apply(lambda x: clip_price(x))[:5]
```

Out[209]:

```
0    200.0000
1     48.2692
2    141.6180
3     53.8614
4     51.4008
Name: Item_MRP, dtype: float64
```

- Suppose you want to label encode Outlet_Location_Type as 0, 1 and 2 for Tier 1, Tier 2 and Tier 3 city, your logic would be:

```
def label_encode(city):
    if city == 'Tier 1':
        label = 0
    elif city == 'Tier 2':
        label = 1
    else:
        label = 2
    return label
```

- You can use the `apply` to operate `label_encode` logic on every row of the Outlet_Location_Type column.

In [210]:

```
# before label encoding
data_BM["Outlet_Location_Type"][:5]
```

Out[210]:

```
0    Tier 1
1    Tier 3
2    Tier 1
3    Tier 3
4    Tier 3
Name: Outlet_Location_Type, dtype: object
```

In [211]:

```
# label encode city type
def label_encode(city):
    if city == 'Tier 1':
        label = 0
    elif city == 'Tier 2':
        label = 1
    else:
        label = 2
    return label

# operate label_encode on every row of Outlet_Location_Type
data_BM["Outlet_Location_Type"] = data_BM["Outlet_Location_Type"].apply(label_encode)
```

In [212]:

```
# after label encoding
data_BM["Outlet_Location_Type"][:5]
```

Out[212]:

```
0    0
1    2
2    0
3    2
4    2
Name: Outlet_Location_Type, dtype: int64
```

CHAPTER : Aggregating Data

1. Aggregating data

There are multiple functions that can be used to perform useful aggregations on data in pandas:

- groupby
- crosstab
- pivottable

a. What is the mean price for each item type? : groupby

- In the given data set, I want to find out **what is the mean price for each item type**?
- You can use **groupby()** to achieve this.
- The first step would be to group the data by Item_Type column.

In [214]:

```
# group price based on item type
price_by_item = data_BM.groupby('Item_Type')

# display first few rows
price_by_item.first()
```

Out[214]:

Item_Type	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_MRP	Outlet_I
Baking Goods	FDP36	10.395	Regular	0.000000	51.4008	
Breads	FDW11	12.600	Low Fat	0.048981	61.9194	
Breakfast	FDP49	9.000	Regular	0.069089	56.3614	
Canned	FDC02	21.350	Low Fat	0.069103	259.9278	
Dairy	FDA15	9.300	Low Fat	0.016047	249.8092	
Frozen Foods	FDR28	13.850	Regular	0.025896	165.0210	
Fruits and Vegetables	FDY07	11.800	Low Fat	0.000000	45.5402	
Hard Drinks	DRJ59	11.650	low fat	0.019356	39.1164	
Health and Hygiene	NCB42	11.800	Low Fat	0.008596	115.3492	
Household	NCD19	8.930	Low Fat	0.000000	53.8614	
Meat	FDN15	17.500	Low Fat	0.016760	141.6180	
Others	NCM43	14.500	Low Fat	0.019472	164.8210	
Seafood	FDH21	10.395	Low Fat	0.031274	160.0604	
Snack Foods	FDO10	13.650	Regular	0.012741	57.6588	
Soft Drinks	DRC01	5.920	Regular	0.019278	48.2692	
Starchy Foods	FDB11	16.000	Low Fat	0.060837	226.8404	

- Now that you have grouped by Item_Type, the next step would be to calculate the mean of Item_MRP.

In [215]:

```
# mean price by item
price_by_item.Item_MRP.mean()
```

Out[215]:

```
Item_Type
Baking Goods             125.795653
Breads                   141.300639
Breakfast                134.090683
Canned                   138.551179
Dairy                    149.481471
Frozen Foods             140.095830
Fruits and Vegetables    145.418257
Hard Drinks              140.102908
Health and Hygiene       131.437324
Household                149.884244
Meat                     140.279344
Others                   137.640870
Seafood                  146.595782
Snack Foods              147.569955
Soft Drinks              130.910182
Starchy Foods            151.256747
Name: Item_MRP, dtype: float64
```

- You can use `groupby` with **multiple** columns of the dataset too.
- In this case, if you want to group first based on the Item_Type and then Item_MRP you can simply pass a list of column names.

In [216]:

```
# group on multiple columns
multiple_groups = data_BM[:10].groupby(['Item_Type', 'Item_Fat_Content'])
multiple_groups.first()
```

Out[216]:

Item_Type	Item_Fat_Content	Item_Identifier	Item_Weight	Item_Visibility	Item_MRP	Outlet_I
Baking Goods	Regular	FDP36	10.395	0.000000	51.4008	
Dairy	Low Fat	FDA15	9.300	0.016047	249.8092	
	Regular	FDA03	18.500	0.045464	144.1102	
Fruits and Vegetables	Low Fat	FDY07	11.800	0.000000	45.5402	
	Regular	FDX32	15.100	0.100014	145.4786	
Household	Low Fat	NCD19	8.930	0.000000	53.8614	
Meat	Low Fat	FDN15	17.500	0.016760	141.6180	
Snack Foods	Regular	FDO10	13.650	0.012741	57.6588	
Soft Drinks	Regular	DRC01	5.920	0.019278	48.2692	

You can read more about **groupby** and other related functions [here. (http://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html)](http://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html)

b. How are outlet sizes distributed based on the city type? : crosstab

- This function is used to get an initial “feel” (view) of the data. Here, we can validate some basic hypothesis.
- For example, in this case, "Outlet_Location_Type" is expected to affect the "Outlet_Size" significantly. This can be tested using cross-tabulation as shown below:

In [217]:

```
# generate crosstab of Outlet_Size and Outlet_Location_Type
pd.crosstab(data_BM["Outlet_Size"],data_BM["Outlet_Location_Type"],margins=True)
```

Out[217]:

Outlet_Location_Type	0	1	2	All
Outlet_Size				
High	0	0	932	932
Medium	930	0	928	1858
Small	930	930	0	1860
All	1860	930	1860	4650

- If you notice in the above `crosstab` there are interesting insights like 50% of medium size outlets are present only in either Tier 1 or Tier 2 cities.
- Another counter intuitive thing to notice is that high outlet size is only present in Tier 3 city though general assumption would be towards Tier 1 cities having larger outlet sizes.

c. How are the sales changing per year? : pivottable

- Pandas can be used to create MS Excel style pivot tables.
- The fun thing about pandas `pivot_table` is you can get another point of view on your data with only one line of code.
- Most of the `pivot_table` parameters use default values, so the only mandatory parameters you must add are `data` and `index` .
  - **data** is self explanatory – it's the DataFrame you'd like to use
  - **index** is the column, grouper, array (or list of the previous) you'd like to group your data by.
  - **values (optional)** is the column you'd like to aggregate. If you do not specify this then the function will aggregate all numeric columns.

In [219]:

```
# create pivot table
pd.pivot_table(data_BM, index=['Outlet_Establishment_Year'], values= "Item_Outlet_Sales")
```

Out[219]:

Outlet_Establishment_Year	Item_Outlet_Sales
1987	2298.995256
1997	2277.844267
1999	2348.354635
2004	2438.841866
2009	1995.498739

- In the above example, the mean sales for each year is shown.
- You can also pass multiple columns to pivot table, in the next exammple we try to see mean sales not just by the year but also taking into account the **outlet size** and type of the city.

In [220]:

```
# create pivot table
pd.pivot_table(data_BM, index=['Outlet_Establishment_Year', 'Outlet_Location_Type', 'Outlet_Size'], values= "Item_Outlet_Sales")
```

Out[220]:

Outlet_Establishment_Year	Outlet_Location_Type	Outlet_Size	Item_Outlet_Sales
1987	**2**	**High**	2298.995256
1997	**0**	**Small**	2277.844267
1999	**0**	**Medium**	2348.354635
2004	**1**	**Small**	2438.841866
2009	**2**	**Medium**	1995.498739

- This makes it easier to see that Tier 1 cities have good sales irrespective of year and outlet size.
- We also notice that Tier 2 and Tier 3 cities dominate during the later years. This might mean both they are performing better or we have less data of later years.
- You can also perform multiple aggregations like mean, median, min, max etc. in a pivot table by using **aggfunc** parameter.

In [221]:

```
pd.pivot_table(data_BM, index=['Outlet_Establishment_Year', 'Outlet_Location_Type', 'Outlet_Size'], values= "Item_Outlet_Sales", aggfunc= [np.mean, np.median, min, max, np.std])
```

Out[221]:

			mean	median
			Item_Outlet_Sales	Item_Outlet_S
Outlet_Establishment_Year	Outlet_Location_Type	Outlet_Size		
1987	2	High	2298.995256	2050
1997	0	Small	2277.844267	1945
1999	0	Medium	2348.354635	1966
2004	1	Small	2438.841866	2109
2009	2	Medium	1995.498739	1655

CHAPTER : Introduction to 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

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 (https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.html).

In [222]:

```
# 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

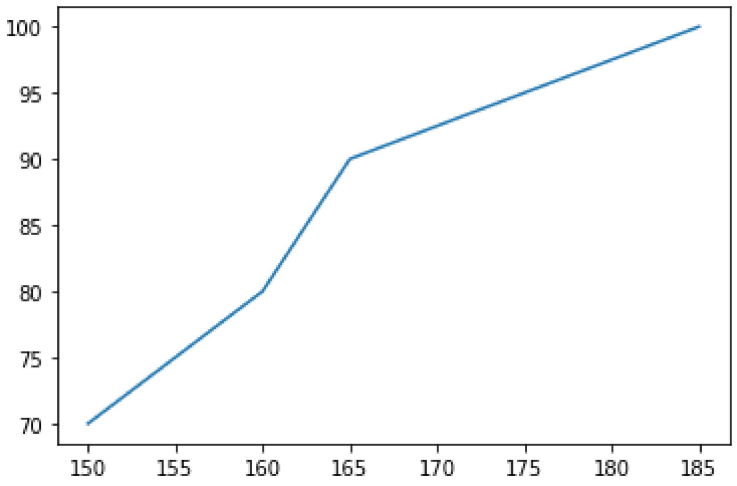
Let's create a basic plot to start working with!

In [223]:

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

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

Out[223]:

```
[<matplotlib.lines.Line2D at 0x1d5966e1c48>]
```



We pass two arrays 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

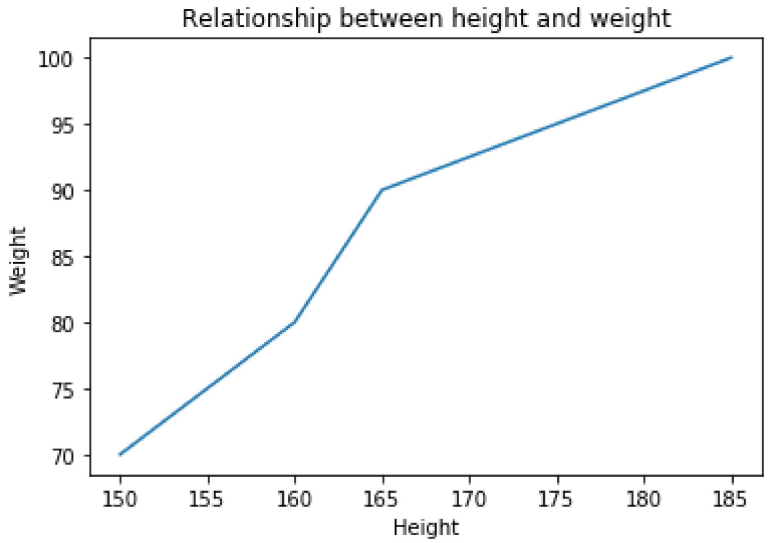
- Now that our first plot is ready, let us add the title, and name x-axis and y-axis using methods title(), xlabel() and ylabel() respectively.

In [224]:

```
# 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[224]:

```
Text(0, 0.5, 'Weight')
```

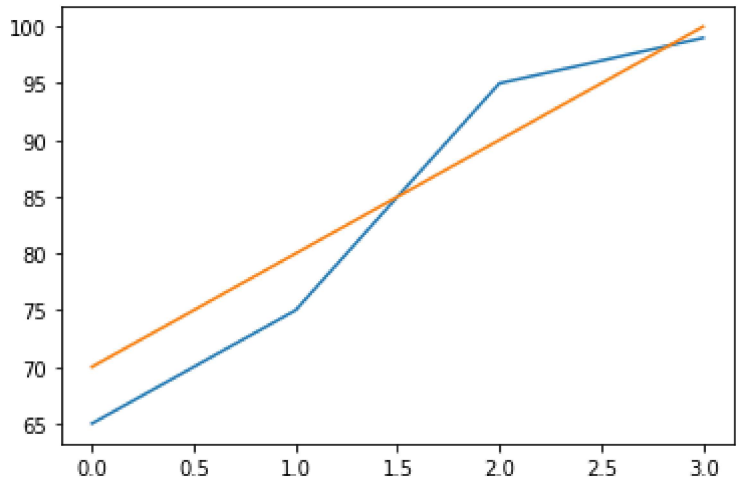


In [225]:

```
calories_burnt = [65, 75, 95, 99]

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

Out[225]:

```
[<matplotlib.lines.Line2D at 0x1d596645488>]
```



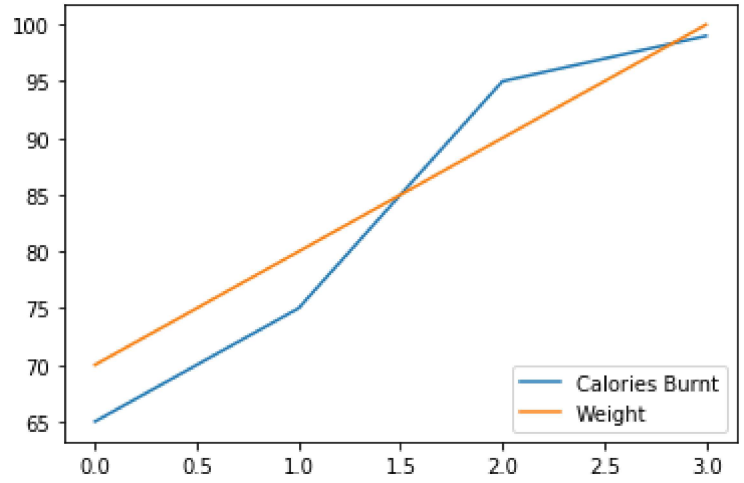
- Adding **legends** is also simple in matplotlib, you can use the `legend()` which takes **labels** and **loc** as label names and location of legend in the figure as paremeters.

In [226]:

```
# 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')
```

Out[226]:

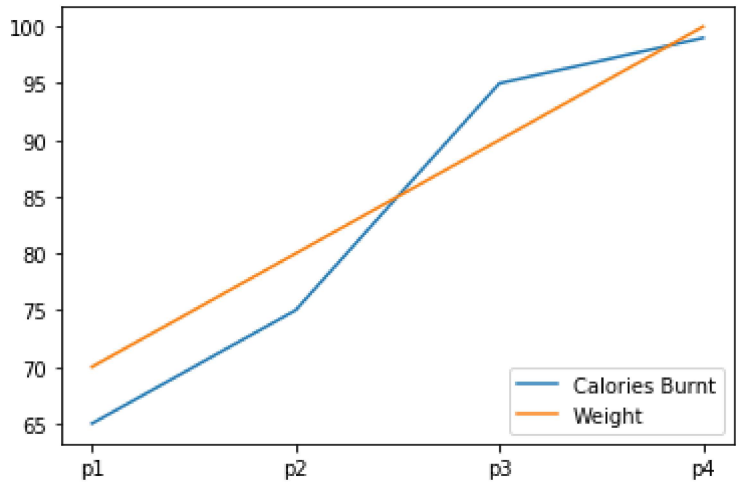
```
<matplotlib.legend.Legend at 0x1d596bdb388>
```



- 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 [227]:

```
# 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']);
```

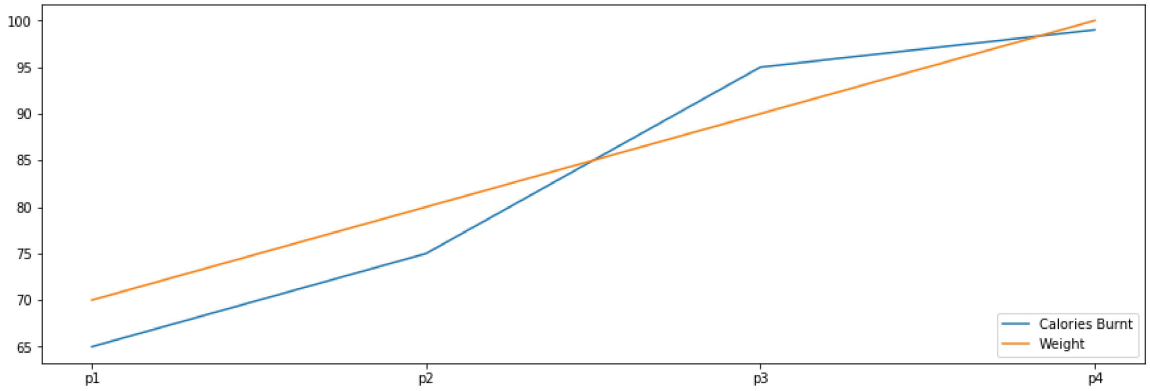


Size, Colors, Markers and Line styles

- 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 [228]:

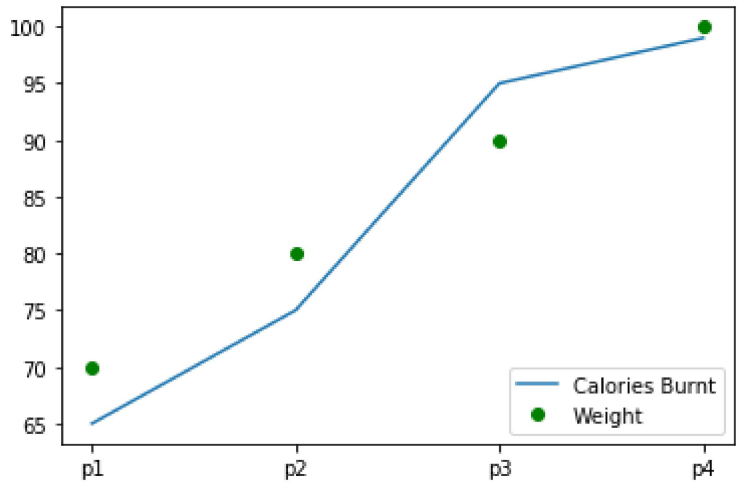
```
# figure size in inches
plt.figure(figsize=(15,5))

# 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']);
```



- 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.

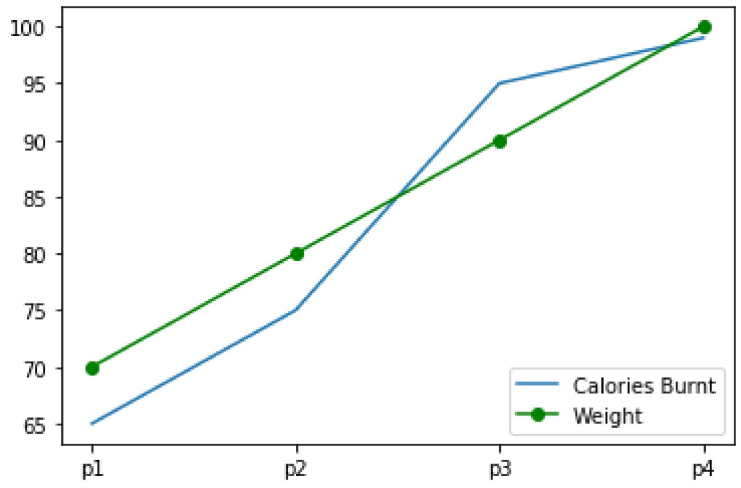
In [232]:

```
# draw the plot
plt.plot(calories_burnt)
plt.plot(weight,  'go')

# 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']);
```



In [233]:

```
# draw the plot
plt.plot(calories_burnt)
plt.plot(weight,  'go-')

# 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']);
```



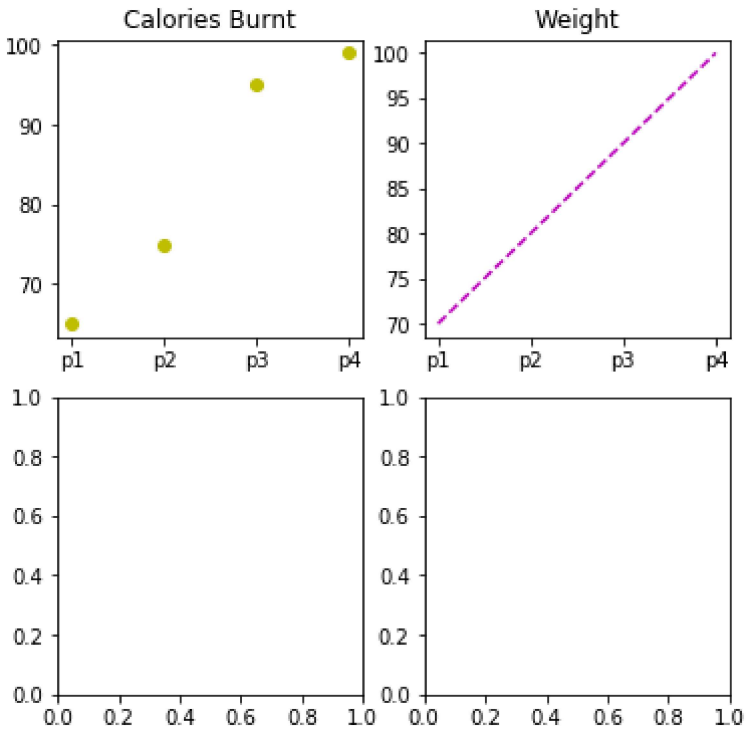
- We can also plot multiple sets of data by passing in multiple sets of arguments of X and Y axis in the `plot()` method as shown.

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 [250]:

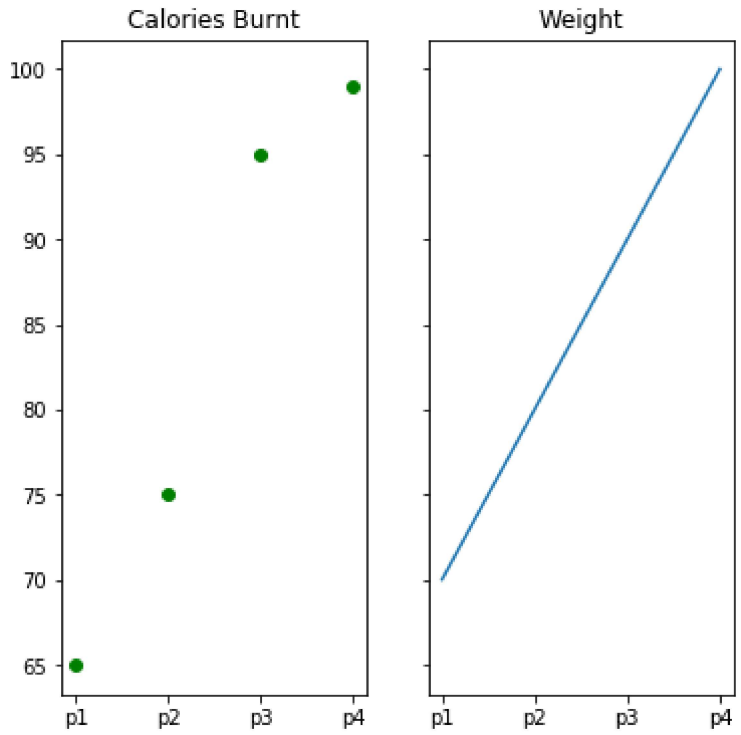
```
# create 2 plots
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(6,6))

# plot on 0 row and 0 column
ax[0,0].plot(calories_burnt,'yo') # just yellow dots

# plot on 0 row and 1 column
ax[0,1].plot(weight,'m--') #m--, means maroon,broken line. if we remove the 'm--',it'll
use default line which is a thick blue.

# 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 [251]:

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

# 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 (https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.subplots.html).

Load dataset

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

In [255]:

```
# read the dataset
bigmart = r'C:\Users\Hamzat\Downloads\ANALYTICS VIDH\data\bigmart_data.csv'
data_BM = pd.read_csv(bigmart)
# drop the null values
data_BM = data_BM.dropna(how="any")
# view the top results
data_BM.head()
```

Out[255]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outle
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	
4	NCD19	8.930	Low Fat	0.000000	Household	53.8614	
5	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	

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 [256]:

```
price_by_item = data_BM.groupby('Item_Type').Item_MRP.mean()[:10]
price_by_item
```

Out[256]:

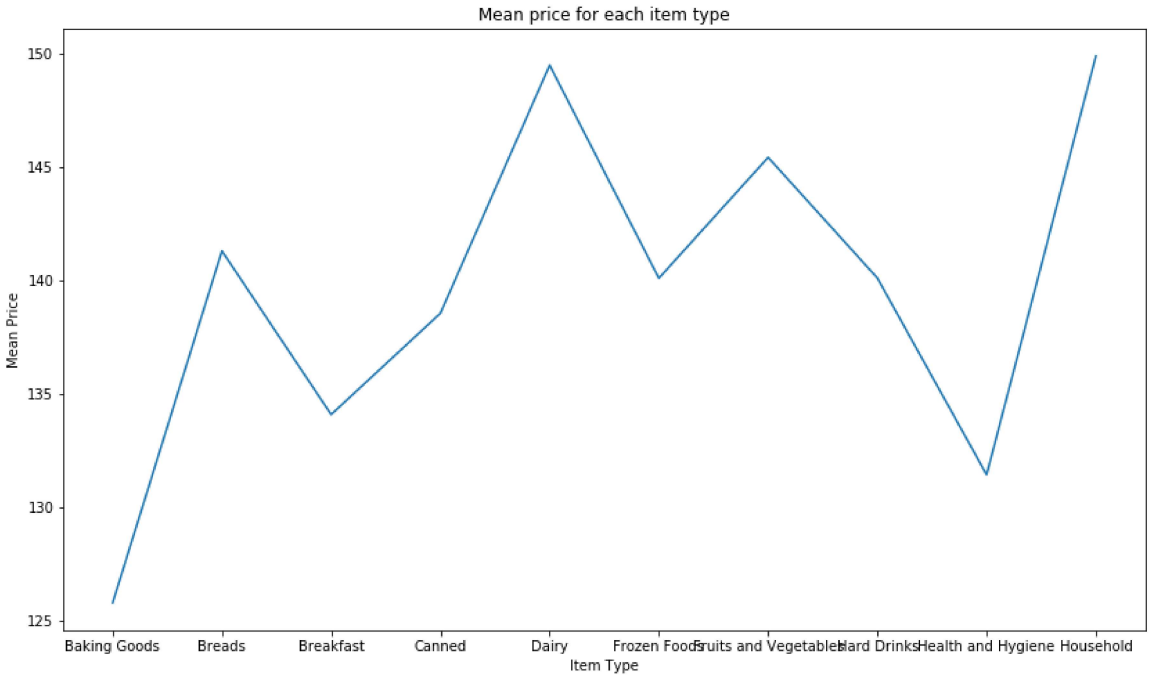
```
Item_Type
Baking Goods             125.795653
Breads                   141.300639
Breakfast                134.090683
Canned                   138.551179
Dairy                    149.481471
Frozen Foods             140.095830
Fruits and Vegetables    145.418257
Hard Drinks              140.102908
Health and Hygiene       131.437324
Household                149.884244
Name: Item_MRP, dtype: float64
```

In [257]:

```
# 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)
```

Out[257]:

```
[<matplotlib.lines.Line2D at 0x1d598627d08>]
```



4. Bar Chart

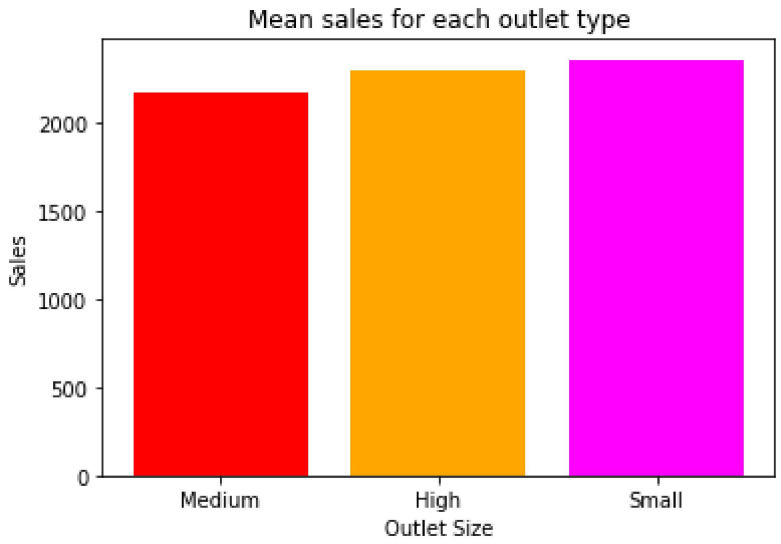
- Suppose we want to have a look at **what is the mean sales for each outlet type?**
- A bar chart is another simple type of visualization that is used for categorical variables.
- You can use `plt.bar()` instead of `plt.plot()` to create a bar chart.

In [258]:

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

# 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'])
```

Out[258]:

```
<BarContainer object of 3 artists>
```

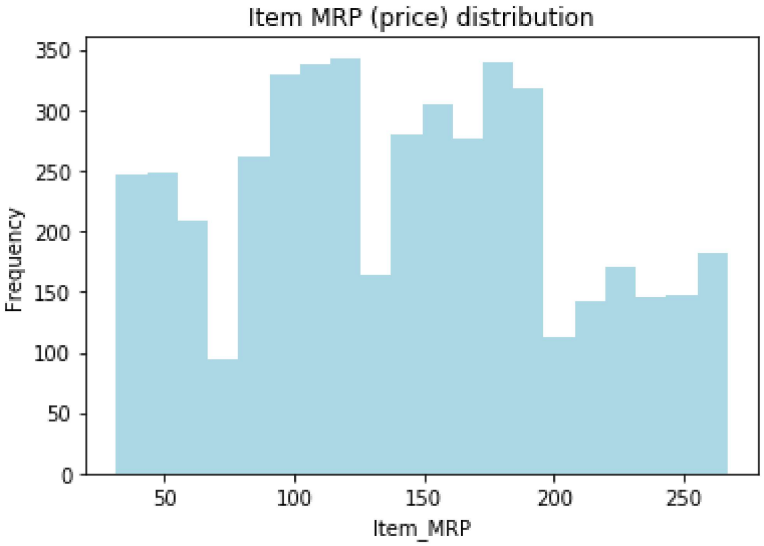


5. Histogram

- **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 (https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.hist.html).

In [259]:

```
# 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');
```

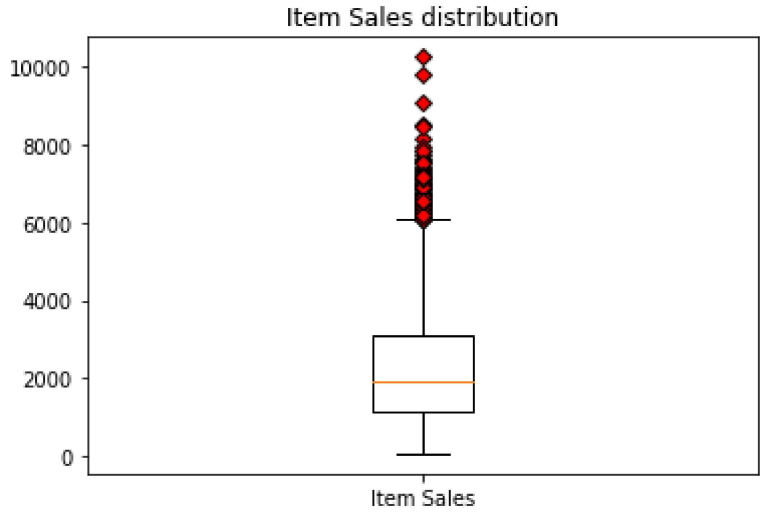


6. Box Plots

- **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 [260]:

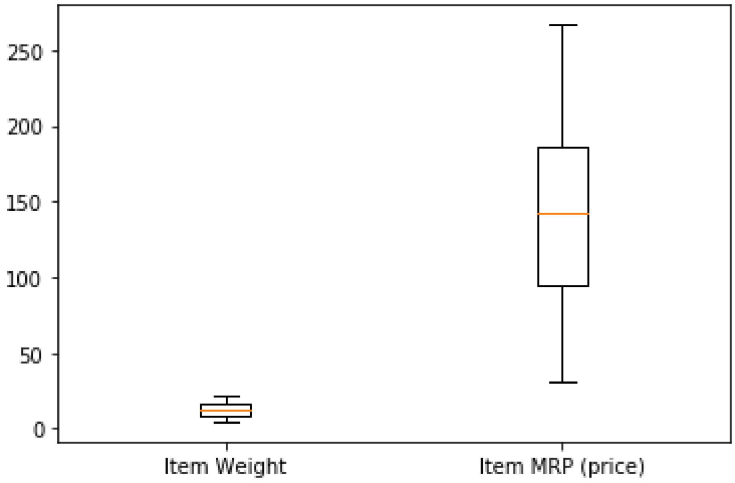
```
data = data_BM[['Item_Outlet_Sales']]

# 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 [261]:

```
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)'], flierprops=red_dia
mond);
```

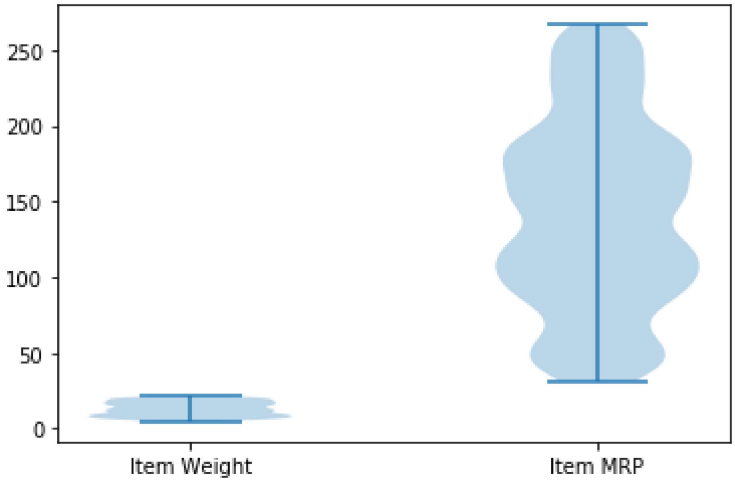


7. Violin Plots

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

In [262]:

```
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.

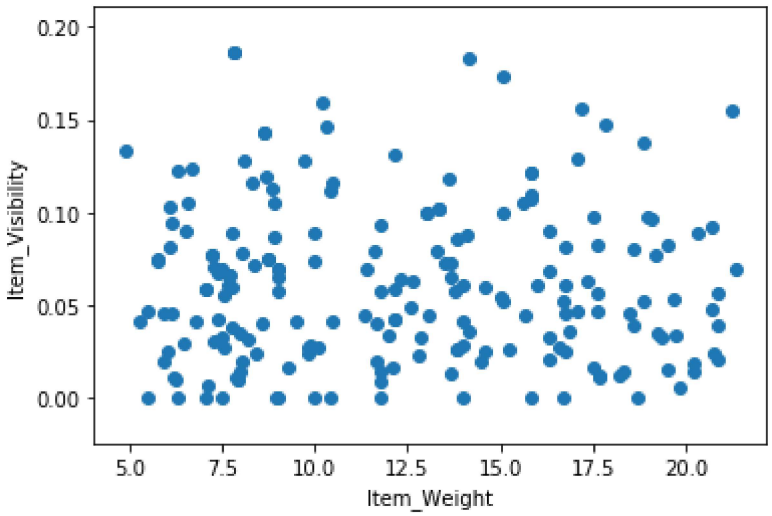
NOTE : Here, we are going to use only a subset of the data for the plots.

In [263]:

```
# 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])
```

Out[263]:

```
<matplotlib.collections.PathCollection at 0x1d599a49908>
```



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.

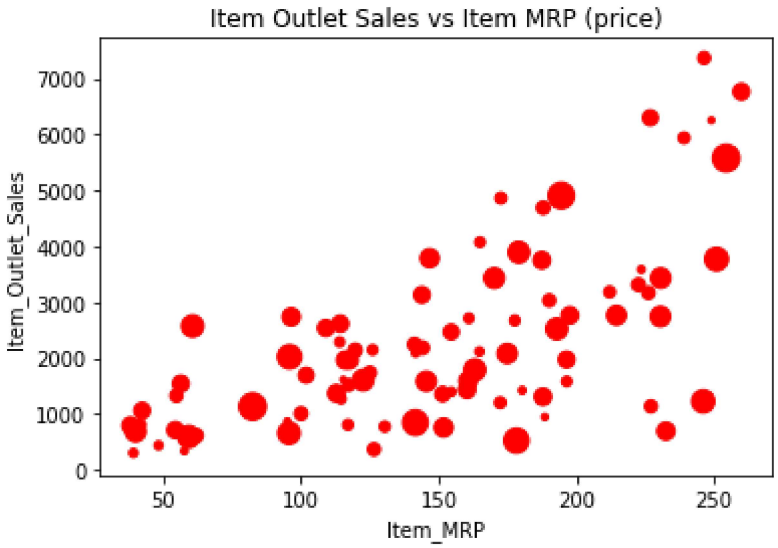
Note that we are only using a subset of data for the plots.

In [266]:

```
# 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], s=data_BM["I
tem_Visibility"][:100]*1000, c='red')
```

Out[266]:

```
<matplotlib.collections.PathCollection at 0x1d59852e088>
```



CHAPTER :

Visualization With Seaborn

- Seaborn is a Python data visualization library based on matplotlib.
- It provides a high-level interface for drawing attractive and informative statistical graphics. It provide choices for plot style and color defaults, defines simple high-level functions for common statistical plot types, and integrates with the functionality provided by Pandas DataFrames.
- The main idea of Seaborn is that it provides high-level commands to create a variety of plot types useful for statistical data exploration, and even some statistical model fitting.

Table of Contents

In [267]:

```
# importing required libraries
import seaborn as sns
sns.set()
sns.set(style="darkgrid")


import numpy as np
import pandas as pd

# importing matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

import warnings
warnings.filterwarnings("ignore")
plt.rcParams['figure.figsize']=(10,10)
```

We will keep using the Big Mart Sales Data. You can download the data from : https://datahack.analyticsvidhya.com/contest/practice-problem-big-mart-sales-iii/download/train-file (https://datahack.analyticsvidhya.com/contest/practice-problem-big-mart-sales-iii/download/train-file)

Loading dataset

In [268]:

```
# read the dataset
bigmart = r'C:\Users\Hamzat\Downloads\ANALYTICS VIDH\data\bigmart_data.csv'
data_BM = pd.read_csv(bigmart)
# drop the null values
data_BM = data_BM.dropna(how="any")
# multiply Item_Visibility by 100 to increase size
data_BM["Visibility_Scaled"] = data_BM["Item_Visibility"] * 100
# view the top results
data_BM.head()
```

Out[268]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outle
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	
4	NCD19	8.930	Low Fat	0.000000	Household	53.8614	
5	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	

1. Creating basic plots

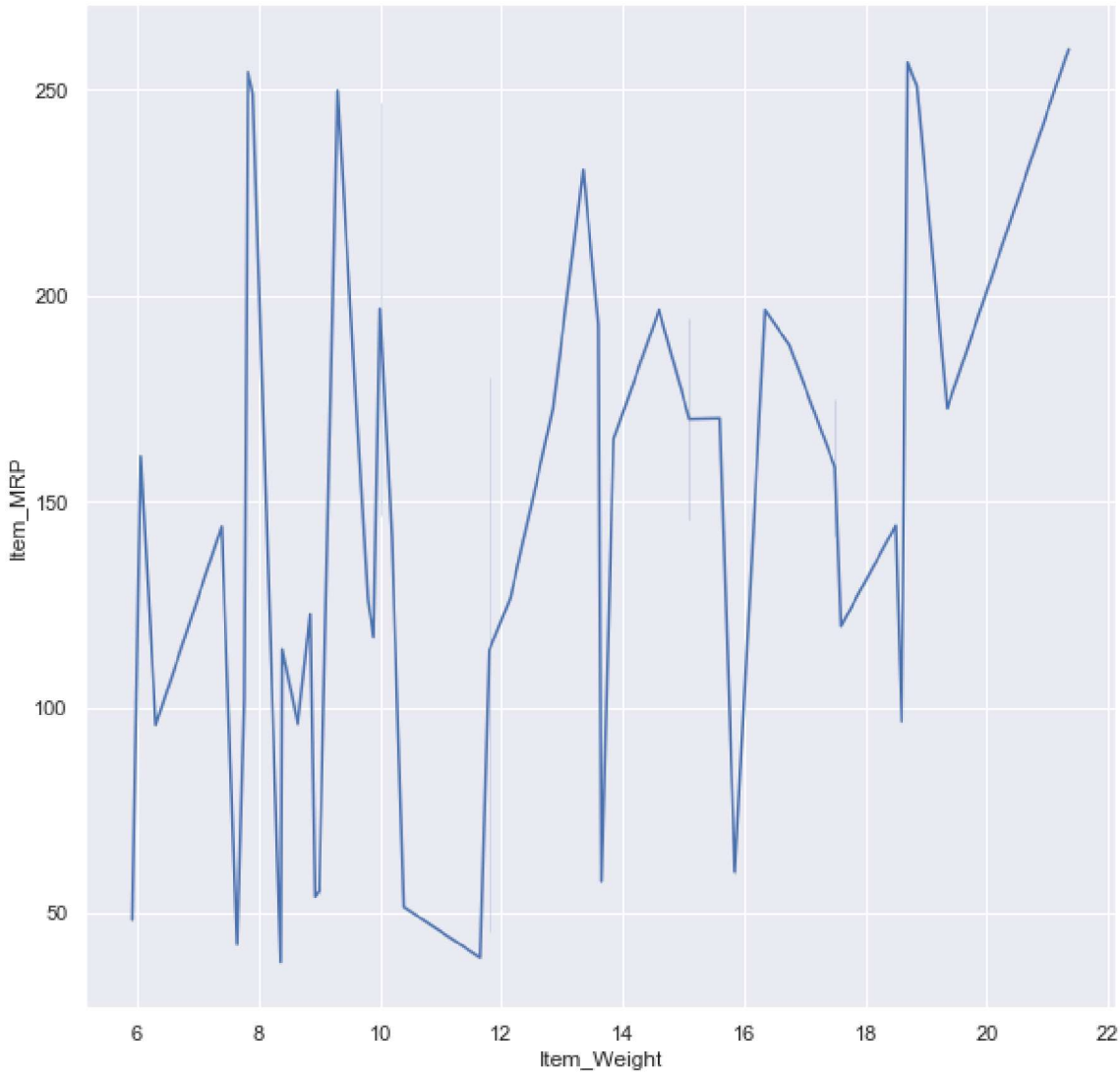
Let's have a look on how can you create some basic plots in seaborn in a single line for which multiple lines were required in matplotlib.

Line Chart

- With some datasets, you may want to understand changes in one variable as a function of time, or a similarly continuous variable.
- In seaborn, this can be accomplished by the **lineplot()** function, either directly or with **relplot()** by setting **kind="line":**

In [274]:

```
# line plot using relplot
sns.lineplot(x="Item_Weight", y="Item_MRP",data=data_BM[:50]);
```

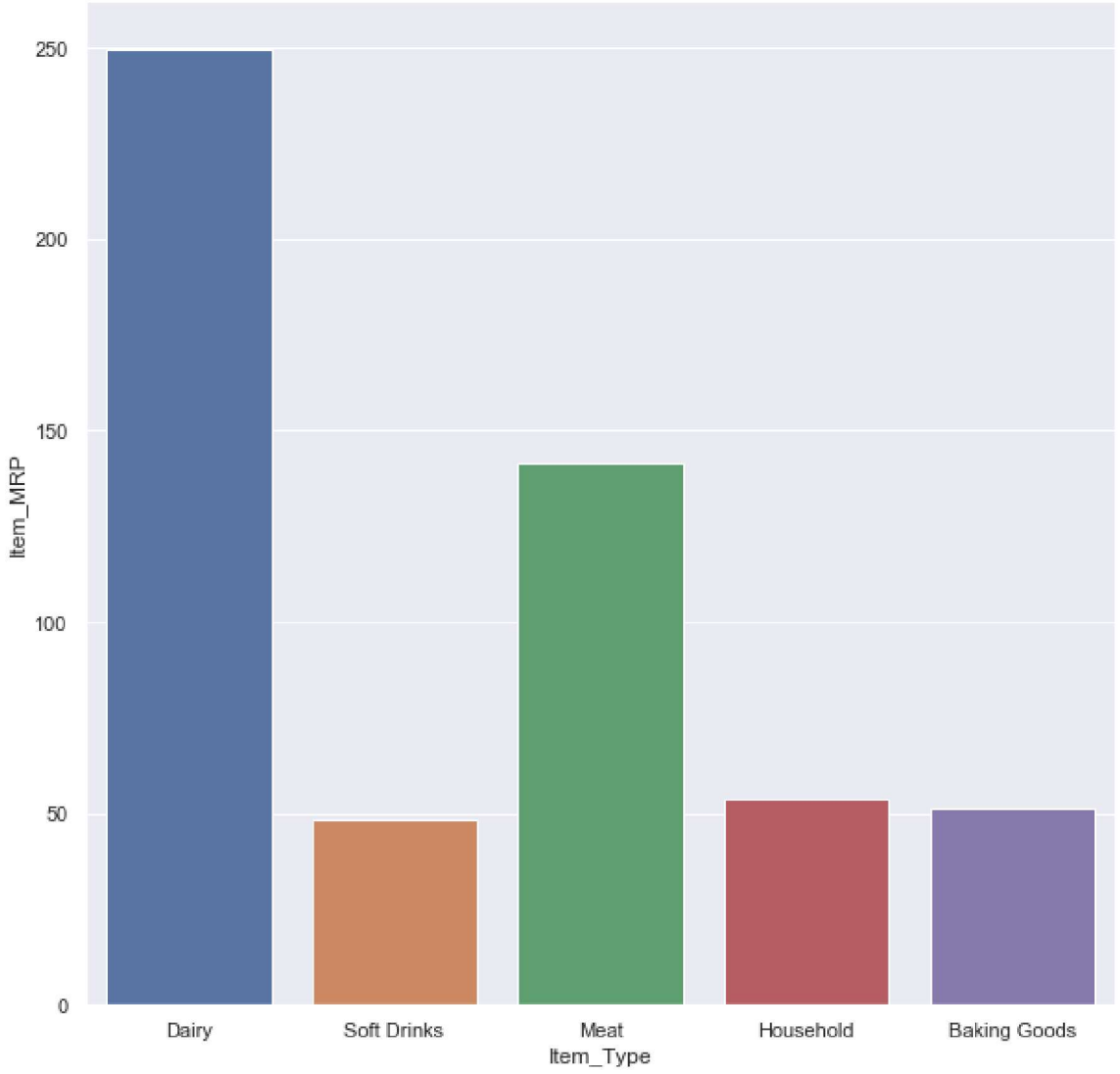


Bar Chart

- In seaborn, you can create a barchart by simply using the **barplot** function.
- Notice that to achieve the same thing in matplotlib, we had to write extra code just to group the data category wise.
- And then we had to write much more code to make sure that the plot comes out correct.

In [283]:

```
sns.barplot(x="Item_Type", y="Item_MRP", data=data_BM[:5]);
```



Histogram

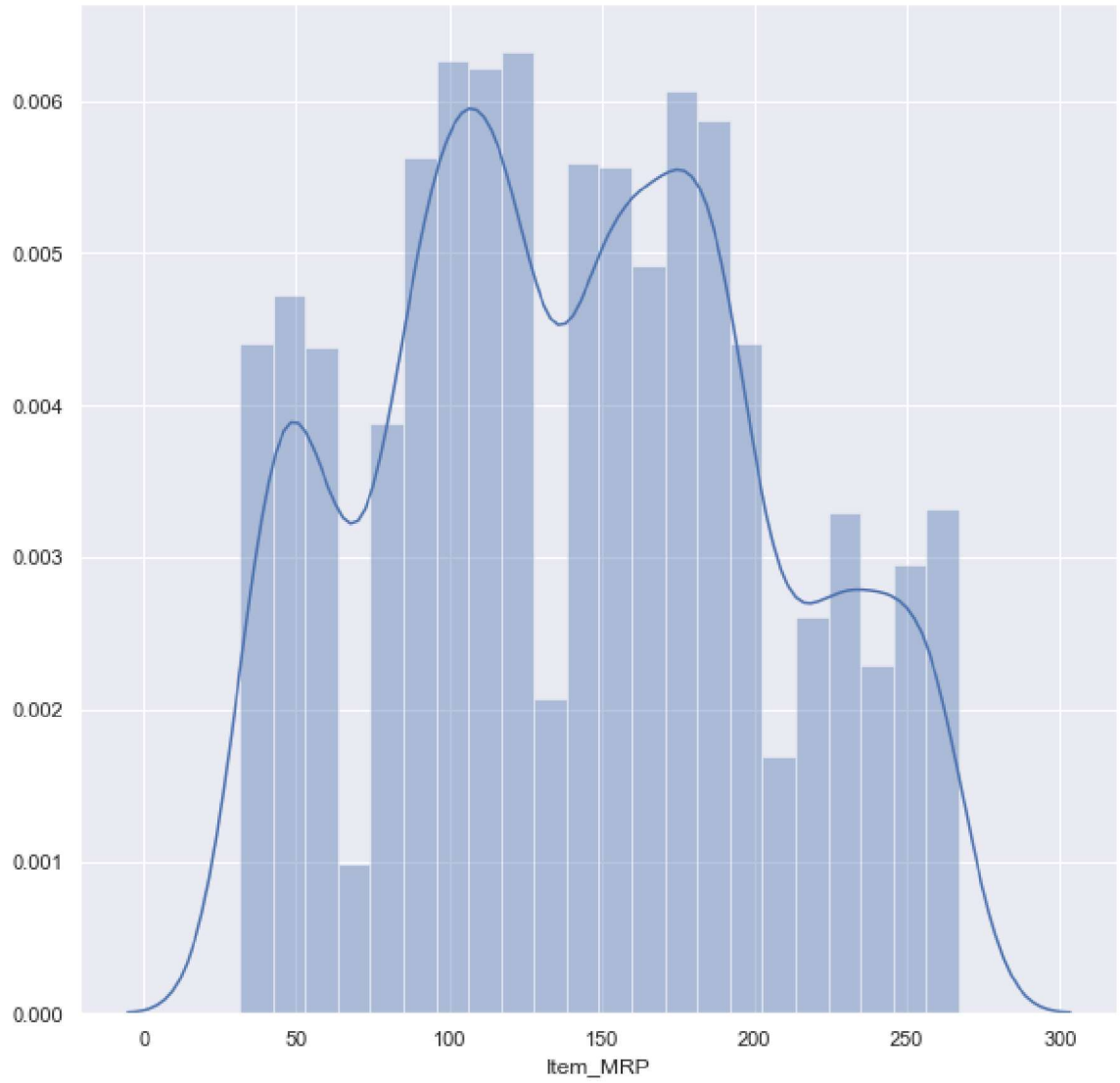
- You can create a histogram in seaborn by simply using the **distplot()**. There are multiple options that we can use which we will see further in the notebook.

In [279]:

```
sns.distplot(data_BM['Item_MRP'])
```

Out[279]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1d59bdaf148>
```



Box plots

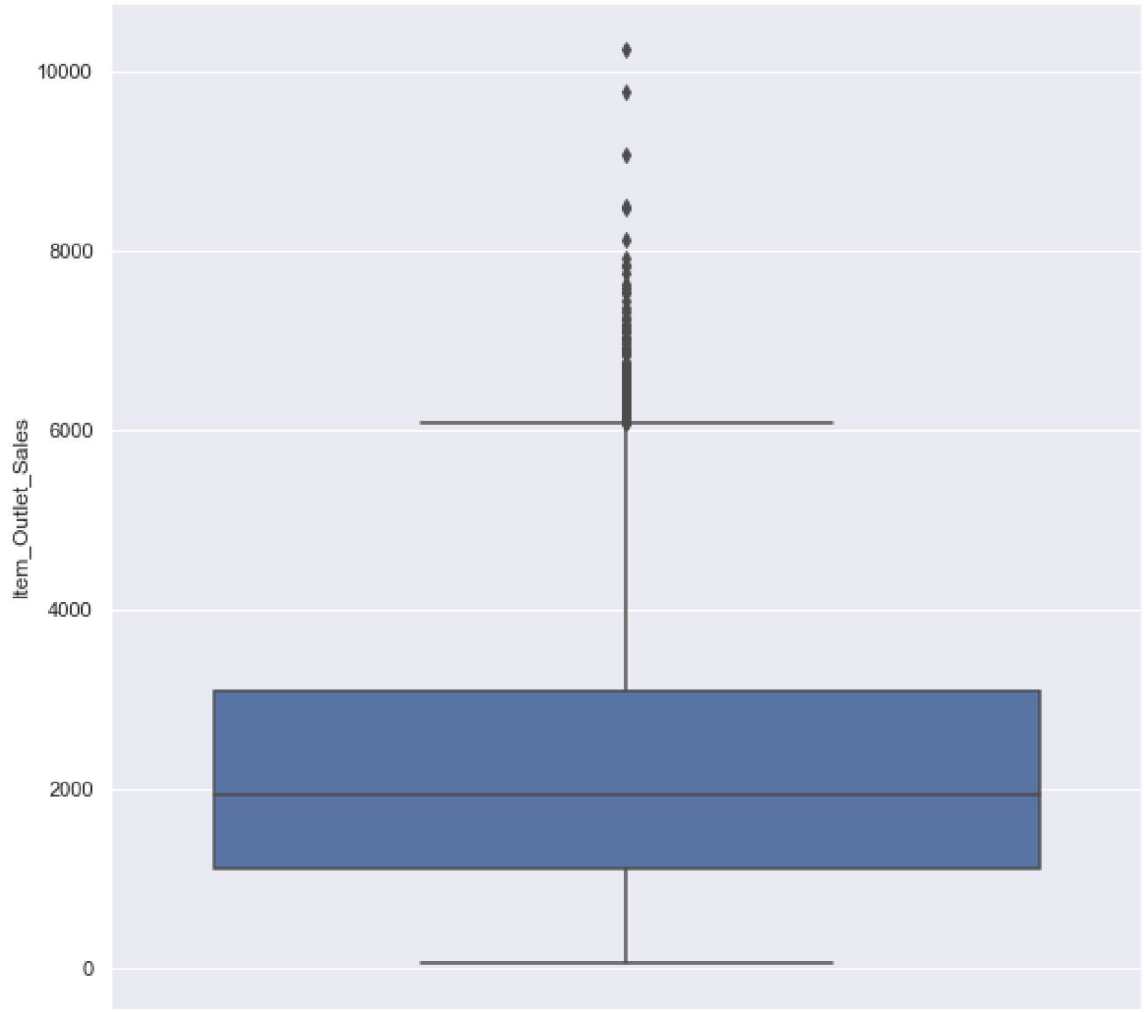
- You can use the **boxplot()** for creating boxplots in seaborn.
- Let's try to visualize the distribution of Item_Outlet_Sales of items.

In [284]:

```
sns.boxplot(data_BM['Item_Outlet_Sales'], orient='vertical')
```

Out[284]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1d59c40cf08>
```



Violin plot

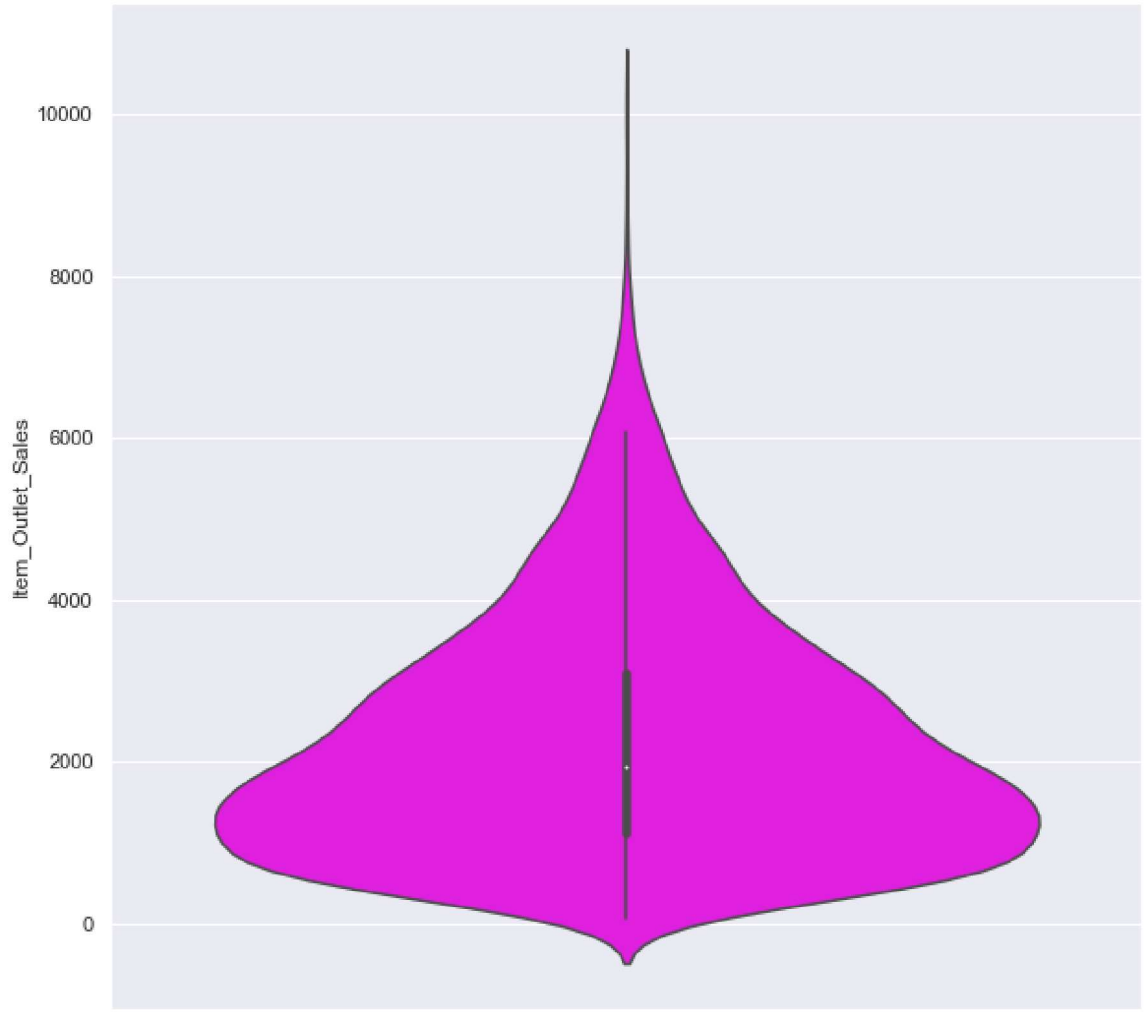
- A violin plot plays a similar role as a box and whisker plot.
- It shows the distribution of quantitative data across several levels of one (or more) categorical variables such that those distributions can be compared.
- Unlike a box plot, in which all of the plot components correspond to actual datapoints, the violin plot features a kernel density estimation of the underlying distribution.
- You can create a violinplot using the **violinplot()** in seaborn.

In [285]:

```
sns.violinplot(data_BM['Item_Outlet_Sales'], orient='vertical', color='magenta')
```

Out[285]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1d59c214c48>
```



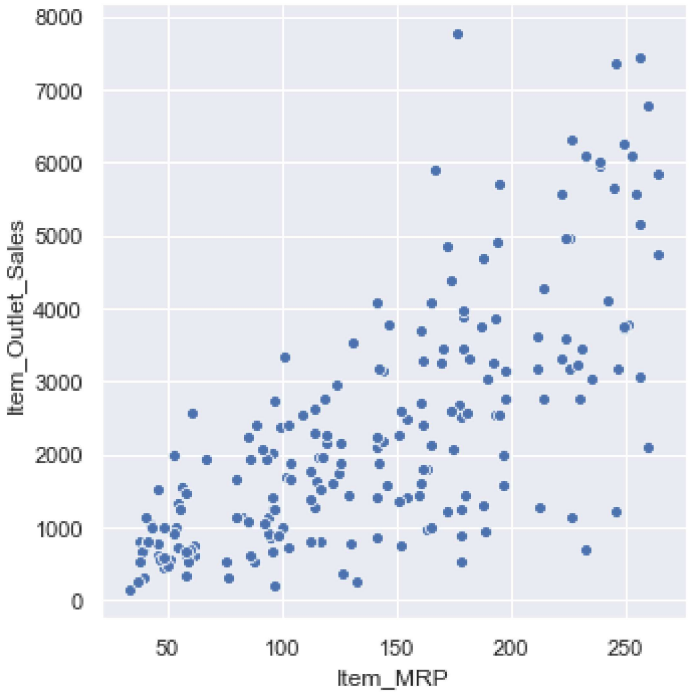
Scatter plot

- 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.
- You can use **relplot()** with the option of `kind=scatter` to plot a scatter plot in seaborn.

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

In [286]:

```
# scatter plot
sns.relplot(x="Item_MRP", y="Item_Outlet_Sales", data=data_BM[:200], kind="scatter");
```

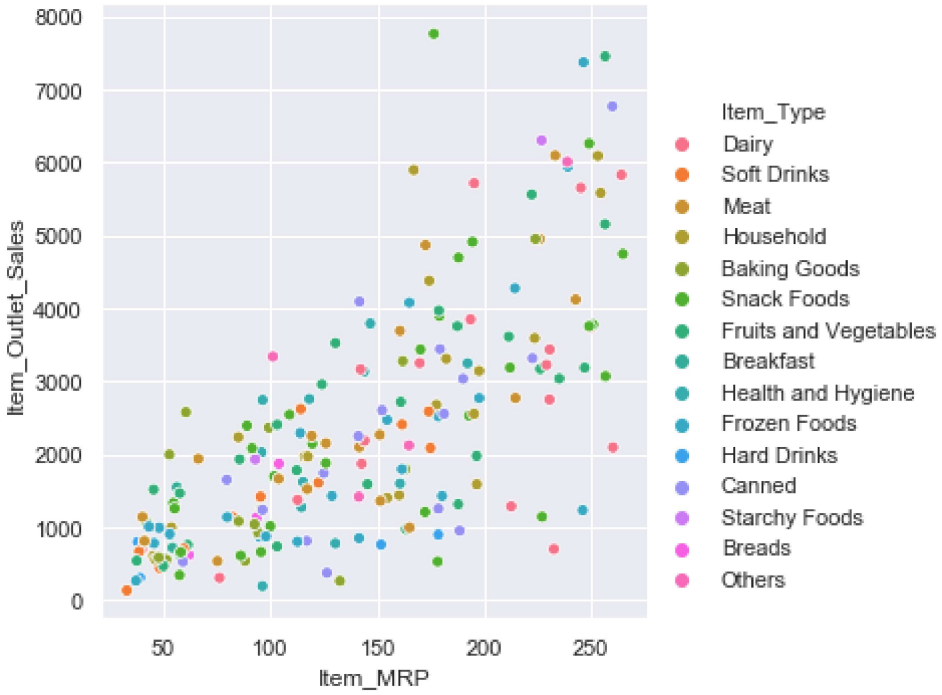


Hue semantic

We can also add another dimension to the plot by coloring the points according to a third variable. In seaborn, this is referred to as using a “hue semantic”.

In [287]:

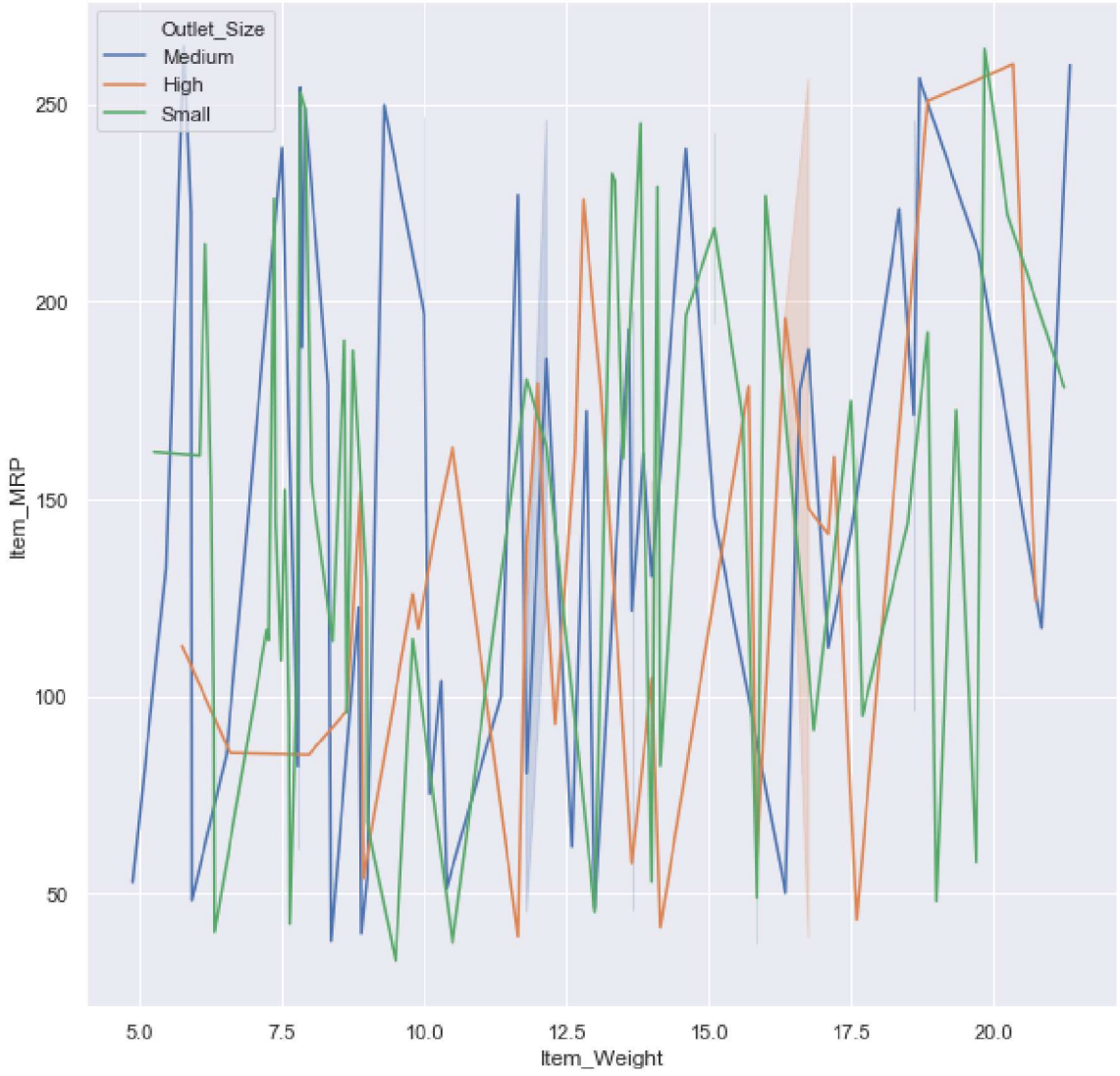
```
sns.relplot(x="Item_MRP", y="Item_Outlet_Sales", hue="Item_Type",data=data_BM[:200]);
```



- Remember the **line chart** that we created earlier? When we use **hue** semantic, we can create more complex line plots in seaborn.
- In the following example, **different line plots for different categories of the Outlet_Size** are made.

In [288]:

```
# different line plots for different categories of the Outlet_Size
sns.lineplot(x="Item_Weight", y="Item_MRP",hue='Outlet_Size',data=data_BM[:150]);
```

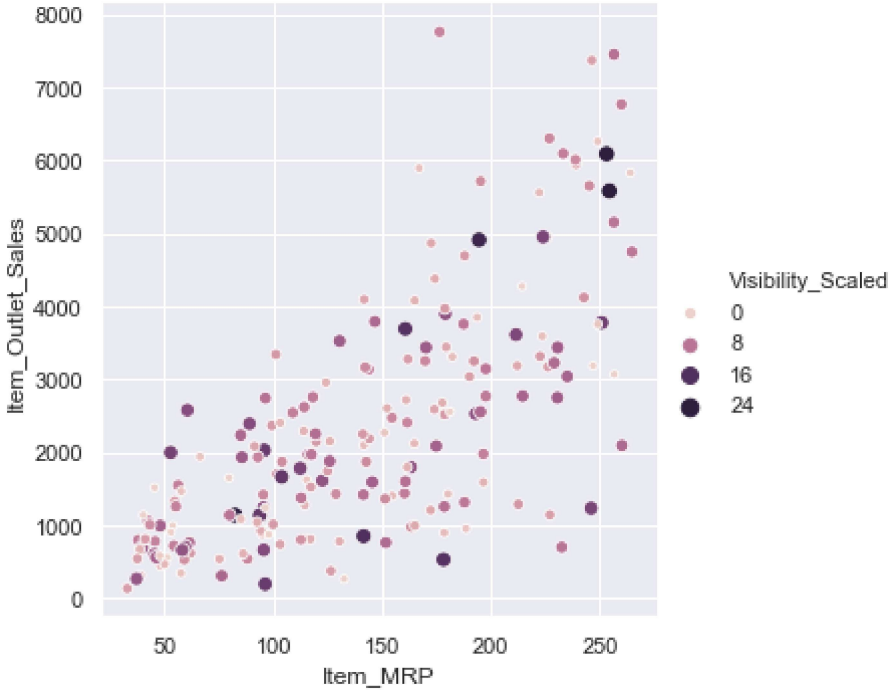


Bubble plot

- We utilize the **hue** semantic to color bubbles by their Item_Visibility and at the same time use it as size of individual bubbles.

In [289]:

```
# bubble plot
sns.relplot(x="Item_MRP", y="Item_Outlet_Sales", data=data_BM[:200], kind="scatter", si
ze="Visibility_Scaled", hue="Visibility_Scaled");
```

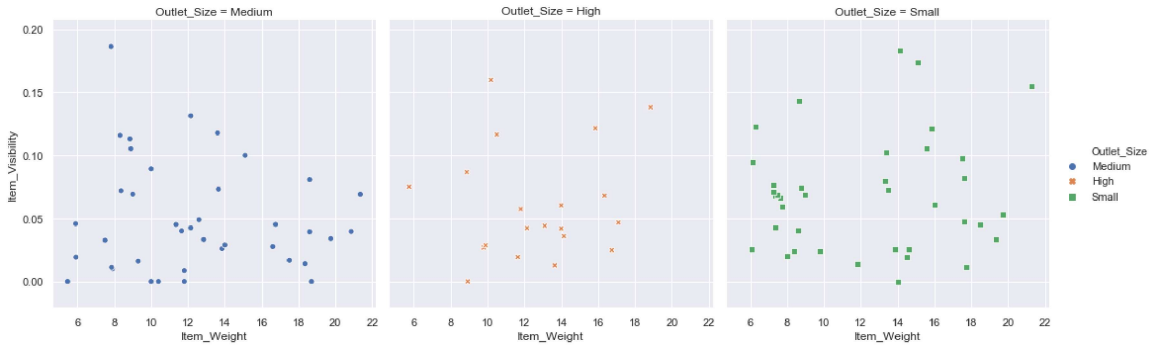


Category wise sub plot

- You can also create **plots based on category** in seaborn.
- We have created scatter plots for each Outlet_Size

In [292]:

```
# subplots for each of the category of Outlet_Size
sns.relplot(x="Item_Weight", y="Item_Visibility",hue='Outlet_Size',style='Outlet_Size',
col='Outlet_Size',data=data_BM[:100]);
```



2. Advance categorical plots in seaborn

For categorical variables we have three different families in seaborn.

- **Categorical scatterplots:**
  - stripplot() (with kind="strip"; the default)
  - swarmplot() (with kind="swarm")
- **Categorical distribution plots:**
  - boxplot() (with kind="box")
  - violinplot() (with kind="violin")
  - boxenplot() (with kind="boxen")
- **Categorical estimate plots:**
  - pointplot() (with kind="point")
  - barplot() (with kind="bar")

The default representation of the data in catplot() uses a scatterplot.

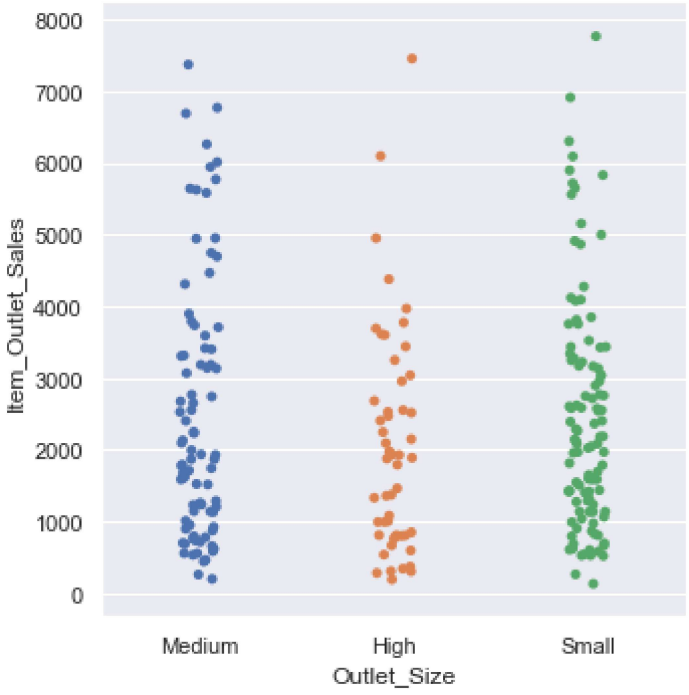
a. Categorical scatterplots

Strip plot

- Draws a scatterplot where one variable is categorical.
- You can create this by passing **kind=strip** in the `catplot()` .

In [293]:

```
sns.catplot(x="Outlet_Size", y="Item_Outlet_Sales", kind='strip',data=data_BM[:250]);
```



In [300]:

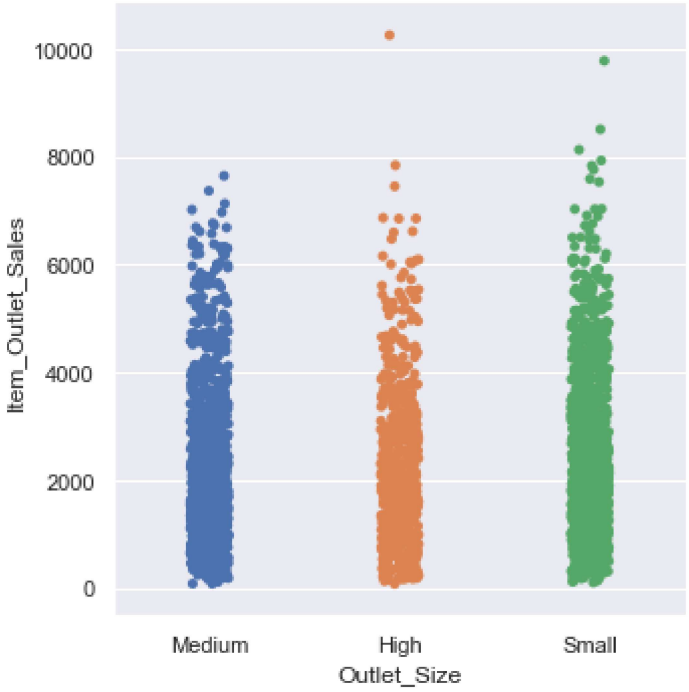
```
data_BM.shape
```

Out[300]:

```
(4650, 13)
```

In [301]:

```
sns.catplot(x="Outlet_Size", y="Item_Outlet_Sales", kind='strip',data=data_BM[:3000]);
# 3000 from the dataset
```

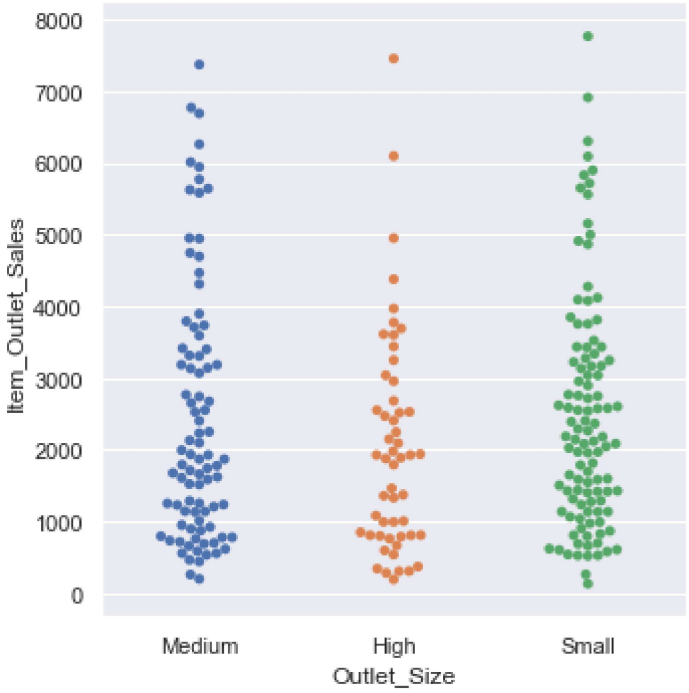


Swarm plot

- This function is similar to `stripplot()` , but the points are adjusted (only along the categorical axis) so that they don't overlap.
- This gives a better representation of the distribution of values, but it does not scale well to large numbers of observations. This style of plot is sometimes called a "beeswarm".
- You can create this by passing **kind=swarm** in the `catplot()` .

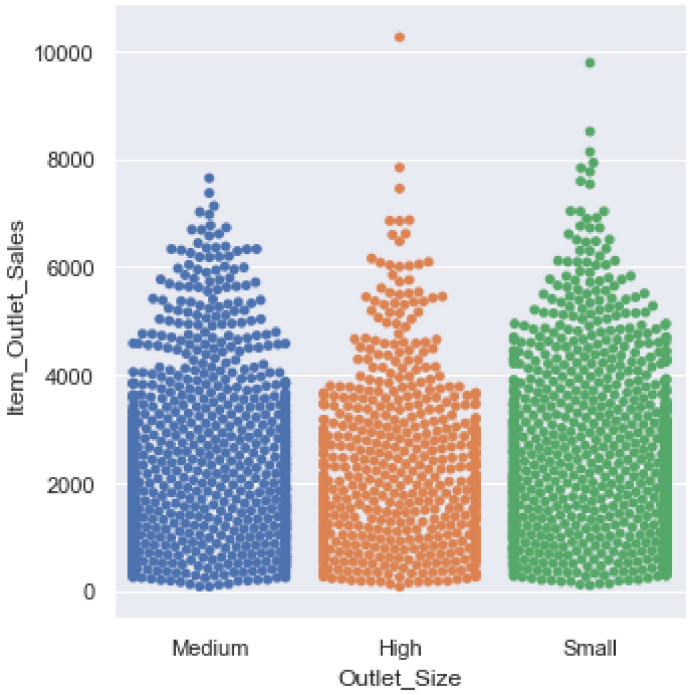
In [297]:

```
sns.catplot(x="Outlet_Size", y="Item_Outlet_Sales", kind='swarm',data=data_BM[:250]); # just visualizing 250 data from the set
```



In [302]:

```
sns.catplot(x="Outlet_Size", y="Item_Outlet_Sales", kind='swarm',data=data_BM[:3000]);
#I plotted 3000 data here
```



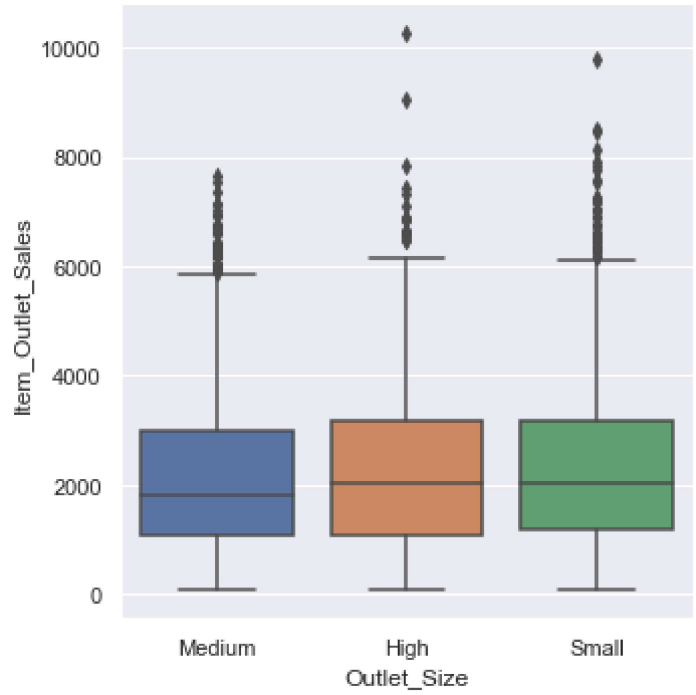
b. Categorical distribution plots

Box Plots

- 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.

In [303]:

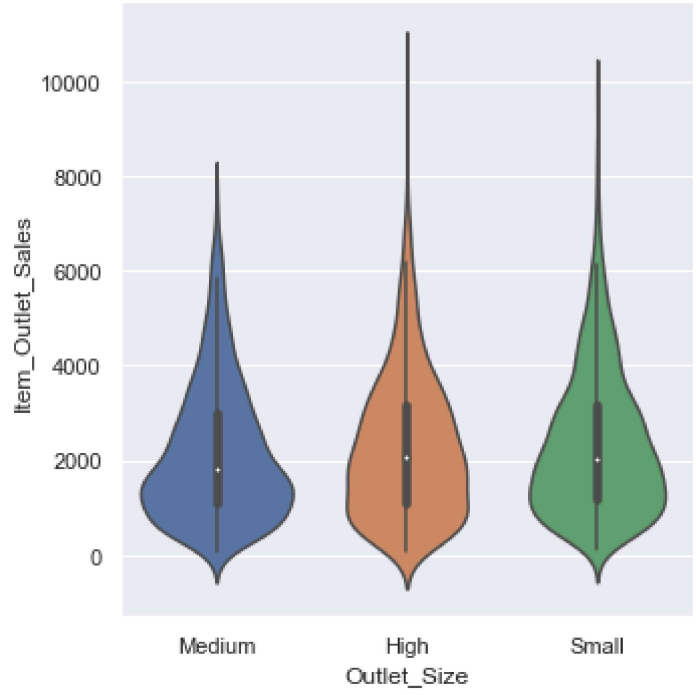
```
sns.catplot(x="Outlet_Size", y="Item_Outlet_Sales",kind="box",data=data_BM);
```



Violin Plots

In [304]:

```
sns.catplot(x="Outlet_Size", y="Item_Outlet_Sales",kind="violin",data=data_BM);
```

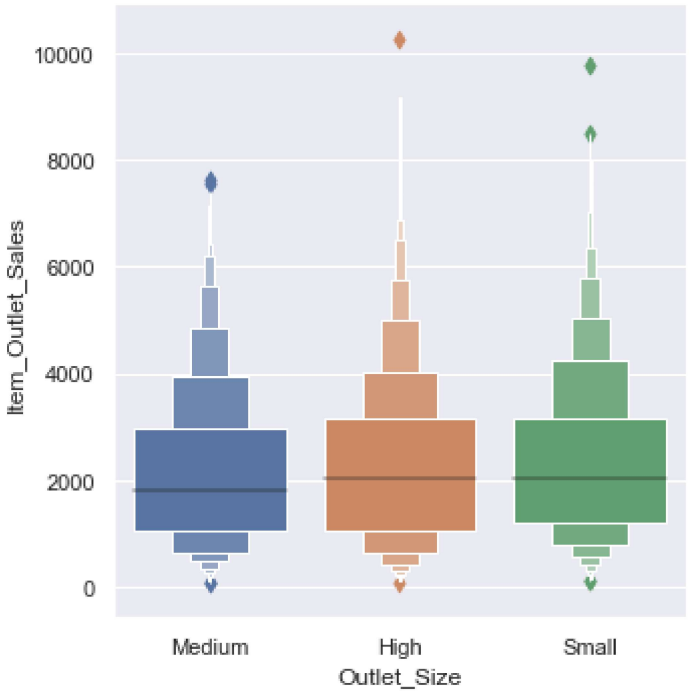


Boxen plots

- This style of plot was originally named a “letter value” plot because it shows a large number of quantiles that are defined as “letter values”.
- It is similar to a box plot in plotting a nonparametric representation of a distribution in which all features correspond to actual observations.
- By plotting more quantiles, it provides more information about the shape of the distribution, particularly in the tails.

In [305]:

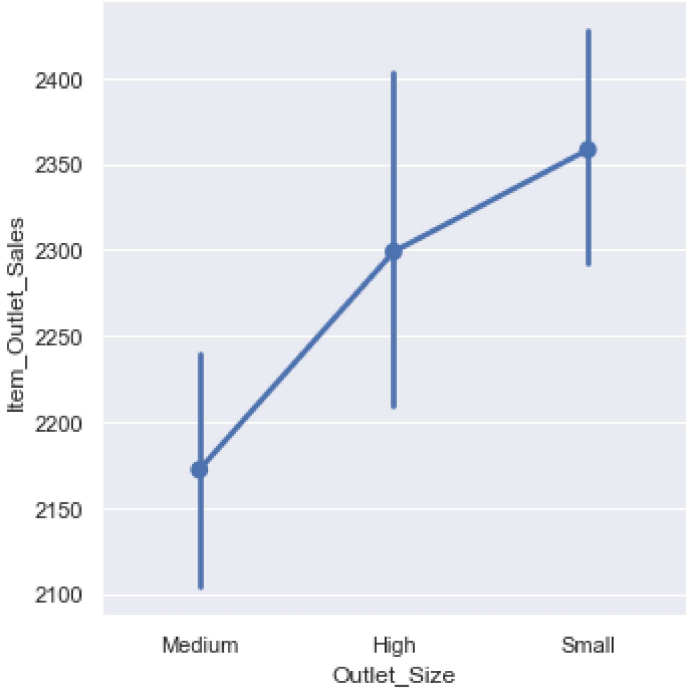
```
sns.catplot(x="Outlet_Size", y="Item_Outlet_Sales",kind="boxen",data=data_BM);
```



Point plot

In [306]:

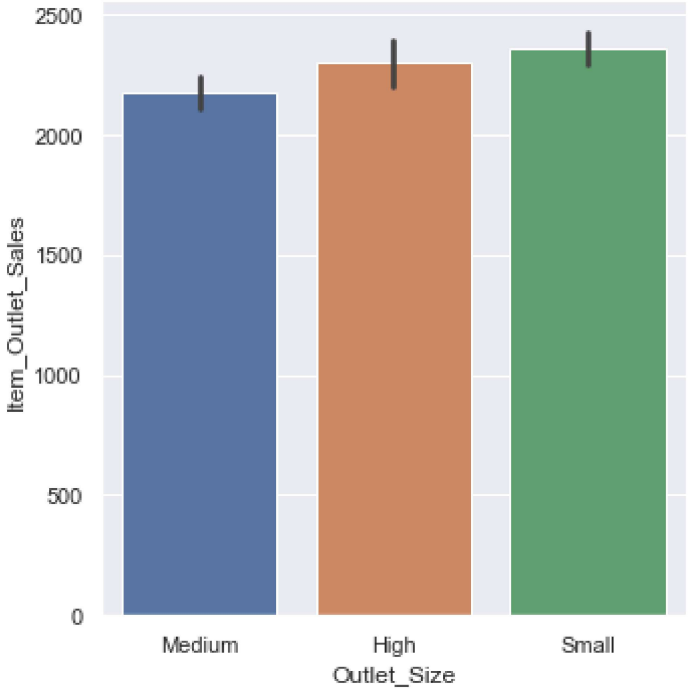
```
sns.catplot(x="Outlet_Size", y="Item_Outlet_Sales",kind="point",data=data_BM);
```



Bar plots

In [307]:

```
sns.catplot(x="Outlet_Size", y="Item_Outlet_Sales",kind="bar",data=data_BM);
```

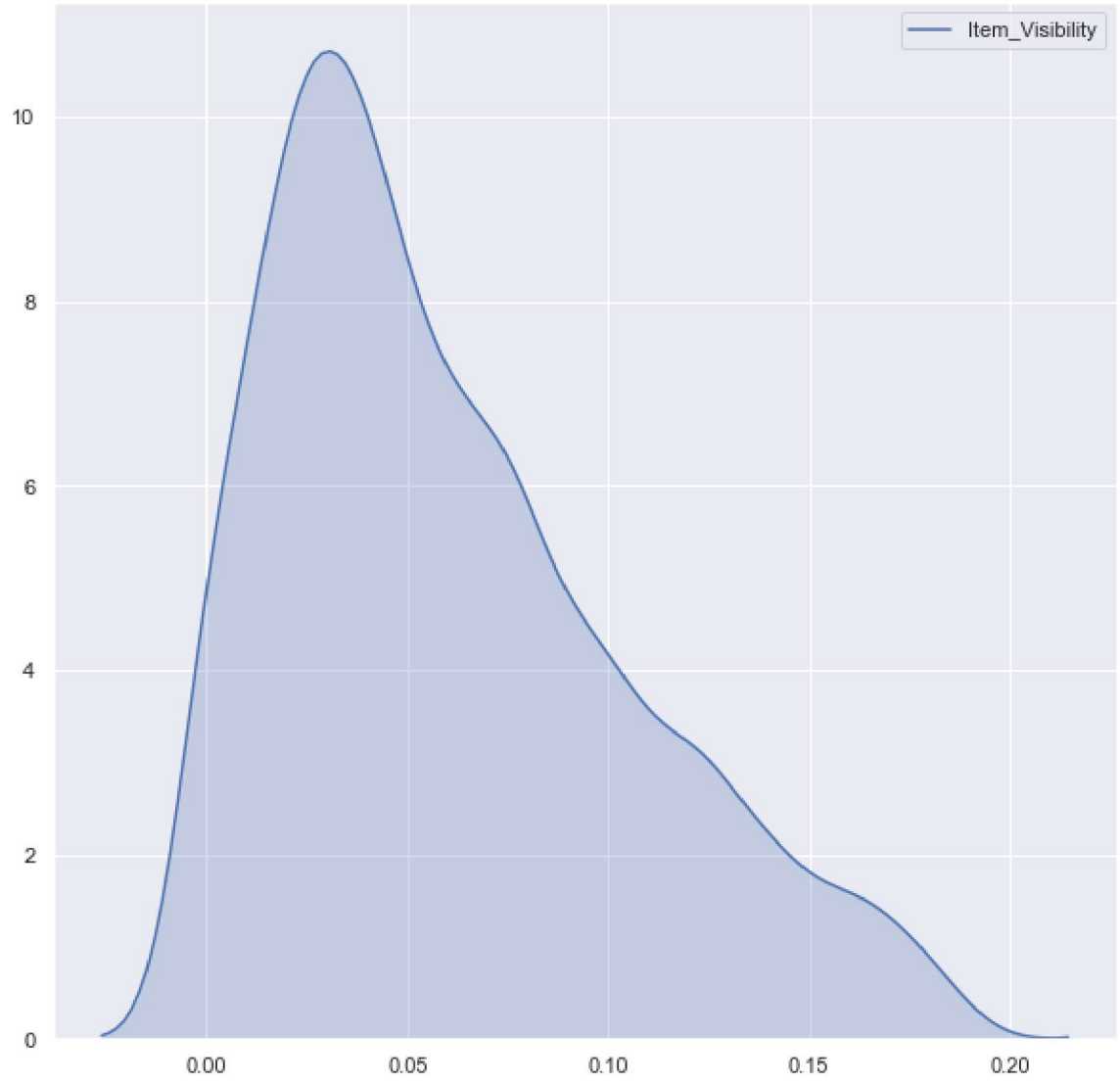


3. Density Plots

Rather than a histogram, we can get a smooth estimate of the distribution using a kernel density estimation, which Seaborn does with sns.kdeplot:

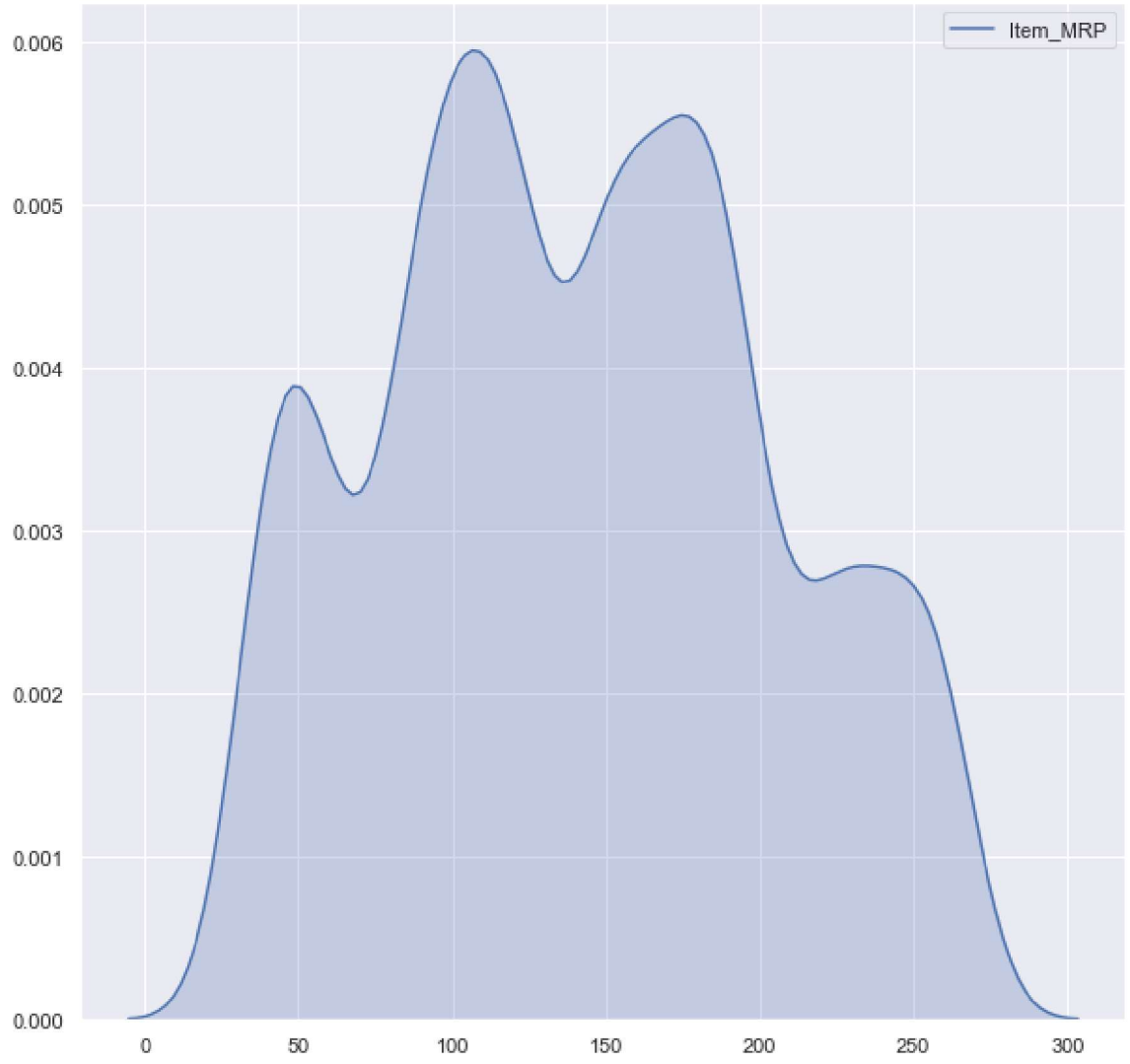
In [308]:

```
# distribution of Item Visibility
plt.figure(figsize=(10,10))
sns.kdeplot(data_BM['Item_Visibility'], shade=True);
```



In [309]:

```
# distribution of Item MRP
plt.figure(figsize=(10,10))
sns.kdeplot(data_BM['Item_MRP'], shade=True);
```

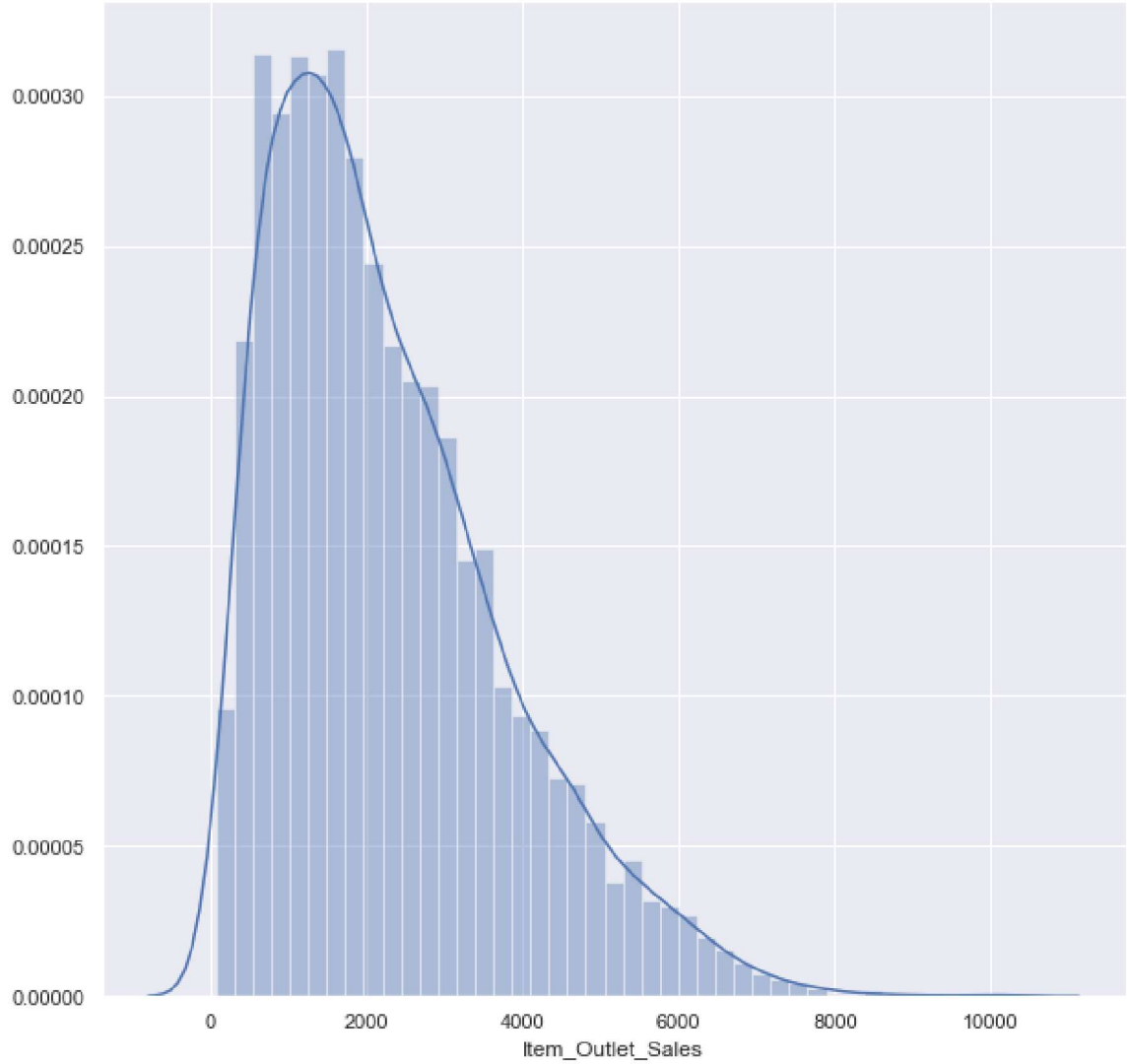


Histogram and Density Plot

Histograms and KDE can be combined using distplot:

In [310]:

```
plt.figure(figsize=(10,10))
sns.distplot(data_BM['Item_Outlet_Sales']);
```



4. Pair plots

- When you generalize joint plots to datasets of larger dimensions, you end up with pair plots. This is very useful for exploring correlations between multidimensional data, when you'd like to plot all pairs of values against each other.
- We'll demo this with the well-known Iris dataset, which lists measurements of petals and sepals of three iris species:

In [311]:

```
iris = sns.load_dataset("iris")
iris.head()
```

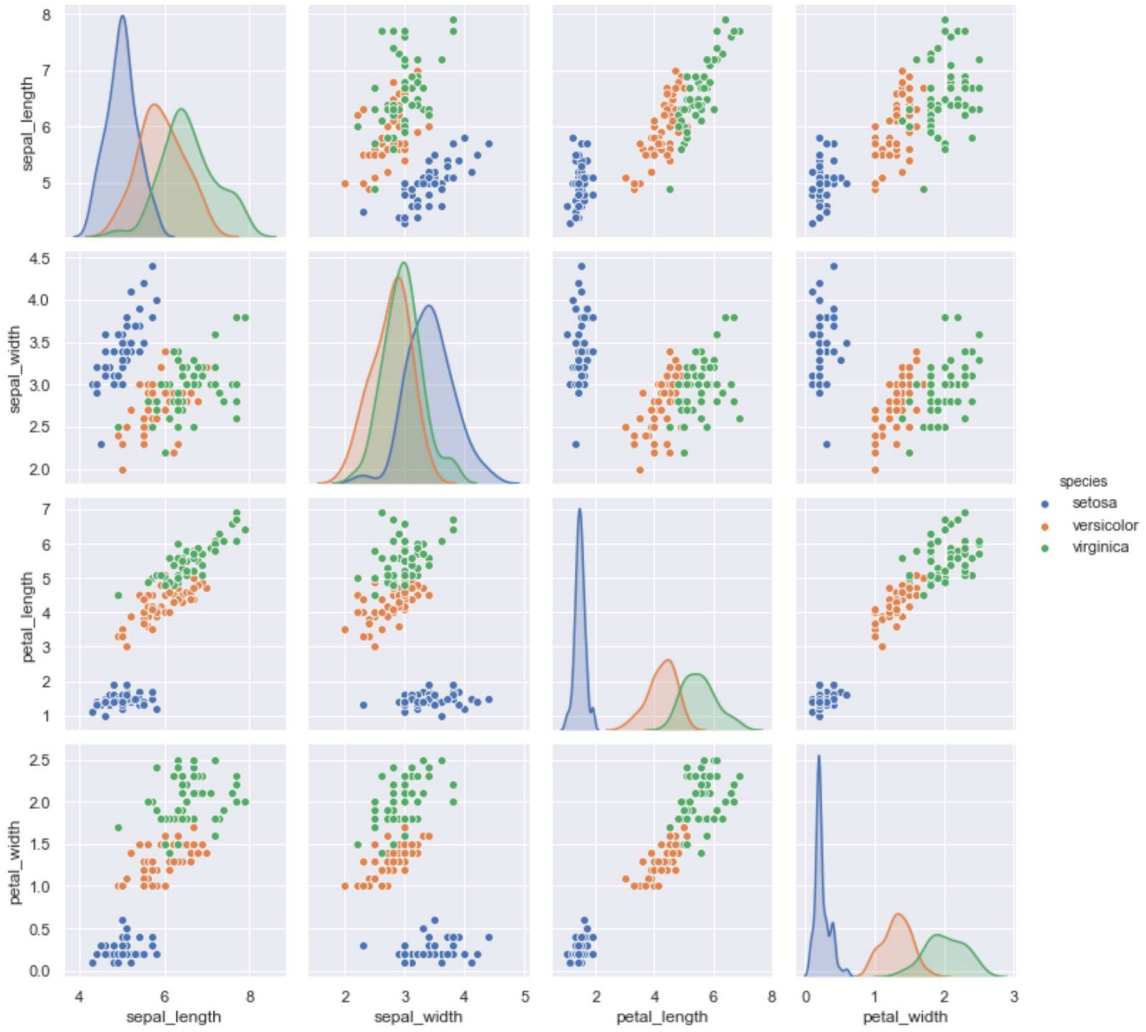
Out[311]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Visualizing the multidimensional relationships among the samples is as easy as calling sns.pairplot:

In [313]:

```
sns.pairplot(iris, hue='species', height=2.5);
```



CHAPTER :

Extract Information Using Regular Expressions (RegEx)

The first thing that i want to start off is the notion of raw string

r expression is used to create a raw string. Python raw string treats backslash (\) as a literal character.

Let us see some examples!

In [315]:

```
pwd
```

Out[315]:

```
'C:\\Users\\Hamzat'
```

In [321]:

```
# normal string vs raw string
path = "C:\desktop\nazeez"  #string
print("string:",path)
```

```
string: C:\desktop
azeez
```

In [322]:

```
path= r"C:\desktop\nazeez"  #raw string
print("raw string:",path)
```

```
raw string: C:\desktop\nazeez
```

So, it is always recommended to use raw strings while dealing with regular expressions.

Python has a built-in module to work with regular expressions called **re**. Some of the commonly used methods from the **re** module are listed below:

1.re.match(): This function checks if

2.re.search()

3.re.findall()

Let us look at each method with the help of example.

1. re.match()

The re.match function returns a match object on success and none on failure.

In [323]:

```
import re

#match a word at the beginning of a string

result = re.match('Analytics',r'Analytics Vidhya is the largest data science community
 of India')
print(result)

result_2 = re.match('largest',r'Analytics Vidhya is the largest data science community
 of India')
print(result_2)
```

```
<re.Match object; span=(0, 9), match='Analytics'>
None
```

Since output of the re.match is an object, we will use *group()* function of match object to get the matched expression.

In [324]:

```
print(result.group())  #returns the total matches
```

```
Analytics
```

2. re.search()

Matches the first occurence of a pattern in the entire string.

In [325]:

```
# search for the pattern "founded" in a given string
result = re.search('founded',r'Andrew NG founded Coursera. He also founded deeplearnin
g.ai')
print(result.group())
```

```
founded
```

3. re.findall()

It will return all the occurrences of the pattern from the string. I would recommend you to use *re.findall()* always, it can work like both *re.search()* and *re.match()*.

In [327]:

```
result = re.findall('founded',r'Andrew NG founded Coursera. He also founded deeplearnin
g.ai')
print(result)
```

```
['founded', 'founded']
```

Special sequences

1. **\A** returns a match if the specified pattern is at the beginning of the string.

In [328]:

```
str = r'Analytics Vidhya is the largest data science community of India'

x = re.findall("\AAnalytics", str)

print(x)
```

```
['Analytics']
```

This is useful in cases where you have multiple strings of text, and you have to extract the first word only, given that first word is 'Analytics'.

If you would try to find some other word, then it will return an empty list as shown below.

In [329]:

```
str = r'Analytics Vidhya is the largest Analytics community of India'

x = re.findall("\AVidhya", str)

print(x)
```

```
[]
```

1. **\b** returns a match where the specified pattern is at the beginning or at the end of a word.

In [330]:

```
#Check if there is any word that ends with "est"
x = re.findall(r"est\b", str)
print(x)
```

```
['est']
```

It returns the last three characters of the word "largest".

1. **\B** returns a match where the specified pattern is present, but NOT at the beginning (or at the end) of a word.

In [331]:

```
str = r'Analytics Vidhya is the largest data science community of India'

x = re.findall(r"\Ben", str)

print(x)
```

```
['en']
```

1. **\d** returns a match where the string contains digits (numbers from 0-9)

In [332]:

```
str = "2 million monthly visits in Jan'19."

# Check if the string contains any digits (numbers from 0-9):
# adding '+' after '\d' will continue to extract digits till encounters a space
x = re.findall("\d+", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['2', '19']
Yes, there is at least one match!
```

We can infer that **\d+** repeats one or more occurences of **\d** till the non maching character is found where as **\d** does character wise comparison.

1. **\D** returns a match where the string does not contain any digit.

In [333]:

```
str = "2 million monthly visits in Jan'19."

#Check if the word character does not contain any digits (numbers from 0-9):
x = re.findall("\D", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
[' ', 'm', 'i', 'l', 'l', 'i', 'o', 'n', ' ', 'm', 'o', 'n', 't', 'h',
'l', 'y', ' ', 'v', 'i', 's', 'i', 't', 's', ' ', 'i', 'n', ' ', 'J', 'a',
'n', "'", '.']
Yes, there is at least one match!
```

In [334]:

```
str = "2 million monthly visits'19"

#Check if the word does not contain any digits (numbers from 0-9):

x = re.findall("\D+", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
[" million monthly visits'"]
Yes, there is at least one match!
```

1. \w helps in extraction of alphanumeric characters only (characters from a to Z, digits from 0-9, and the underscore _ character)

In [335]:

```
str = "2 million monthly visits!"

#returns a match at every word character (characters from a to Z, digits from 0-9, and
 the underscore _ character)

x = re.findall("\w",str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['2', 'm', 'i', 'l', 'l', 'i', 'o', 'n', 'm', 'o', 'n', 't', 'h', 'l',
'y', 'v', 'i', 's', 'i', 't', 's']
Yes, there is at least one match!
```

In [336]:

```
str = "2 million monthly visits!"

#returns a match at every word (characters from a to Z, digits from 0-9, and the unders
core _ character)

x = re.findall("\w+",str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['2', 'million', 'monthly', 'visits']
Yes, there is at least one match!
```

1. **\W** returns match at every non alphanumeric character.

In [337]:

```
str = "2 million monthly visits9!"

#returns a match at every NON word character (characters NOT between a and Z. Like "!",
"?" white-space etc.):

x = re.findall("\W", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
[' ', ' ', ' ', '!']
Yes, there is at least one match!
```

Metacharacters

Metacharacters are characters with a special meaning

1. **(.)** matches any character (except newline character)

In [338]:

```
str = "rohan and rohit recently published a research paper!"

#Search for a string that starts with "ro", followed by three (any) characters

x = re.findall("ro.", str)
x2 = re.findall("ro...", str)

print(x)
print(x2)
```

```
['roh', 'roh']
['rohan', 'rohit']
```

1. **(^)** starts with

In [339]:

```
str = "Data Science"

#Check if the string starts with 'Data':
x = re.findall("^Data", str)

if (x):
  print("Yes, the string starts with 'Data'")
else:
  print("No match")

#print(x)
```

```
Yes, the string starts with 'Data'
```

In [340]:

```
# try with a different string
str2 = "Big Data"

#Check if the string starts with 'Data':
x2 = re.findall("^Data", str2)

if (x2):
  print("Yes, the string starts with 'data'")
else:
  print("No match")

#print(x2)
```

```
No match
```

1. **($)** ends with

In [343]:

```
str = "Data Science"

#Check if the string ends with 'Science':

x = re.findall("Science$", str)

if (x):
  print("Yes, the string ends with 'Science'")

else:
  print("No match")

#print(x)
```

```
Yes, the string ends with 'Science'
```

1. (*) matches for zero or more occurences of the pattern to the left of it

In [345]:

```
str = "easy easssy eay ey"

#Check if the string contains "ea" followed by 0 or more "s" characters and ending with
y
x = re.findall("eas*y", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['easy', 'easssy', 'eay']
Yes, there is at least one match!
```

1. **(+)** matches one or more occurences of the pattern to the left of it

In [346]:

```
#Check if the string contains "ea" followed by 1 or more "s" characters and ends with y
x = re.findall("eas+y", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['easy', 'easssy']
Yes, there is at least one match!
```

1. **(?)** matches zero or one occurrence of the pattern left to it.

In [347]:

```
x = re.findall("eas?y",str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['easy', 'eay']
Yes, there is at least one match!
```

1. **(|)** either or

In [348]:

```
str = "Analytics Vidhya is the largest data science community of India"

#Check if the string contains either "data" or "India":

x = re.findall("data|India", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['data', 'India']
Yes, there is at least one match!
```

In [349]:

```
# try with a different string
str = "Analytics Vidhya is one of the largest data science communities"

#Check if the string contains either "data" or "India":

x = re.findall("data|India", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['data']
Yes, there is at least one match!
```

Sets

1. A set is a bunch of characters inside a pair of square brackets [ ] with a special meaning.

In [350]:

```
str = "Analytics Vidhya is the largest data science community of India"

#Check for the characters y, d, or h, in the above string
x = re.findall("[ydh]", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['y', 'd', 'h', 'y', 'h', 'd', 'y', 'd']
Yes, there is at least one match!
```

In [351]:

```
str = "Analytics Vidhya is the largest data science community of India"

#Check for the characters between a and g, in the above string
x = re.findall("[a-g]", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['a', 'c', 'd', 'a', 'e', 'a', 'g', 'e', 'd', 'a', 'a', 'c', 'e', 'c',
'e', 'c', 'f', 'd', 'a']
Yes, there is at least one match!
```

Let's solve a problem.

In [352]:

```
str = "Mars' average distance from the Sun is roughly 230 million km and its orbital pe
riod is 687 (Earth) days."

# extract the numbers starting with 0 to 4 from in the above string
x = re.findall(r"\b[0-4]\d+", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['230']
Yes, there is at least one match!
```

1. **[^]** Check whether string has other characters mentioned after ^

In [353]:

```
str = "Analytics Vidhya is the largest data sciece community of India"

#Check if every word character has characters than y, d, or h

x = re.findall("[^ydh]", str)

print(x)

if (x):
  print("Yes, there is at least one match!")
else:
  print("No match")
```

```
['A', 'n', 'a', 'l', 't', 'i', 'c', 's', ' ', 'V', 'i', 'a', ' ', 'i',
's', ' ', 't', 'e', ' ', 'l', 'a', 'r', 'g', 'e', 's', 't', ' ', 'a', 't',
'a', ' ', 's', 'c', 'i', 'e', 'c', 'e', ' ', 'c', 'o', 'm', 'm', 'u', 'n',
'i', 't', ' ', 'o', 'f', ' ', 'I', 'n', 'i', 'a']
Yes, there is at least one match!
```

1. **[a-zA-Z0-9]** : Check whether string has alphanumeric characters

In [356]:

```
str = "@AVLargest *B Data Science community #AV!!"

# extract words that start with a special character
x = re.findall("[^a-zA-Z0-9 ]\w+", str)

print(x)
```

```
['@AVLargest', '*B', '#AV']
```

Solve Complex Queries

Let us try solving some complex queries using regex.

Extracting Email IDs

In [358]:

```
str = 'Send a mail to fatimah.1997@gmail.com, adewumi_adeola@yahoo.com and aisha1@yaho
o.com about the meeting @2PM'

# \w matches any alpha numeric character
# + for repeats a character one or more times
#x = re.findall('\w+@\w+\.com', str)
x = re.findall('[a-zA-Z0-9._-]+@\w+\.com', str)

# Printing of List
print(x)
```

```
['fatimah.1997@gmail.com', 'adewumi_adeola@yahoo.com', 'aisha1@yahoo.com']
```

Extracting Dates

In [362]:

```
text = "London Olympic 2012 was held from 2012-07-27 to 2012/08/12."

# '\d{4}' repeats '\d' 4 times
match = re.findall('\d{4}.\d{2}.\d{2}', text)
print(match)
```

```
['2012-07-27', '2012/08/12']
```

In [363]:

```
text="London Olympic 2012 was held from 27 Jul 2012 to 12-Aug-2012."

match = re.findall('\d{2}.\w{3}.\d{4}', text)

print(match)
```

```
['27 Jul 2012', '12-Aug-2012']
```

In [364]:

```
# extract dates with varying lengths
text="London Olympic 2012 was held from 27 July 2012 to 12 August 2012."

#'\w{3,10}' repeats '\w' 3 to 10 times
match = re.findall('\d{2}.\w{3,10}.\d{4}', text)

print(match)
```

```
['27 July 2012', '12 August 2012']
```

Extracting Title from Names - Titanic Dataset

In [372]:

```
import pandas as pd

# load dataset
titanic = r"C:\Users\Hamzat\Downloads\ANALYTICS VIDH\data\titanic.csv"
data=pd.read_csv(titanic)
```

In [373]:

```
data.head()
```

Out[373]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500

In [374]:

```
# print a few passenger names
data['Name'].head(10)
```

Out[374]:

```
0                              Braund, Mr. Owen Harris
1    Cumings, Mrs. John Bradley (Florence Briggs Th...
2                               Heikkinen, Miss. Laina
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                             Allen, Mr. William Henry
5                                     Moran, Mr. James
6                              McCarthy, Mr. Timothy J
7                       Palsson, Master. Gosta Leonard
8    Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
9                  Nasser, Mrs. Nicholas (Adele Achem)
Name: Name, dtype: object
```

Method 1: One way is to split on the pandas dataframe and extract the title

In [375]:

```
name = "Allen, Mr. William Henry"
name2 = name.split(".")
```

In [376]:

```
name2[0].split(',')
```

Out[376]:

```
['Allen', ' Mr']
```

In [377]:

```
title=data['Name'].apply(lambda x: x.split(".")[0].split(",")[1])
title.value_counts()
```

Out[377]:

```
 Mr              517
 Miss            182
 Mrs             125
 Master           40
 Dr                7
 Rev               6
 Mlle              2
 Col               2
 Major             2
 Sir               1
 Don               1
 Jonkheer          1
 Ms                1
 Capt              1
 Lady              1
 Mme               1
 the Countess      1
Name: Name, dtype: int64
```

This method might not work all the time. Therefore, another more robust way is to define pattern and search for it using regex

Method 2: Use RegEx to extract titles

In [378]:

```
def split_it(name):
    return re.findall("\w+\.",name)[0]
```

In [379]:

```
title=data['Name'].apply(lambda x: split_it(x))
title.value_counts().sum()
```

Out[379]:

```
891
```

In the above result, we observe that the title is followed by '.' since we are searching for a pattern that includes '.'