APPLIED ROOTS

AAIC Technologies Pvt Ltd

Essentials of Artificial Intelligence

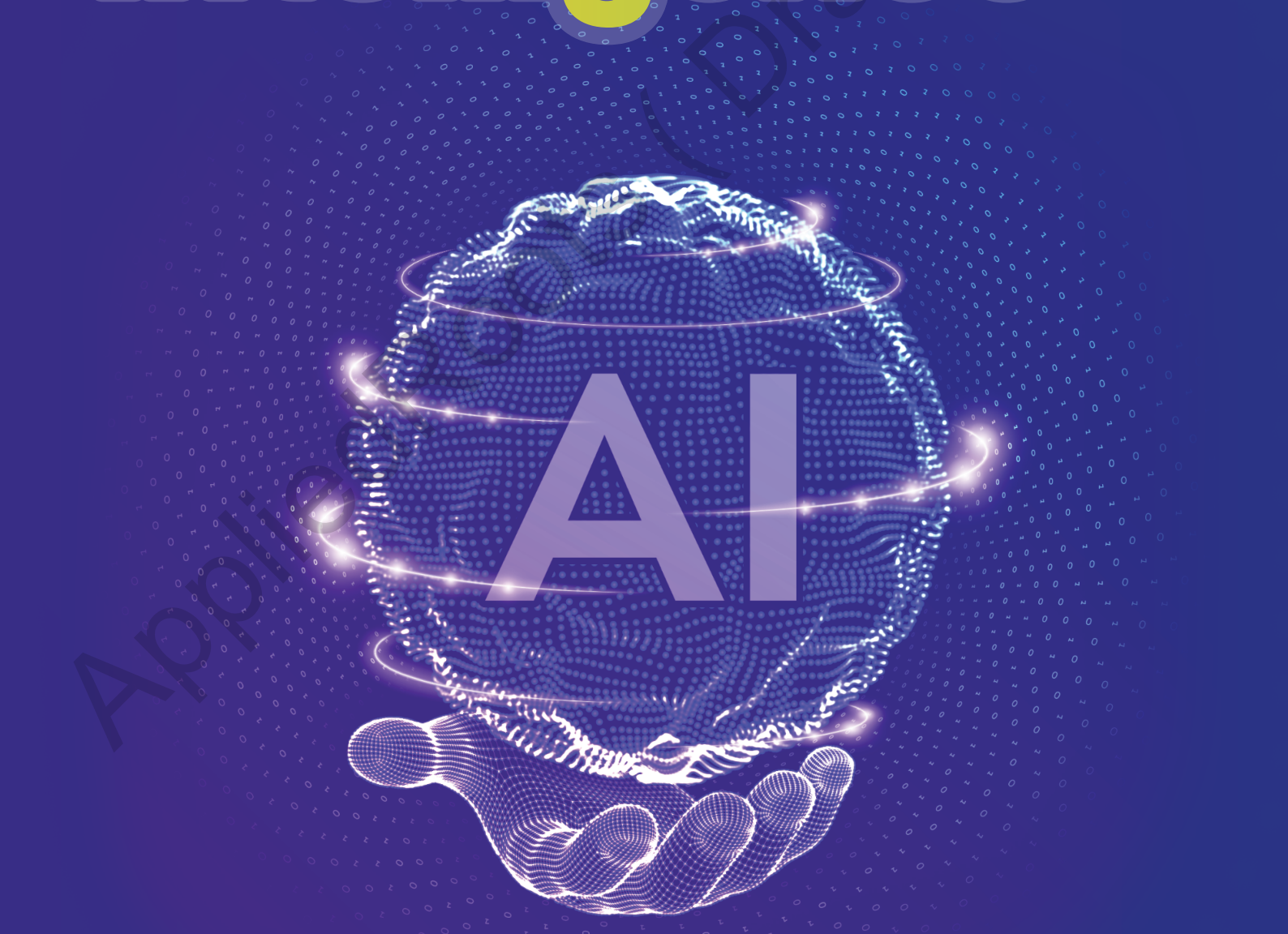


Table of Contents

1. PYTHON FUNDAMENTALS – 1

PYTHON FUNDAMENTALS - 1

Python is an interpreted, high-level, general-purpose programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It's syntax was designed with an emphasis on readability and expressibility (Express logical concepts in fewer lines of code). It supports multiple programming paradigms, including procedural, object-oriented, and functional.

Python was designed to be highly extensible and modular. Modularity means organizing/defining a system such that the overall set of major tasks is broken down into clearly defined independent repetitive sub tasks (modules), which can be optimally combined together. Modules are such that they can be replaced or upgraded without affecting other components of the system.

Python has a unique "white space" based syntax which is quite similar to the natural English language. It has its own built in data types, operators and functions which one could use to execute specific tasks or to create reusable modules. These modules can be further grouped together to form libraries. A Library is a set of modules grouped together with some broad subject domain in mind.

Due to python being open source, generally any python library can be accessed, modified and contributed to, by any other python user. Over the years this extensibility and modularity has led to a vibrant ecosystem of python libraries that cover a vast array of subjects like linear algebra, data visualization, astrophysics, medicine, Stock Trading, Machine Learning and Data Science to name just a few.

HIGH LEVEL AND LOW LEVEL LANGUAGES:

Python is a high level, interpreted language. High level languages use programming syntax styles that incorporate natural language elements, thus making them more accessible and programmable. Low level languages are slight abstractions over machine code. Machine code is just instructions in binary. Low level language is barely readable, but faster for the computer to understand and also memory efficient.

Broadly speaking computers can only run low level code. High level code has to first be converted to low level code before they can be executed.There are two kinds of programs that do this:

1. **Interpreters**

   An interpreter translates high level code and executes it one line at a time. This makes interpreted languages highly interactive due to the immediate feedback one receives by running an individual line of code (or a set of lines). In other words code can be inspected sequentially and at each step the outcomes verified, before the code is converted into a file and run as a whole.

   While executing python code in a jupyter notebook cell, the interpreter converts the code to low level, executes it and then presents the results of the execution if requested.

1. **Compilers**

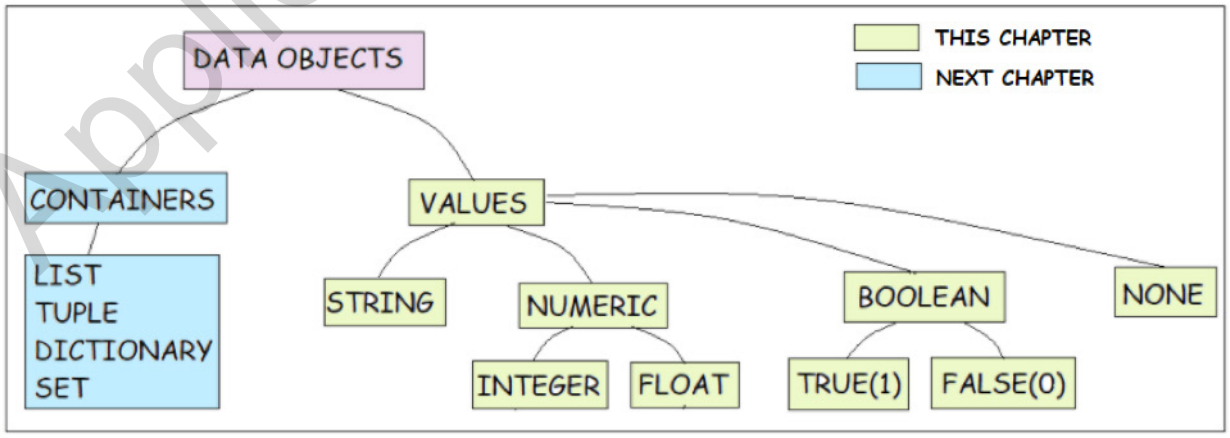
   A compiler converts complete/whole high level programs, into low level code called the object code, that can be executed as and when required. Once a high level program is compiled, its object code can be reused repetitively without needing to translate the original high level code again.

VARIABLES AND OBJECTS:

Everything in python is an object. All data stored in a python program is an object. All programming entities/constructs used within the python's language syntax are objects. Objects are the fundamental building blocks of the python language. Multiple objects can be programmatically combined together to form new complex objects. The concept of **objects** will be further elaborated in the chapter titled **"Object Oriented programming".** In this chapter we shall discuss only about objects that are specifically associated with **data** (ie: values).

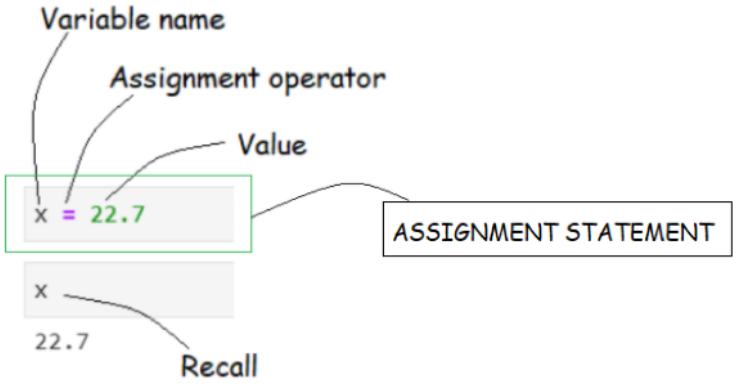
DATA OBJECTS IN PYTHON:

There are two fundamental types of data object types in python, values and containers. Shown below is an overview about the data objects in python. :



1. VALUES:

Values can be **numeric** (7, 0.2, -1.23) or **string** ('a', 'python'). Values are the fundamental information that programs work with. Values can be assigned to **variable names** (or variables), so that they can be **recalled**. We assign values to variable names using the **assignment operator (' = ')** as shown in the image below:



New values can be assigned to previously used variable names, this results in the previous value being deleted as shown below:

```
x = 'hello'

x

'hello'
```

Note the following:

a. **Strings** are simply any keyboard symbol between a pair of single or double quotes. (ex: "12", 'car_1', "<?#$w%|@>" etc). They are generally used to represent text.

b. **Variable names** must start with an alphabet and can only contain alphabets, numbers and the underscore character (ie: '_'). They cannot contain spaces or other characters like brackets and quotes. They also cannot be the same as certain keywords reserved exclusively to be used within python's language vocabulary. These are:

[and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield]

c. Apart from the numeric values discussed above, there are also the Boolean data type (**True** and **False**) and the **None** type which will be discussed further on.

d. A **python statement** is an instruction that the python interpreter can execute. When a python statement is run in a python cell or command line, the Interpreter executes it and displays the result (or results) if there is one. In the example above " **x = 22.7** " is an

assignment statement. It displays no result unless the the assigned value is recalled using its variable name.

Generally the end of a line is considered as the end of a statement. But more than one statement can also be expressed in a single line. This will be discussed further on in the book

2. CONTAINERS:

Containers are data objects that can hold multiple objects. The containers data types in python are: **Lists, Tuples, Dictionaries & Sets**. Each of these container data types have specific attributes and operations associated with them, which make them suitable for performing various tasks associated with grouped data. We will be discussing containers further in the next chapter.

Just like values, containers also can be assigned to variable names. The below code shows a **List** containing a group of values being assigned to a variable name:

```
group_1 = [1, 2, 'three']

group_1
```

```
[1, 2, 'three']
```

NUMERIC DATA TYPES:

Python's numeric data types can further be classified into three major categories: Integers, floats and Complex. We will limit our discussions to integers and float numeric types as they are more relevant:

INTEGERS:

Are all positive or negative whole number values (ie: non decimals, ex: 1, -200, etc)

FLOATS:

Are all positive or negative real numbers (ie: decimals, ex: -0.1, 7.22, etc). Floats can also be represented using scientific notation which contain exponents. (ex: 7.22e10, 1.2e-5, etc)

ARITHMETIC OPERATORS:

Numeric data types in Python support the normal arithmetic operations. The following example code blocks demonstrate the various python arithmetic operators and their corresponding operations:

```
x = 10
y = 12

x, y
```

```
(10, 12)
```

In the code above, the first line assigns the integer data 10 to variable name x using the assignment operator, similarly the second line assigns the value 12 to y. The third line prompts python to return the values corresponding to the variable names x and y in that order, after the cell is executed.

1. ADDITION AND SUBTRACTION:

Mathematical operation can be performed directly using numeric types as shown below or using variable name assignment.

```
5 + 1
```

```
6
```

```
v1 = x + y
v2 = x - y

v1, v2
```

```
(22, -2)
```

The first line of code in the second code block above performs the addition operation between the numerical values assigned to variable names x and y and assigns the result of the computation to another variable name v1. The second line performs the subtraction operation between x and y and assigns the result of the computation to v2. These operations are performed using the '+' and '-' operators respectively.

2. DIVISION BASED OPERATIONS:

```
v1/x,  v1//x,  v1%x
```

```
(2.2, 2, 2)
```

The line of code in the cell above performs three mathematical operations: Division, floor division and modulus. Note the operators used for each.

3. MULTIPLICATION:

```
v1 * v2
```

```
-44
```

Python uses the star symbol " * " as the multiplication operator.

4. EXPONENTIATION

```
v1**2, v1**(1/3)
```

```
(484, 2.802039330655387)
```

The code in the cell above performs two exponentiation operations. It computes the square and the cube root of v1. Python uses the double star symbol " ** " as the exponentiation operator.

COMPARISON OPERATORS:

Apart from the arithmetic operators described earlier, python also has several other kinds of operators. They are:

a. Identity operators.

b. Membership operators

c. Logical operators

d. Comparison operators

Broadly speaking, the outputs of all statements using non arithmetic operators result in the Boolean values True or False as their outputs. Note the capitalization of the first alphabet of each boolean value representation. We shall only discuss the comparison operator in this chapter. The rest of these operators will be discussed in the second chapter.

Comparison operators are used when two values have to be compared. Comparison statements result in Boolean outputs that either validate or invalidate the comparison. Consider the three code cells shown below.

```
x = 0
y = 1
z = 7

x, y, z
```

```
(0, 1, 7)
```

```
x == 0,  x == y,  y == z - 6
```

```
(True, False, True)
```

```
x == False,  y == True,  True + False
```

```
(True, True, 1)
```

The code in the second cell uses python's equal-to operator (ie: ' **==** ') to compare three value pairs. The first and third comparisons result in the **True** value as their outputs, since the values on either side of the equal-to operator are the same. The second comparison results in a **False** value output.

Note that:

1. The Boolean values True and False correspond to the numeric values one and zero, this is confirmed by the outputs of the third cell above.
2. The ' = ' symbol is used for assignment and the ' == ' symbol is used as the equal to operator in python.

Other comparison operators along with their corresponding symbols in python are shown below:

- Not equal to (!=)
- Greater than (>)
- Lesser than (<)
- Greater than or equal to (>=)
- Lesser than or equal to (<=)

Shown below are some examples of the above mentioned operators:

```
x == y,   x != y
```

```
(False, True)
```

```
x > y,   x < y,   x <= y,   z >= x
```

```
(False, True, True, True)
```

```
True > False
```

```
True
```

THE None VALUE:

The None value is used to indicate null or "no value". None is not equivalent to any other data type. The code below expresses the nature of the None data type.

```
zzz = None

zzz
```

No output (Null recall) variable zzz is not associated with any value

```
None == zzz,  None == False, None == 0
```

```
(True, False, False)
```

```
zzz + 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-f684dfc191ff> in <module>
----> 1 zzz + 1

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

The above cell produces an error message since None represents "no value", and hence it is unresponsive to any arithmetic operation.

CONDITIONAL EXECUTION:

The if statement in python aids in executing code blocks conditionally. The code shown below is simplistic demonstration of the conditional execution concept:

White space indentations:

Equal Spaces at the beginning of code lines signifying a single "block" of code whose execution depends on whether the if statement is satisfied.

```
x = 22

a = 0
b = 1
c = 2

if x < 25:

    a = b
    c = a**2

c
```

EXECUTED

```
1
```

NOT EXECUTED

```
x = 25.5

a = 0
b = 1
c = 2

if x < 25:

    a = b
    c = a**2

c
```

```
2
```

As can be seen from the code in the image above, the if statement has two elements associated with it:

1. A task (or set of tasks).
2. A condition (or set of conditions).

The execution of the task is dependent on the satisfaction of the condition. In other words if the comparison operation after the "if" statement results in the Boolean value **"True"**, then the code block belonging to the if statement is executed.

The semicolon (" : ") at the end of the if statement informs python that the equally indented code block from the next line onwards is associated with the if statement. In other words, white spaces are used to differentiate the code block belonging to the if statement, from the rest of the code.

ALTERNATIVE CONDITIONAL STATEMENTS (if – else):

If – else Statements are another form of the **if** statement, which is meant for "alternative" execution, Here there are two possibilities and the condition determines which one get executed. The code shown below demonstrates this:

```
x = 0

if x >= 0:
    result = 'positive'

else:
    result = 'negative'

result
```

```
'positive'
```

CHAINED CONDITIONAL STATEMENTS (if – elif):

If – elif statements are **if** statements that allow for multiple conditional executions to be performed. The code shown below demonstrates the If - Elif format:

```
x = 0

if x > 0:
    result = 'greater than zero'

elif x < 0:
    result = 'less than zero'

else:
    result = 'zero'

result
```

```
'zero'
```

NESTED CONDITIONAL STATEMENTS:

It is possible to nest one conditional execution within another. This can help in implementing complex control flows when needed. The code shown below demonstrates the concept of nested conditionals.

```
x = -0.0001

if x >= 0:

    if x < 1:
        result = 'positive value lesser than one'
    elif x >= 1:
        result = 'positve value greater than one'

else:

    if x >= -1:
        result = 'negative value greater than minus one'
    elif x < -1:
        result = 'negative value lesser than minus one'

result
```

```
'negative value greater than minus one'
```

TRIAL BASED ALTERNATE EXECUTION (try - except):

It is quite common that errors occur during execution of code. Every programming language provides a framework for dealing with them. Python's error model is based on **exceptions**. Whenever the python interpreter encounters **errors** (ie: code that it cannot process) it does the following:

1. It raises an exception.
2. It then interrupts/stops the normal flow of code at that error point.

There are basically two types of errors that python catches:

1. SYNTAX ERRORS:

```
x = 33
if x > 2
    result = 'greater than 2'

result
```

SEMICOLON MISSING

```
  File "<ipython-input-17-f2aeedcedf47>", line 2
    if x > 2
            ^
SyntaxError: invalid syntax
```

The python interpreter can only execute code that follows the rules of the python syntax. If it encounters code that does not match the python syntax, python raises an exception as shown above and does not execute the code.

2. RUNTIME ERRORS:

```
x = 'hello'
if x > 2:

    result = 'greater than 2'

result
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-3967d1231a57> in <module>
      1 x = 'hello'
----> 2 if x > 2:
      3
      4     result = 'greater than 2'
      5

TypeError: '>' not supported between instances of 'str' and 'int'
```

A runtime error is an error that passes the interpreter's syntax checks. It is not detected until the erroneous code is executed. In the above code we have attempted to perform a math operation on the wrong data type (ie: greater than comparison on a string data type), this resulted in an error during execution of the statement.

Try - except statements are basically designed to help the programmer handle runtime errors. It does this by allowing the programmer to provide alternate execution paths when such errors occur. The two cells shown below demonstrate the structure of try - except statements:

```
x = 2

try:
    if x%1 == 0:
        result = 'integer'
    else:
        result = 'float'

except:
    result = 'wrong_data'

result
```

ACTUAL CODE

ALTERNATE CODE
Code incase of wrong input datatype

```
'integer'
```

```
x = 2.5

try:
    if x%1 == 0:
        result = 'integer'
    else:
        result = 'float'

except:
    result = 'wrong_data'

result
```

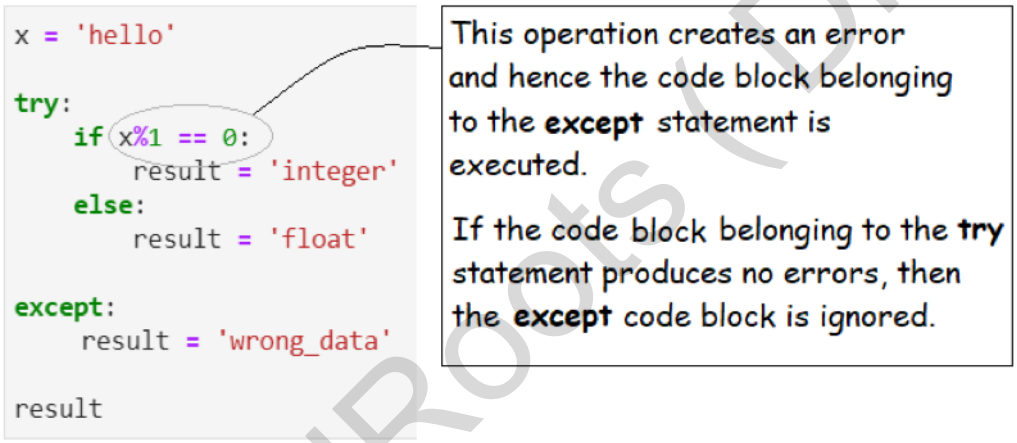
```
'float'
```

The structure of try - except statements are quite similar to that of if - else statements. Here too, we use a semicolon and block indentation to differentiate the following:

1. The actual code block to be first "tried".
2. The alternate code block code which would be run in case the "trial" resulted in a runtime error.
3. The rest of the code from the above two code blocks.

In the above shown code cells, the "actual" code belonging to the try statement creates no runtime error and hence the alternative code block belonging to the except statement is ignored.

Consider the same code, but this time let a string value be assigned to the variable x. On execution, the actual code runs into a runtime error and so the alternate code block belonging to the except statement is executed.



Try-else statements are very useful, especially when dealing with large amounts of data that could contain wrong entries. Try-else statements make it possible to avoid the code from crashing, in the midst of a critical operation, due errors in the input data. They allow us to execute the code on all correct input, while collecting information of wrong inputs if any.

The usefulness of try-except statements will become more apparent when we use them in context to container type data and iteration operations. These will be discussed further on in the book.

Similar to the if-elif format, the try statement also can be followed by more than one except code block. This allows the programmer to try out a series of code blocks of descending levels of criticality.

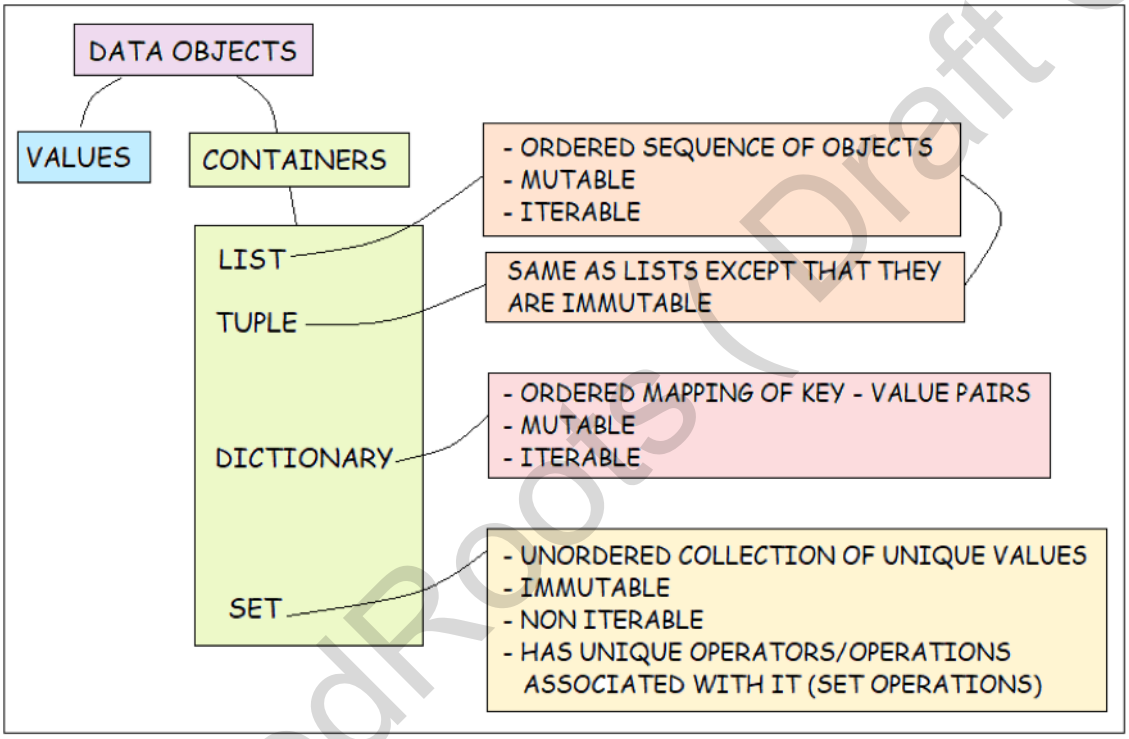
This concludes the first chapter. In the next chapter we will discuss container data types

2. PYTHON FUNDAMENTALS – 2

FUNDAMENTALS OF PYTHON – 2

In the first chapter we mentioned that there are two fundamental data types in python, **values** and **containers** and we then discussed the prior. This chapter we will explore container data types in python.

Containers are data types that can hold/contain multiple objects. The containers data types in python are: Lists, Tuples, Dictionaries & Sets. Shown below is an overview of container data objects and the main properties related to them.



LISTS:

A list is an **ordered mutable** collection of objects. Ordered meaning, the sequence of the objects contained within it, will always be the same, every time it is recalled using its variable name (unlike a random collection). The sequence/collection of objects is represented as a list by placing them within square brackets as shown below. Lists are **heterogeneous**, in that objects of different types can be contained within a single list.

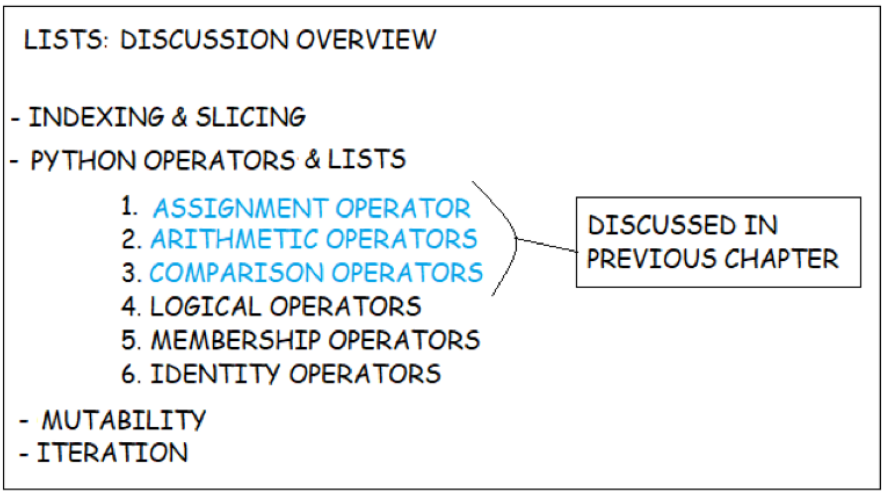
```
l = [111, 'hello', -3, 75, [56, 27]]

l
```

String

List within a list

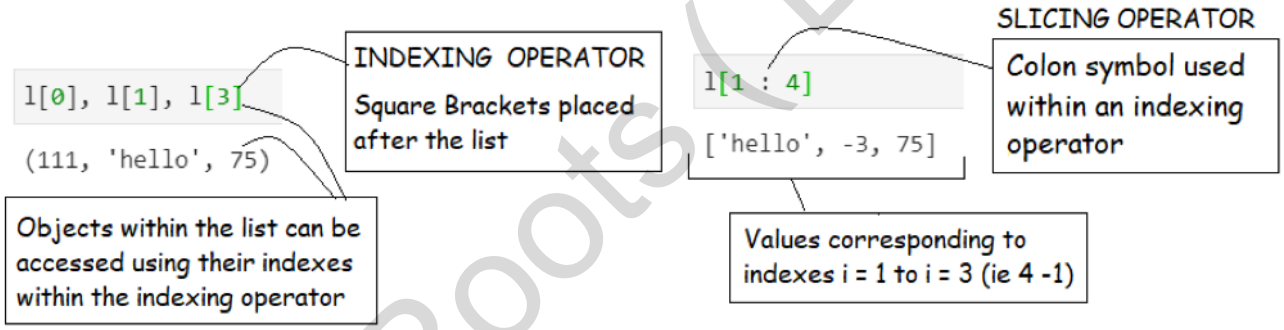
Lists are heterogeneous. Different kinds of objects can be stored within a list

```
[111, 'hello', -3, 75, [56, 27]]
```



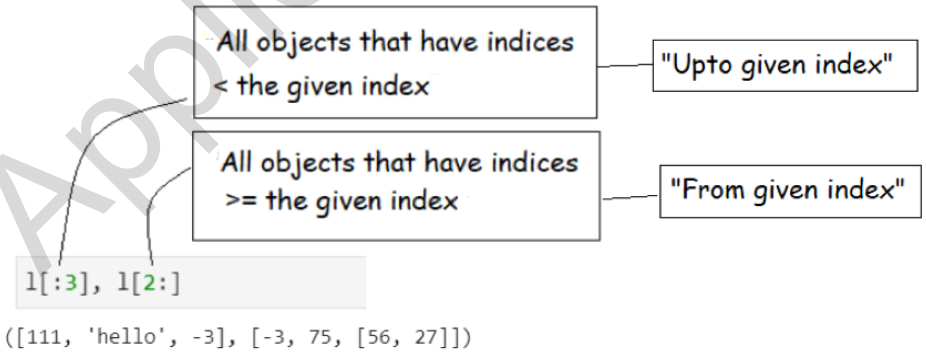
INDEXING AND SLICING:

Every object within a list is indexed with respect to its position and can accessed individually or in slices as shown below:



Note that indexes in python start from zero and that slice ranges are defined by the indexes of the start and end+1th values of the range desired.

We can also perform "open ended" slicing as shown below:



NEGATIVE INDEXING:

Lists can also be indexed using negative integers. Negative integers signify indexes with respect to the reverse sequence of the objects within the list. In other words l[-1] would refer to the last object of list l, l[-2] would refer to the second last object as so on. Shown below are examples of negative indexing and slicing.

```
l[-1], l[-2]
```

```
([56, 27], 75)
```

```
l[:-3], l[-3:]
```

```
([111, 'hello'], [-3, 75, [56, 27]])
```

STEPWISE SLICING:

The list slicing tasks performed previously used a default step size of one. This is further demonstrated in the code block A showb below. Python provides a stepsize parameter for slicing lists, this is demonstrated in code blocks B, c and E showb below.

A

```
l = [11,22,33,44,55,66,77]
l[0:6]
```

```
[11, 22, 33, 44, 55, 66]
```

B

```
l[0:6:1]
```

```
[11, 22, 33, 44, 55, 66]
```

C

```
l[0:6:2]
```

```
[11, 33, 55]
```

D

```
l[0:]
```

```
[11, 22, 33, 44, 55, 66, 77]
```

E

```
l[0::1]
```

```
[11, 22, 33, 44, 55, 66, 77]
```

BLOCK B:

The slicing used in this block consists of a third number after another colon symbol. This number specifies the step size of the slice desired. The code here overtly defines the step size to be one, which is actually the default step size and hence it produces the same result as in Block A.

BLOCK C:

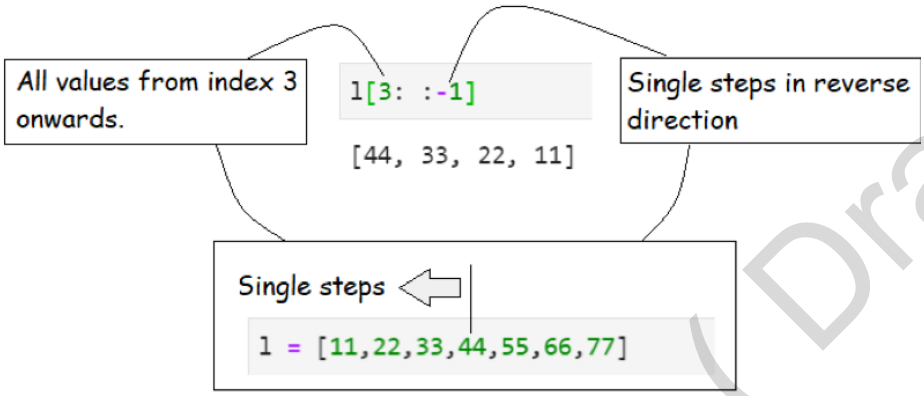
The slicing used in this Block has the same range as that in **Block 2**, but it specifies a step size of two and hence it produces the result as shown above.

BLOCK E:

The slcing shown in this Block has the same meaning as that in **Block D**, but it overrtly specifies the step size as one (default step size)

STEPWISE SLICING IN THE REVERSE DIRECTION:

Using negative values to specify the step size during any slicing operation results in values got by taking steps (of the step size specified) in the reverse direction. This demonstrtaed in the code shown below.



REVERSE THE ORDER OF A LIST USING NEGATIVE STEPWISE SLICING:

One can get a list slice that consists of all the values in the original list in the reversed order by using a negative step size parameter as shown in the code below.

```
l[-1: :-1]
```

```
[77, 66, 55, 44, 33, 22, 11]
```

MORE PYTHON OPERATORS:

In the previous chapter we introduced arithmetic operators and comparison operators and applied them on numeric data types. Lists can be operated upon by **addition** and **multiplication** arithmetic operators, as shown in the examples below.

```
l1 = [12, 33]
l2 = [5, 91]

l3 = l1 + l2
l3
```

```
[12, 33, 5, 91]
```

CONCATENATED LIST OF L1 AND L2

```
l4 = l3 * 2

l4
```

```
[12, 33, 5, 91, 12, 33, 5, 91]
```

DUPLICATION OF THE SEQUENCE OF OBJECTS IN LIST L3

Apart from the arithmetic and comparison operators, there are also the following other operators in python: Logical operators, membership operators and identity operators.

I. LOGICAL OPERATORS:

Logical operators in python are 'and', 'or' and 'not'. They are generally used to combine more than one conditional statements. Examples of Logical operators are shown below:

A. ATHE AND OPERATOR:

The and operator is used to create conditional statements that verify more than one condition simultaneously as shown below:

```
x = 7

if x < 100 and x%2 == 0:
    result = 'even number lesser than 100'

if x < 100 and x%2 !=  0:
    result = 'odd number lesser than 100'

result
```

```
'odd number lesser than 100'
```

B. THE OR OPERATOR:

The **or** operator is used to create statements that allow us to implement "either-or" conditions as shown below:

```
day = 'monday'

if day == 'saturday' or day ==  'sunday':
    result = 'holiday'

else:
    result = 'working day'

result
```

```
'working day'
```

C. THE NOT OPERATOR:

The not operator is used in situations when we want to reverse the meaning of the conditions in a conditional statement - as shown below:

```
day = 'sunday'

if not day == 'saturday' and not day == 'sunday':
    result = 'working day'

else:
    result = 'holiday'

result
```

```
'holiday'
```

II. THE MEMBERSHIP OPERATOR (IN):

The membership operator "in" is used to check if some object is contained within a container or not, as shown below:

```
l1 = [11, 22, 99]
l2 = [22, 44, 66]
x = 22

if x in l1:
    result = 'x in l1'
if x in l2:
    result = ' x in l2'
if x in l1 and x in l2:
    result = 'x in l1 and l2'
if not x in l1 and not x in l2:
    result = 'x not in l1 or l2'

result
```

```
'x in l1 and l2'
```

```
l1 = [11, 22, 99]
l2 = [22, 44, 66]
x = 77

if x in l1:
    result = 'x in l1'
if x in l2:
    result = ' x in l2'
if x in l1 and x in l2:
    result = 'x in l1 and l2'
if not x in l1 and not x in l2:
    result = 'x not in l1 or l2'

result
```

```
'x not in l1 or l2'
```

III. THE IDENTITY OPERATOR (IS):

Identity operators are used to compare objects and check if they are the same or not as shown below:

```
l1 = [11, 22, 99]
l2 = l1

l1 is l2
```

```
True
```

```
l1 = [11, 22, 99]
l2 = [11, 22, 99]

l1 is l2
```

```
False
```

```
l1 = [11, 22, 99]
l2 = [11, 22, 99]

l1 == l2
```

```
True
```

Note that identity operators check if the objects are the same as opposed to if the objects are equal. Both mean different things. The three code blocks shown above demonstrate this fact.

MUTABILITY:

List objects are mutable in the sense that values within the list can be modified/changed as shown below:

```
l1 = [12, 33, 5]

l1
```

```
[12, 33, 5]
```

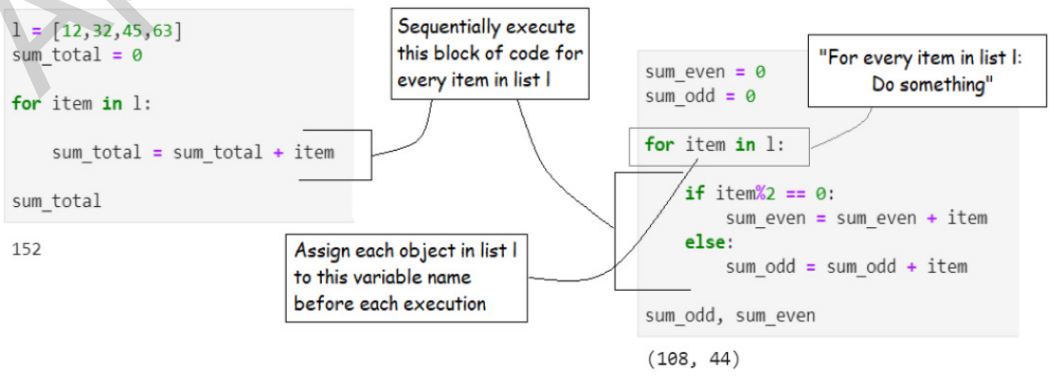
```
l1[-1] = l1[0] * 2

l1
```

```
[12, 33, 24]
```

The length (ie: the number of objects contained within a list) is also mutable. It can be changed by removing existing objects or adding new objects. These operations will be discussed further on in the book.

ITERATION (LOOPED EXECUTION):

Lists are "iterable" container objects. Iterable meaning, looped executions can be performed on them. Loosely speaking looped execution simply means repetitive execution of the same block of code on all objects contained within a container object. The for loop is the most basic form of iteration available in python. The images below demonstrate the structure and functioning of a for loop.



The for statement uses the for operator and the in operator in the syntax shown in the images above. The colon symbol followed by an indentented block differentiates the code belonging to the for statement from the rest of the code.

In the examples shown above, the for statement does the following:

1. It assigns the first object in the container to the variable name item
   (ie: item = container[0])
2. It executes the indented block of code on the object assigned to the variable name item.
3. It repeats the step 1 and 2 sequentially on all other objects in the container.

Shown below is the "segregate and collect" iteration pattern using the for loop.

```
l = [86, 31, 92, 21]
list_even = []
list_odd = []

for i in l:

    if i%2 == 0:
        list_even = list_even + [i]
    else:
        list_odd = list_odd + [i]

list_even, list_odd
```

```
([86, 92], [31, 21])
```

Empty lists assigned to variable names (lists can be empty or even contain just one object)

Object which is assigned to variable name i is placed inside a list and then this list is concatenated with another list using the addition operator.

```
l1 = []
l2 = []

l1 is l2
```

```
False
```

The code examples shown above, demonstrate two of the most common tasks that iterations are used for: **summating** and **collecting**. Both of these use the "**update pattern** " within its loops. The image shown below describes this:

```
l = [12,32,45,63]
sum_total = 0

for item in l:

    sum_total = sum_total + item

sum_total
```

```
152
```

Initial value assigned to the variable name that will be used within the for loop

The value associated with this variable_name is updated at every iteration of the for loop

First loop: sum_total = 0 + 12 = 12
Second loop: sum_total = 12 + 32 = 44
Third loop: sum_total = 44 + 45 = 89
Last loop: sum_total = 89 + 63 = 152

A similar pattern as described above is used for collecting values within a list. This can be seen in the code used to describe the "segregate and collect" pattern discussed earlier.

Shown below is an example of the same pattern as described in the above examples, but with the try-except statement included. The code is designed to segregate integers and floats from a list of numbers l. The try-except block is used to make the code resilient to wrong data inputs. The code does not crash when executing the modulus operator (%) on the wrong data type, but instead an alternate code is executed. This ensures that the entire list is iterated through irrespective of runtime errors encountered during the midst of the iteration. Also note how the errors are collected in a separate list and thus inspectable later on.

```
l = [1, '47', 222.2, '5.5', 66]
l_int = []
l_float = []
l_wrong_data = []

for i in l:
    try:
        if i%1 == 0:
            l_int = l_int + [i]
        else:
            l_float = l_float + [i]
    except:
        l_wrong_data = l_wrong_data + [i]

l_int, l_float, l_wrong_data
```

```
([1, 66], [222.2], ['47', '5.5'])
```

TUPLES:

Lists are mutable data objects, in that it can change or mutate during runtime. The objects within a list can be changed and the length of a list can be increased or decreased by adding or removing objects from it.

A tuple is an ordered list like container object, except that it is immutable. This means that once a sequence of objects is contained within a tuple, it cannot be changed in any way under any circumstance.

A sequence/collection of objects is represented as a tuple by placing them within curved brackets as shown below. Tuples like lists are also heterogeneous in that they can contain objects of different types.

```
t = (67, 55, [1, 2])

t
```

```
(67, 55, [1, 2])
```

Just as with lists, every object within a tuple is indexed with respect to its position and can accessed individually or in slices.

```
t[0], t[-1], t[:-1]
```

```
(67, [1, 2], (67, 55))
```

The example below demonstrates the immutability of tuples:

```
t[-1] = 98
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-120-2c22898c9818> in <module>
----> 1 t[-1] = 98

TypeError: 'tuple' object does not support item assignment
```

Immutable data containers like tuples are used in situations where the data being dealt with is of a critical nature and must be resilient to any form of tampering.

Tuples too are iterable containers. The code below shows iteration applied to a tuple:

```
t = (12,32,45,63)
sum_total = 0

for item in t:

    sum_total = sum_total + item

sum_total
```

```
152
```

ASSIGNING MULTIPLE VALUES TO A SINGLE VARIABLE NAME:

When multiple values are assigned to a single variable name, python by default assumes that those values are contained in a tuple. The two code cells shown below demonstrate this fact:

```
values = 12, 13, 14

values
```

```
(12, 13, 14)
```

tuple

```
values[1] = 555
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-4fdf95727612> in <module>
----> 1 values[1] = 555

TypeError: 'tuple' object does not support item assignment
```

SIMULTANEOUS ASSIGNMENT (TUPLE/LIST UNPACKING):

Multiple assignment statements can be grouped together in a single line of code as shown below:

```
a, b, c = 11, 22, 33

a, b, c
```

```
(11, 22, 33)
```

This is called tuple unpacking. The values on the right side of the assignment operator are assumed by python to be contained in a tuple. Python allows each value within a tuple (or a list) to be assigned to unique variable names just by using those unique variable names on the left side of the assignment operator. The code below further demonstrates this fact. This is called "**unpacking**".

```
t = (11, 22, 33)
a, b, c = t

a, b, c
```

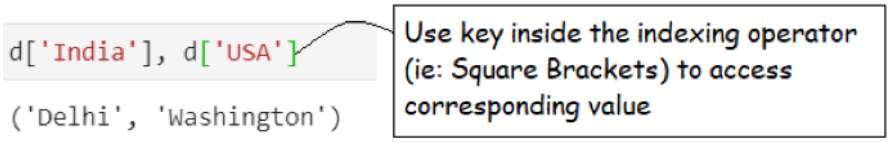
```
(11, 22, 33)
```

DICTIONARIES:

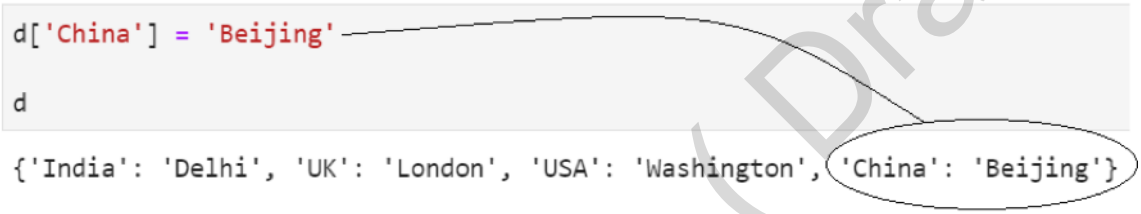
DIctionaries are basically **ordered** sets of **key-value pairs**. DIctionaries can contain any number of key-value pairs and the values associated with a key can be any object (ie: Dictionaries are **heterogeneous** in nature). DIctionaries are **mutable**, which means new key-value pairs can be added to it or existing ones removed, or the values of an existing key can bemodified.

Dictionaries are created in python by inserting key-value pairs within curly brackets in the manner shown shown below. Each key has an object (ie: "value") assigned to it using the colon operator.

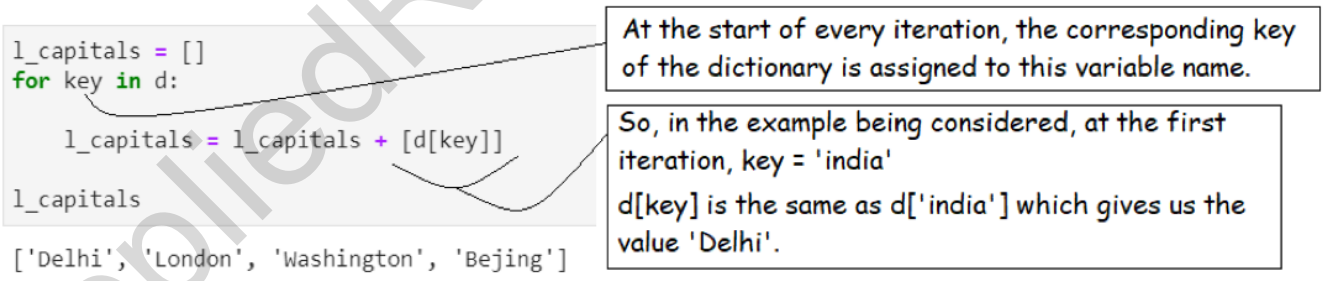
Each value in the dictionary can be accessed using its key in the manner shown below:



New key-value pairs can be added to an existing dictionary as shown below:



Dictionaries are iterable objects. Shown below is an example of using the for loop on the previous dictionary d. When the for loop is applied to dictionaries, it iterates the keys of the dictionary in the same sequence as the key-value pairs inside the it. These keys are then used inside the for loop to access their corresponding values. The accessed values are then used to perform the desired task.



Shown below are some examples where a list is being iterated, but the results are captured using a **dictionary**:

```
l = [86, 31, 92, 21]

d = {}
d['even_sum'] = 0
d['odd_sum'] = 0


for i in l:

    if i%2 == 0:
        d['even_sum'] = d['even_sum'] + i
    else:
        d['odd_sum'] = d['odd_sum']  + i

d
```

Empty dict created and then key value pairs assigned to it.

```
{'even_sum': 178, 'odd_sum': 52}
```

```
l = [86, 31, 92, 21]

d = {}
d['even'] = []
d['odd'] = []


for i in l:

    if i%2 == 0:
        d['even'] = d['even'] + [i]
    else:
        d['odd'] = d['odd']  + [i]

d
```

```
{'even': [86, 92], 'odd': [31, 21]}
```

Note that in the second example shown above, the values corresponding to the keys of the resulting dictionary are list objects. Dictionaries can store any python object as their value, even dictionaries. This is demonstrated in the code below:

```
d = {1: 'hello', 2: 1000, 3: {'a': 10, 'b': 20}}

d
```

```
{1: 'hello', 2: 1000, 3: {'a': 10, 'b': 20}}
```

SETS:

A Set data object is an unordered collection of unique objects. In other words sets do not allow duplicate objects to exist within it. Sets like dictionaries are mutable and heterogeneous. Since sets are unordered, the elements contained within it cannot be accessed individually using indexing techniques. Unlike the previous three container data types, Sets are not iterable.

Dictionaries are created in python by inserting the values belonging to the set within curly brackets in the manner shown shown below.

```
s = {22, 'hello', 22, 7, 'hello', 39}
s
```

```
{22, 39, 7, 'hello'}
```

Sets can contain only unique values. Note that the duplicate values entered during the creation of the set are dropped once the set is created.

Sets have their own set of operations and operators that correspond to set theory:

1. Union operator: s1 | s2
2. Intersection operator: s1 & s2
3. Difference operator: s1 - s2 (Items in s1 but not in s2)
4. Symmetric difference operator: s1 ^ s2 (items in s1 or s2 but not in both)

Shown below are some example that demonstrate set operations:

```
s1 = {1 ,2, 3, 4, 5, 6}
s2 = {6, 7, 8, 9, 10, 'hello'}
```

1. UNION:

```
s1 | s2
```

```
{1, 10, 2, 3, 4, 5, 6, 7, 8, 9, 'hello'}
```

2. INTERSECTION:

```
s1 & s2
```

```
{6}
```

3. DIFFERENCE:

```
s1 - s2
```

```
{1, 2, 3, 4, 5}
```

```
s2 - s1
```

```
{10, 7, 8, 9, 'hello'}
```

4. SYMMETRIC DIFFERENCE:

```
s1 ^ s2
```

```
{1, 10, 2, 3, 4, 5, 7, 8, 9, 'hello'}
```

This concludes the second chapter, in the next chapter we shall discuss the concept of **functions.**

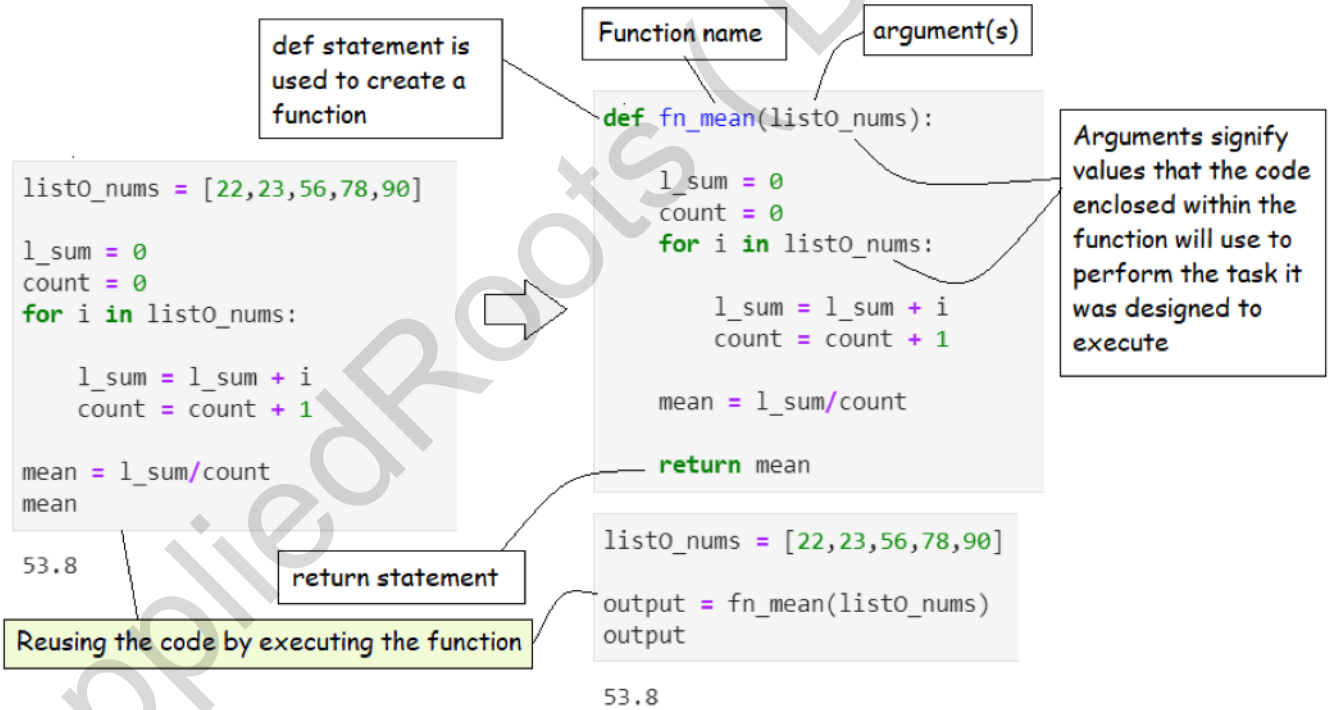
3. FUNCTIONS

FUNCTIONS

All code/programs used in the book until now were single use code blocks. If any of them had to be used again (on another input value or set of input values), the entire code has to be typed again and executed. This can be quite cumbersome and can lead to unorganized/unreadable code.

A function can be defined as a named sequence of statements that performs some desired computation. This named sequence of events can be executed whenever necessary by recalling them using their name, instead of needing to write the whole code (set of statements) again.

Functions are defined using the def statement. The image below shows an example function named "fn_mean" programmed to compute the mean of any list of numbers fed to it. It demonstrates the structure/syntax of functions in python.



The basic components of function are:

1. The def statement
2. The function's name
3. The function's argument(s) or inputs
4. The function's code block
5. The functions output(s) or desired results
6. The return statement

The function is named by choosing some suitable variable name and then typing it one white space away from the def statement. The name of the function is then followed by a curved bracket which is followed by a colon symbol. The arguments of the function, if any, are placed inside this curved bracket. The code belonging to the function is differentiated from the rest of the code by using indentations as described previously. The return statement is used to specify the output that the function should return. In the above

example, the return statement is followed by the variable name - mean and hence the function returns the value associated with this variable name, when it is called or executed.

CALLING A FUNCTION:

Once a function is defined, it can be executed any number of times. This is done by using the function name, followed by curved brackets containing the required arguments, as shown below:

```
listO_nums = [22,23,56,78,90]

output = fn_mean(listO_nums)
output
```

```
53.8
```

When we use the function's name as shown in the code above, it is called a "function call". This is because when this statement is executed, python "calls" the function that corresponds to the name "fn_mean" and executes the code contained within it using the argument(s) specified within the curved brackets. After executing the function's code, it then executes the last line within the function, the return statement. The return statement then "returns" the value specified after it. This returned value can be assigned to a variable name and used further on if required. In the function call shown above, the function returns the mean of the values in the list argument fed to it. This returned value is assigned to variable name "output".

The code shown below, further clarifies the concept of "calling" a function. Note the values specified after the return statements.

```
def some_fn_1(x):
    return x
```

```
def some_fn_2(x):
    return "hello", x + 2
```

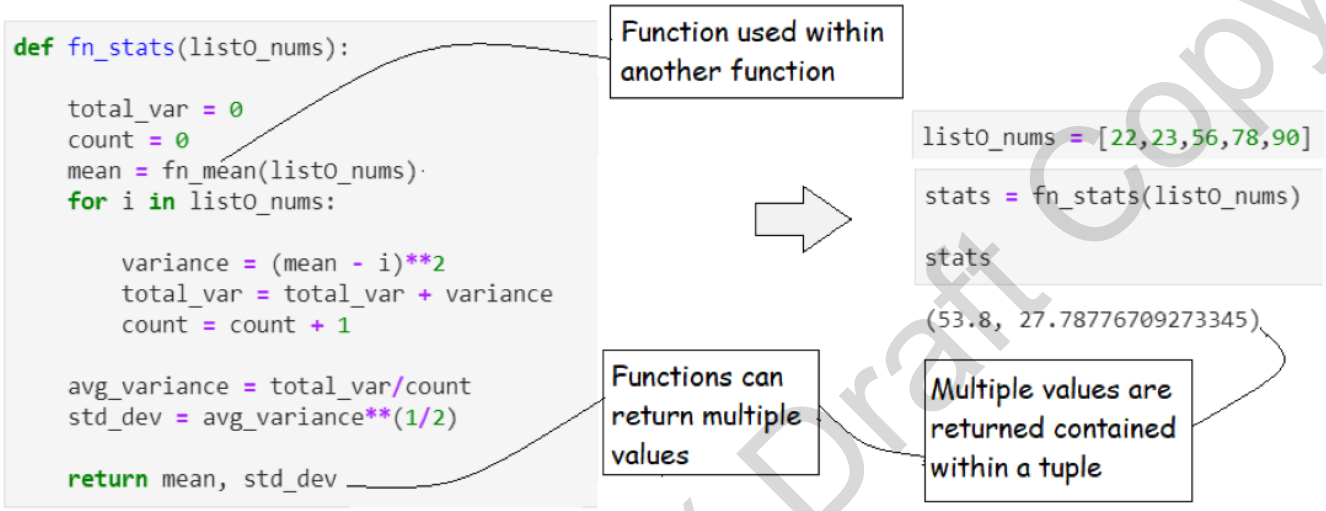
```
def some_fn_3(x):
    y = x**2
    return y
```

```
x = 3

some_fn_1(x), some_fn_2(x), some_fn_3(x)
```

```
(3, ('hello', 5), 9)
```

FUNCTIONS CAN BE USED WITHIN OTHER FUNCTIONS:

Functions can be used within other functions in python. This makes it easy to create modular code. Shown in the image below, the previously defined function **fn_mean** is used within another function **fn_stats**.



Note that in the above example:

1. The function returns two outputs: mean and std_dev. Python functions can return multiple outputs and it does so by enclosing those multiple outputs within a tuple.
2. There are no limits to the number of arguments that a function can have.
3. Arguments need not be of the same kind, they can be any python object that can be used in the code enclosed within the function.
4. It is not necessary for a python function to have arguments. This is demonstrated in the code shown below:

```
def fn_hello():
    output = "welcome to python"
    return output
```

```
fn_hello()
```

```
'welcome to python'
```

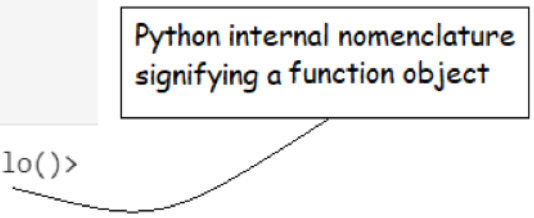
FUNCTIONS ARE OBJECTS:

Once a function is defined, the function name represents a **"function object"**. A function object is an object that can be "executed" using curved brackets. With reference to the previous example, the name

fn_hello represents a python object. This is demonstrated in the code below:

```
x = fn_hello

x
```

```
<function __main__.fn_hello()>
```



The function object will only be called (ie: executed) if the curved brackets are typed after it and the code cell run. This is demonstrated in the code shown below:

```
x()
```

```
'welcome to python'
```

In the code above, the function object **fn_hello** was assigned to variable name **x**. So, when curved brackets are used after **x** and the code cell run, **fn_hello** gets **called**/executed.

Function objects can be used like any other object in python, which means it can be used as an input argument for another function as shown below:

```
def fn_popular():

    result = 'The most popular programming language'

    return result
```

```
def fn_combine(fn_1, fn_2):

    result1 = fn_1()
    result2 = fn_2()

    return result1, result2
```

```
fn_1 = fn_hello
fn_2 = fn_popular

fn_combine(fn_1, fn_2)
```

```
('welcome to python', 'The most popular programming language')
```

This property of functions objects, makes it possible to advantageously combine and execute combinations of many functions within a single function.

KEYWORD ARGUMENTS IN FUNCTIONS:

Sometimes when defining a function, there are certain input arguments to which we would like to assign "**default**" values. This is made possible by using **keyword arguments**. Keyword arguments are such that they allow for default values to be provided by the function's **programmer**, but also give the function's **user** the choice to provide their own values. Keyword

arguments can be considered as "**assigned arguments**" of a function. In the example below, the assignment statement **return_variance = False** is a keyword argument. It is placed after the **normal argument** listO_nums, within the function's curved brackets.

```
def fn_stats(listO_nums, return_variance = False):

    total_var = 0
    count = 0
    mean = fn_mean(listO_nums)
    for i in listO_nums:

        variance = (mean - i)**2
        total_var = total_var + variance
        count = count + 1

    avg_variance = total_var/count
    std_dev = avg_variance**(1/2)

    if return_variance == True:
        stats = mean, avg_variance
    else:
        stats = mean, std_dev

    return stats
```

Default value

Keyword argument infuences what the function will return

In the function shown above, the default value of **False** assigned to r**eturn_variance,** makes the function return the **mean** and **standard deviation** of the values contained within **listO_nums**. While executing the function using its default keyword argument, the user need not specify anything other than the functions argument(s). This is demonstrated in the code shown below:

```
listO_nums = [22,23,56,78,90]

stats = fn_stats(listO_nums)
stats
```

Only argument(s) of the function need to be specified.

```
(53.8, 27.78776709273345)
```

mean | standard_deviation

If one chooses not to use the default argument, they can do so by assigning another value to the keyword_argument, to get the result they expect (In this case making the function return the **mean** and **variance** values instead of mean and standard deviation) as shown below.

```
stats = fn_stats(listO_nums, return_variance = True)

stats
```

Alternate value to be provided to the keyword argument

```
(53.8, 772.1600000000001)
```

mean | variance

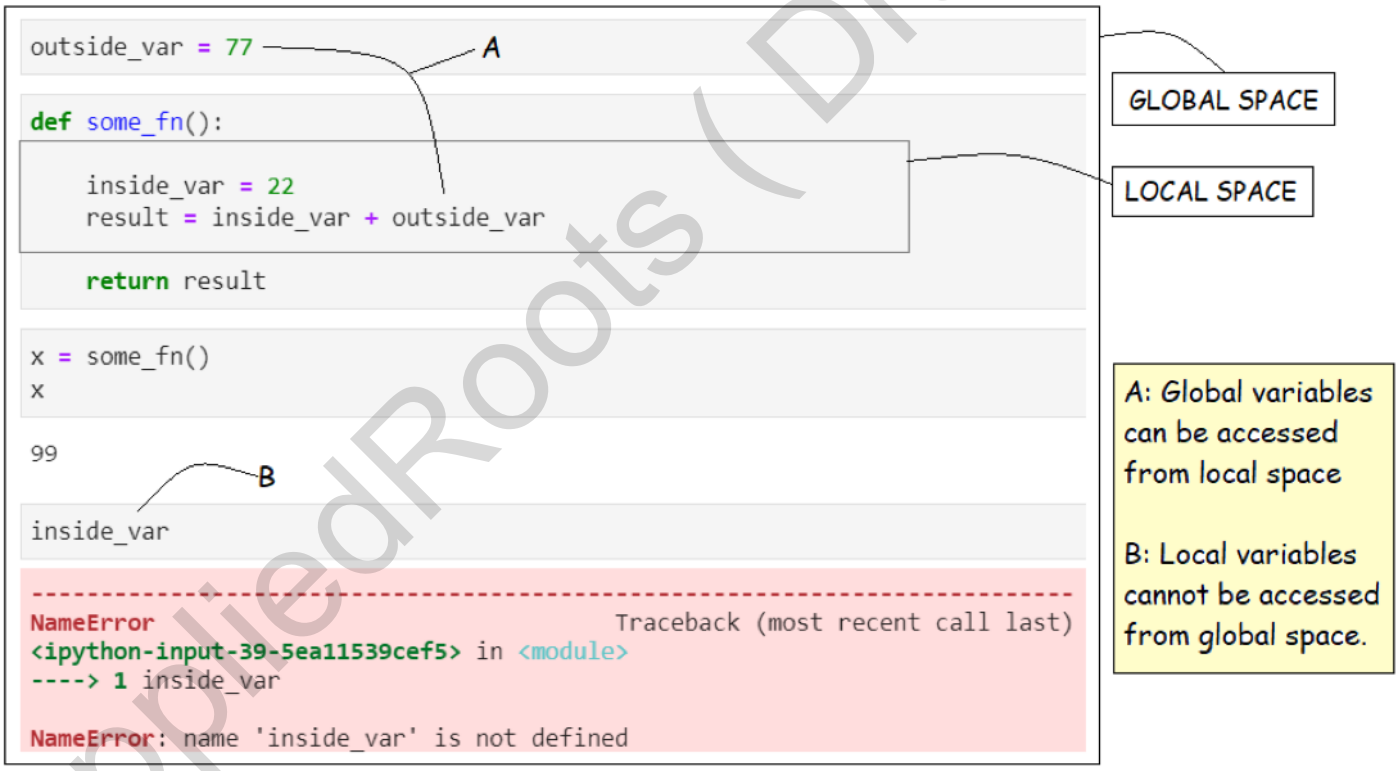
SCOPE OF VARIABLE NAMES AND FUNCTION NAMES:

Variable names and function names in python have scopes (code regions) within which they are accessible. They are not available outside the region they were created in. Therefore a variable created inside a function belongs to the local scope of that function and can only be used inside that function.

Whenever a variable name is encountered within a function, python does the following:

1. It searches the function's local space for an assignment statement that corresponds to that variable name and then uses the assigned value.
2. If it does not find such an assignment statement inside the function's local space, it looks for it in the global space. If it does not find it there too, python raises a "**name error**" exception and stops the program.

The code shown below demonstrates the above statements:



The global and local **scope rules** defined above for variable names also apply to **function names**. Functions existing in global space can be accessed within local spaces but not vice versa.

PYTHON INBUILT FUNCTIONS:

Python comes with its own set of inbuilt functions. These help the programmer by avoiding the need to program archetypical mathematical functions and other requirements from scratch. Show below are some of the ones most relevant for our purposes:

print:

The print function prints whatever data object is given to it as an argument. This is shown in the image below.

```
x = 24
y = 3
print(x)
print(x+y)
print('hello')
```

```
24
27
hello
```

abs:

The abs function, returns the absolute value of the numerical data given to it as an argument:

```
abs(-1), abs(10 - 10.2)
```

```
(1, 0.1999999999999993)
```

round:

The round function returns the rounded version (to the nearest integer) of the number given to it as an argument. It also contains a keyword argument "**ndigits**", to specify the decimal precision required after rounding off an input number.

```
round(2.348), round(2.348, ndigits = 2)
```

```
(2, 2.35)
```

len:

The len function returns the size or the count of the number of objects present in a list/tuple objects given to it as an argument.

```
l = [22, 33, 44, (55, 555)]
len(l)
```

```
4
```

min and max:

The min and max functions return the minimum and maximum values of the list/tuple fed to it as an argument.

```
l = [4, 2.2, 7, 21.87]

min(l), max(l)
```

```
(2.2, 21.87)
```

sum:

The sum function returns the summation of all the values of a list/tuple fed to it as an argument.

```
l = [4, 2.2, 7, 21.87]

sum(l)
```

```
35.07
```

sorted:

The sorted function returns the sorted version of the list/tuple fed to it as an argument. It has a keyword argument "**reverse**", which has two choices: **True** and **False**. If True, it sorts the values in descending order.

```
l = [4, 2.2, 7, 21.87]

sorted(l), sorted(l, reverse = True)
```

```
([2.2, 4, 7, 21.87], [21.87, 7, 4, 2.2])
```

all:

The all function returns the boolean value **True** if all the values within the list/tuple fed to it are non zero. It returns **False** otherwise:

```
l1 = [4, 2.2, 7, 21.87]
l2 = [4, 2.2, 7, 0]

all(l1), all(l2)
```

```
(True, False)
```

any:

The any function returns the boolean value **True** if any value within the list/tuple fed to it are non zero. It returns **False** otherwise:

```
l1 = [4, 2.2, 7, 21.87]
l2 = [4, 2.2, 7, 0]

any(l1), any(l2)
```

```
(True, True)
```

Using the built in functions provided by python, the function **fn_stats** defined earlier could be simplified further as shown below:

```
def fn_stats(list0_nums):

    mean = sum(list0_nums)/len(list0_nums)
    list0_variations = []
    for i in list0_nums:

        variance = (mean - i)**2
        list0_variations = list0_variations + [variance]

    avg_variance = sum(list0_variations)/len(list0_variations)
    std_dev = avg_variance**(1/2)

    return mean, std_dev
```

```
list0_nums = [22,23,56,78,90]

stats = fn_stats(list0_nums)
stats
```

```
(53.8, 27.78776709273345)
```

This concludes the third chapter, next chapter we will explore **Object Oriented programming.**

4. OBJECTORIENTED PROGRAMMING(OOP)

OBJECT ORIENTED PROGRAMMING(OOP)

OBJECTS AND CLASSES:

As mentioned earlier, everything is an object in python. Object Oriented programming is a programming paradigm, where programs are seen as interactions between objects. To understand what an **object** is, one needs to understand what a **Class** is. Classes can be thought of as a blueprint for creating objects, they are a particular way of writing code that allows the programmer to bundle **state** and **behaviour** together in an object. It is the fundamental building block of python. Every object in python belongs to some particular class.

In OOP we try to simulate objects by defining their **state** and **behavior**. Let us take the example of a water container. We can simulate water containers by first clearly defining their state and behaviour.

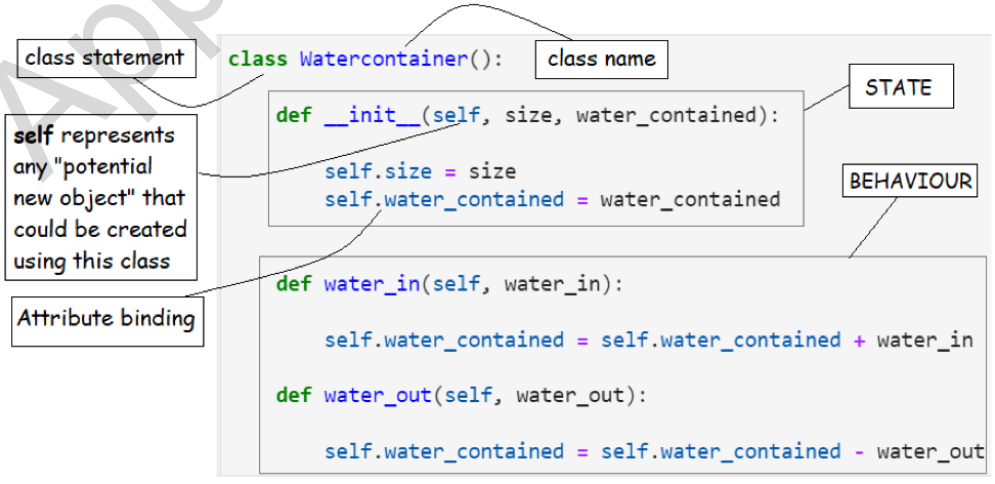
1. The state of a water container could be defined by its size and the quantity of water it contains.
2. The behaviour could be defined as follows: Water can be taken out of the container, thus decreasing the quantity of water it contains or water can be put into the container, thus increasing the quantity of water it contains.

As can be inferred from above, behaviours are a set of actions inherent to the object. These actions have the ability to change the state of the object.

Classes are code constructs in python (like functions) which enables one to:

1. 1. Define the **state** and **behaviour** of the object we wish to simulate.
2. Easily create multiple usable objects that follow these defined behaviours and reflect corresponding changes in state

The code shown below describes how classes are created, using the example of a water container.



Classes are defined in a way similar to defining functions. We use the **class statement** followed by the **name** we wish to give the class, followed by the curved brackets and the **colon** symbol. All indentend code from the next line after the colon symbol belongs to the class statement.

The code inside the class statement contains functions that define the state and behaviour of the object (ie: water container). This is done as follows:

1. The first function inside the class statement is generally always the "**__init__**" function. These are specialized python functions that allow the programmer to define the **attributes** that describe the object's **state**. In the case of a water container the attributes that define its state would be its **size** and the **amount of water** it initially contains. It is the **__init__** function that actually "creates" the object.

   The first argument of the **__init__** function is the **self** argument. This represents any future water container **object** that would be created using this class. The arguments following the self argument, represent actual values associated with the object's attributes (ie: size and water_contained).

2. **ATTRIBUTE BINDING:** Inside the **__init__** function, the dot operator (".") is used to bind any future object assigned to the **self** argument, to its attributes.

   The "**self.size = size**" statement assigns the value assigned to size to **self.size**, where self.size signifies the attribute "size" of self. Similarly with the "**self.water_contained = water_contained**" statement.

3. **METHOD BINDING:** The subsequent functions inside the class statement define the behaviour of objects that belong to this class. These functions basically modify the state of the object whenever they are used and they are applicable only to objects that belong to this class. Such class bound functions are called methods.

We instantiate or create objects of the Watercontainer class as shown below.

```
container_1 = Watercontainer(size = 100, water_contained = 90)
container_2 = Watercontainer(size = 500, water_contained = 200)
```

Now the variable names container_1 and container_2 represent two Watercontainer objects, with size and water_content of 100, 90 and 500, 200 respectively. Objects belonging to a certain class are called **instances** of that class

Consider the first statement)in the code shown above. When it is executed, a Watercontainer object is instantiated (created) and then assigned to the variable name **container_1** in the following way:

1. Python first uses the **__init__** function to create an "**raw**" instance/object of the class. This raw object has no functionality and is assigned to **self**.

2. It then "activates" the **state** of this **self** object, via the attribute binding statements contained within the **__init__** function. In other words **self** is given two attributes which define its **state** (ie: size and water_contained).
3. Python then binds all the other functions defined within the class statement to **self**. This is called method binding. Functions that are exclusively associated with objects of a certain class are called **methods**.
4. Once the raw object's state and behaviour are "activated" it becomes a "ready to be used" object.
5. This ready to be used object is then assigned to the variable_name container_1.

Now the attributes of the newly instantiated Watercontainer object (container_1), can be accessed using the dot operator and the name of the required attribute, as shown below:

```
container_1.size
```

```
100
```

```
container_1.water_contained
```

```
90
```

Since container_1 is an instance of the class Watercontainer, it also has some **methods** associated with it. These are the **water_in** and **water_out** methods. These methods can applied to it using the **dot** operator in the manner shown below:

```
container_1.water_out(30)
container_1.water_in(10)
```

Quantity of water contained in container_1 has reduced by 20

```
container_1.water_contained
```

```
70
```

The same operations can be applied to container_2 also. Note that **methods** are actually functions and so they have curved brackets that (may) contain arguments. **Attributes** are not functions, but are just a special kind of **variable name** that have values assigned to them. So, using the **dot** operator on object instances we can:

1. Make queries about the state of the object (ie: Inspect the attributes of the object).
2. Use the inherent methods related to its class to modify it (ie: perform some task/computation).

Going a little deeper, we realize that a water container should also have the following properties:

1. They cannot be filled more than their capacity
2. You cannot take water from an empty container

These properties can be incorporated into the Watercontainer class by modifying the code as shown below:

```
class Watercontainer():

    def __init__(self, size, water_contained):

        self.size = size
        self.water_contained = water_contained


    def water_in(self, water_in):

        if self.water_contained + water_in > self.size:
            print('excess water')
            self.water_contained = self.size
        else:
            self.water_contained = self.water_contained + water_in


    def water_out(self, water_out):

        if self.water_contained - water_out <= 0:
            print('container empty')
            self.water_contained = 0
        else:
            self.water_contained = self.water_contained - water_out
```

Now any instantiated Watercontainer object will exhibit the behaviour expressed below:

```
container_1 = Watercontainer(size = 50, water_contained = 40)

container_1.water_contained
```

```
40
```

```
container_1.water_in(30)
```

```
excess water
```

```
container_1.water_contained
```

```
50
```

```
container_1.water_out(60)
```

```
container empty
```

```
container_1.water_contained
```

```
0
```

Consider an another example - The class definition for creating "**financial account** objects" would be as shown below:

```
class Account():

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def deposite(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        self.balance = self.balance - amt
```

```
account_1 = Account('bob', 10000)

account_1.name, account_1.balance
```

```
('bob', 10000)
```

```
account_1.deposite(50000)
account_1.withdraw(5000)

account_1.balance
```

```
55000
```

INBUILT CLASSES IN PYTHON:

Object Oriented Programming is a vast subject by itself, capable of creating data objects which encompass complex states and behaviours. In fact all of the python data objects used till now (integers, floats, strings, lists, tuples, dictionaries and sets) were actually instances of predefined python classes. So for example, when we created a list by enclosing values within square brackets, what is actually happening is that an instance of the list class is created and the sequence of values entered within the square bracket defines its state.

The discussions on OOP covered in this chapter are just an introduction to the subject. They serve the basic purpose of familiarizing the reader with:

1. The concept of classes and objects/instances.
2. The concept of state and behaviour of an object (attributes and methods).
3. The underlying nature and structure of all data objects within python.

THE TYPE FUNCTION:

Python has a builtin function, the **type function** which returns the name of the class (ie: type) of any object fed to it as an argument. Shown below is the type function applied to all the common data types in python.

```
type(1), type(1.1), type('hello')
```

```
(int, float, str)
```

```
l = [1,2,3]
t = (1,2,3)
d = {'a': 1, 'b': 2}
s = {1,2,3}

type(l), type(t), type(d), type(s)
```

```
(list, tuple, dict, set)
```

INTEGER AND FLOAT CLASSES:

Everytime we assign an integer/float object to a variable name in python, what actually happens is that an instance of the **int/float** class is first instantiated and then that object is assigned to the variable name. The code below demonstrates this:

```
i = int()
i
```

```
0
```

```
i, ii = int(1), int(2.27)

i, ii
```

```
(1, 2)
```

```
f = float()
f
```

```
0.0
```

```
f, ff = float(1.1), float(227)

f, ff
```

```
(1.1, 227.0)
```

As can be inferred from the code above, if during instantiation of an **int object**, the number fed to the **int class** is a float, it only uses the non fractional part of the float number to instantiate the int object. Similarly during the instantiation of a float **object**, if the number fed to the **float class** is an integer, the object returned will be the float version of that integer.

Strings cannot be used while instantiating an **int** or **float** object, unless the strings used are actually numerical values between quotation marks. The behaviour of str class during object instantiation is demonstrated in the code shown below:

```
int('22'), float('22'), float('22.2')
```

```
(22, 22.0, 22.2)
```

```
int('22.2')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-54f11d8fd4a2> in <module>
----> 1 int('22.2'), float('22')

ValueError: invalid literal for int() with base 10: '22.2'
```

```
int('hello')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-35-1e0cfe654e81> in <module>
----> 1 int('hello')

ValueError: invalid literal for int() with base 10: 'hello'
```

```
float('hello')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-36-879c4bea9d62> in <module>
----> 1 float('hello')

ValueError: could not convert string to float: 'hello'
```

STRING CLASS:

As with integers and floats, everytime a string value is assigned to a variable name, what actually happens is that an instance of the **str** class is first instantiated and then that object is assigned to the variable name. The code below demonstrates this:

```
s = str()
s
```

```
''
```

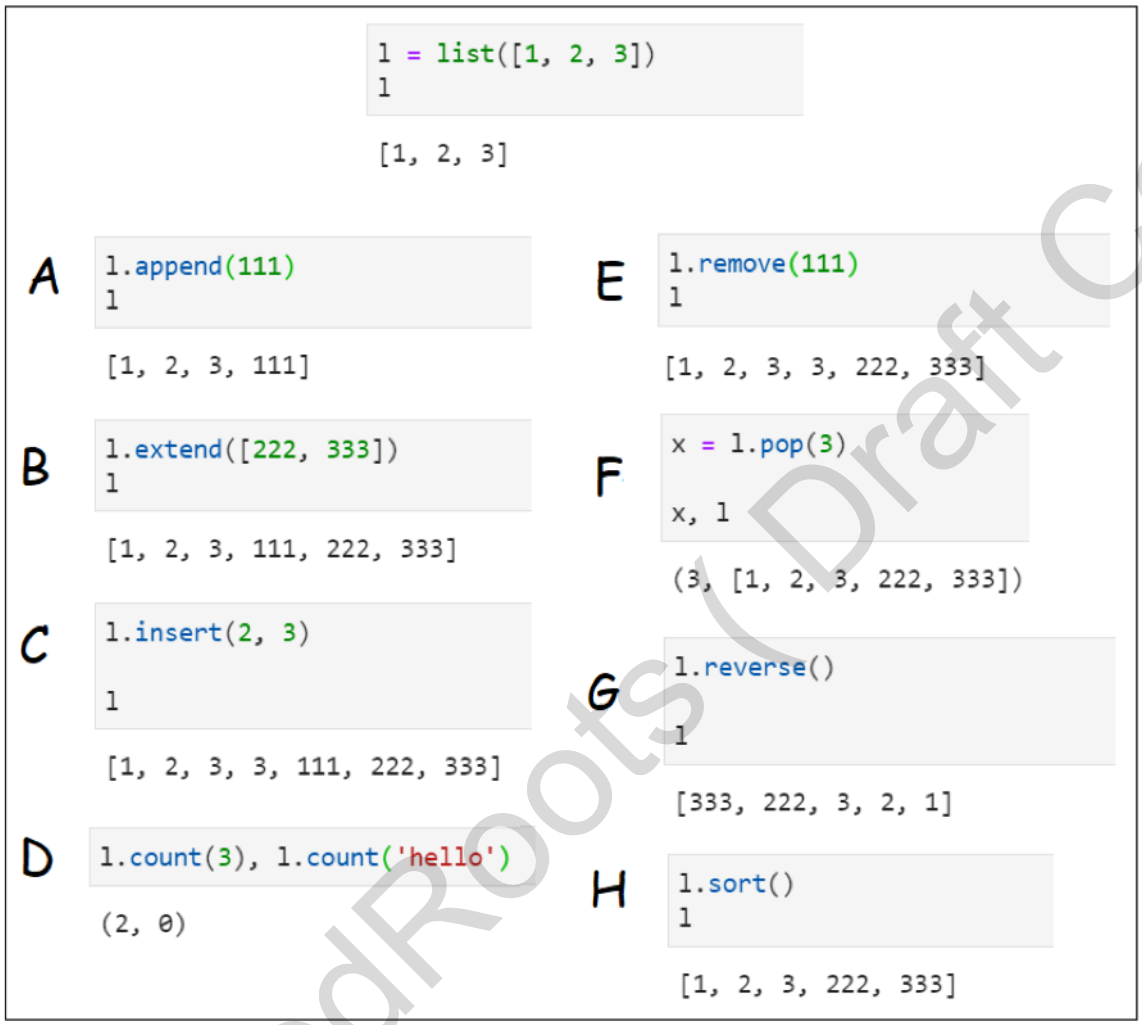
```
s, ss, sss = str('hello'), str(22.7), str(227)
s, ss, sss
```

```
('hello', '22.7', '227')
```

As can be inferred from the code above, if during instantiation of an **str object**, the value fed to the **str class** is an integer or a float, it converts the integer/float to **str** object.

LIST METHODS:

All list objects have predefined **methods** associated with them, shown below are some of the most useful/relevant ones:



Block A shown above demonstrates the **append** method. The append method allows one to add one more object to the end of the list.

Block B demonstrates the **extend** method. Th extend method allows one to concatenate another list to the end of an already existing one.

Block C demonstrates the **insert** method. The insert method allows one to insert a new value into a list at a specific location (index). The first argument of this method corresponds to the required index and the second argument corresponds to the value to be inserted.

Block D demonstrates the **count** method. The count method allows one to get a count of the number of times a particular value appears within a list.

Block E demonstrates the **remove** method. The remove method allows one to remove a particular value from a list.

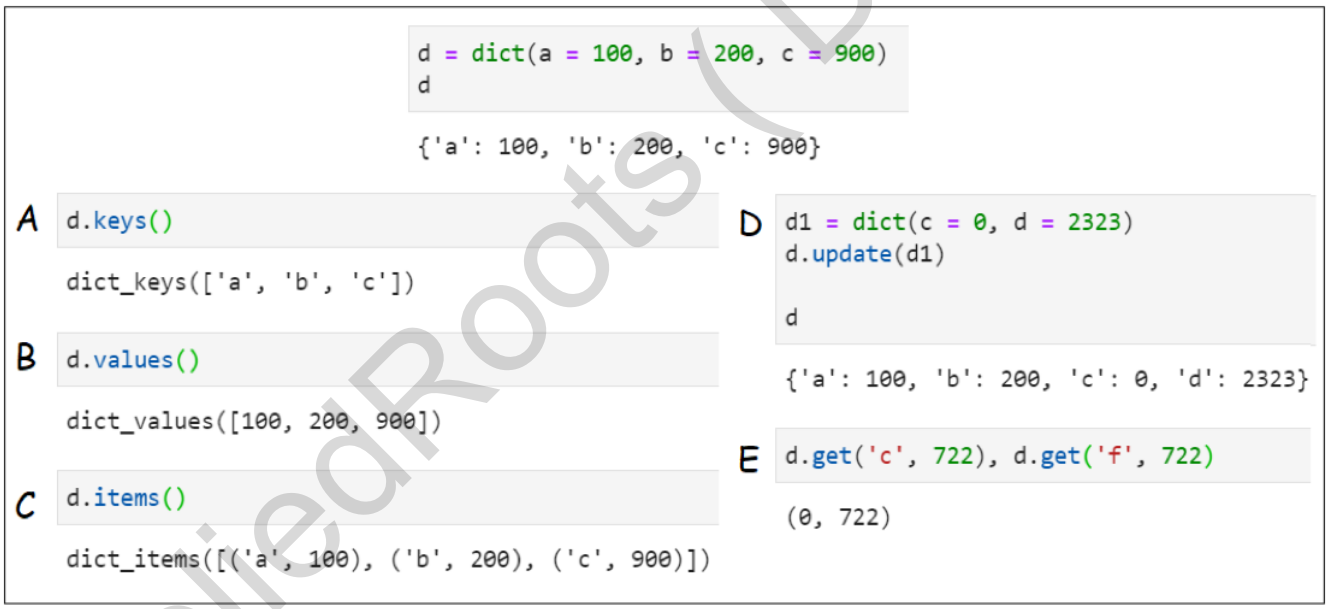
Block F demonstrates the **pop** method. The pop method is similar to the remove method, except that it returns the value that was removed from the list.

Block G demonstrates the **reverse** method. The reverse method allows one to reverse the order of the elements within a list.

Block H demonstrates the **sort** method. The sort method allows one to sort a list containing numerical values in ascending order.

DICT METHODS:

All dictionary objects have predefined methods associated with them, shown below are some of the most useful/relevant ones:



Dicts can also be instantiated/created using the format shown above by directly using the dict class. Inside the dict class key value pairs are defined using assignment statements that represent each pair.

Block A shown above demonstrates the **keys** method. The keys method returns all the keys of the dictionary it is applied to.

Block B shown above demonstrates the values method. The values method returns all the values of the dictionary it is applied to.

Block C shown above demonstrates the items method. The items method returns all the key-value pairs of the dictionary it is applied to. It returns these pairs enclosed within tuples.

Block D shown above demonstrates the update method. The update method updates the key value pairs of the dict it is applied to with those of another dict.

Block E shown above demonstrates the **get** method. The get method returns the value of the key referred to in the first argument of the method. If this key does not exist in the dict it is applied to, it returns the second argument of the method instead.

THE TUPLE AND SET CLASSES:

The **tuple** class is very similar to **list** class except that they do not have any state altering methods contained within them. All indexing and slicing protocols are the same as those for lists. The tuple class takes in a list as its argument and converts it into a tuple as shown below:

```
t = tuple([1, 2, 2, 2, 4, 5])

t
```

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

The **set** class is similar to lists in that they are mutable, but they do not allow for repetitions of the objects within them and they are unordered. In other words they represent an unordered collection of unique objects. It also requires that the objects enclosed within them be immutable. The set class too takes in a list as its argument and converts it into a set. All these characteristics of objects of the set class are shown below:

```
s = set([1, 2, 2, 2, 4, 5])

s
```

```
{1, 2, 4, 5}
```

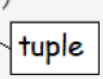
```
s = set([1, 2, 2, 2, [4, 5]])

s
```

list

sets can only contain immuatable objects

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-139-071181ce4b8a> in <module>
----> 1 s = set([1, 2, 2, 2, [4, 5]])
      2
      3 s

TypeError: unhashable type: 'list'
```

```
s = set([1, 2, 2, 2, (4, 5)])

s
```




```
{(4, 5), 1, 2}
```

Sets basically interact with other sets creating new sets. Each of the set **operators** discussed previously have **method counterparts** as shown below.

```
s1 = {44, 1, 2, 55, 99}
s1
```

```
{1, 2, 44, 55, 99}
```

```
s.union(s1)
```

```
{(4, 5), 1, 2, 44, 55, 99}
```

```
s.intersection(s1)
```

```
{1, 2}
```

```
s.difference(s1), s1.difference(s)
```

```
({(4, 5)}, {44, 55, 99})
```

```
s.symmetric_difference(s1)
```

```
{(4, 5), 44, 55, 99}
```

Values can be added to and removed from sets using the add and remove set methods as shown below:

```
s1.add(7)

s1
```

```
{1, 2, 7, 44, 55, 99}
```

```
s1.remove(7)

s1
```

```
{1, 2, 44, 55, 99}
```

This concludes the fourth chapter, in the next chapter we shall delve deeper into the concept of **iterators** and we shall also explore the concept of **generators**.

5. ITERATORS, GENERATORS & COMPREHENSIONS

ITERATORS, GENERATORS & COMPREHENSIONS

In the previous chapters, we explored the concept of iteration using python's **for** loop. In this chapter we shall delve a little deeper into iteration and how it is internally implemented in python. Iteration is also core to one of python's most powerful tools: Generators. Generators help us to create and design custom **iterators**. This helps us build highly scalable python applications/programs that can handle large datasets in performant and efficient ways.

THE ITER AND NEXT INBUILT FUNCTIONS:

Python has two inbuilt functions **iter** and **next**, based on which the **for** loop operates. The **iter** function takes in a container object as its argument and creates an **iterator** object. The **iterator** is a separate object with its own identity as shown in the code below:

```
l = [122, 44, 77, 9]

l_iterator = iter(l)
```

```
l_iterator
```

```
<list_iterator at 0x204871b2610>
```

```
l_iterator is l, l_iterator == l
```

```
(False, False)
```

```
type(l), type(l_iterator)
```

```
(list, list_iterator)
```

Iterator objects are such that they produce the sequence of objects embedded within it, one at a time, each time the next function is applied to it. This is shown in the code below.

```
l = [98,76,54]

l_iterator = iter(l)
```

```
next(l_iterator)
```

```
98
```

```
next(l_iterator)
```

```
76
```

```
next(l_iterator)
```

```
54
```

```
next(l_iterator)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-15-04042345ccda> in <module>
----> 1 next(l_iterator)

StopIteration:
```

In other words, the **next** function makes the **iterator** object "**pop**" the first item of the sequence contained within it each time it is executed. This means that each time the **next** function is executed, the first object within the **iterator** is **ejected** from the sequence and **returned** to the user to be used as necessary. The iterator now contains only the un-ejected part of the initial sequence. This behaviour can be replicated until the sequence contained within the iterator is exhausted. If the **next** function is used again, python raises a **StopIteration** exception as shown above.

ITERABLES & ITERATORS :

Any container object to which the **iter** function can be applied is called an **iterable**. Lists, tuples and Dicts are **iterables**. The python object that results from applying the iter function on an iterable is called an **iterator**.

THE FOR LOOP DECONSTRUCTED:

Though the **for** loop in python seems quite simple to construct, there are number steps that happen under the hood when it is executed. Consider the for loop used in the code below.

```
l = [1 , 22, 333, 444]
l_even, l_odd = [], []

for i in l:
    if i%2  == 0:
        l_even.append(i)
    else:
        l_odd.append(i)

print(l_even)
print(l_odd)
```

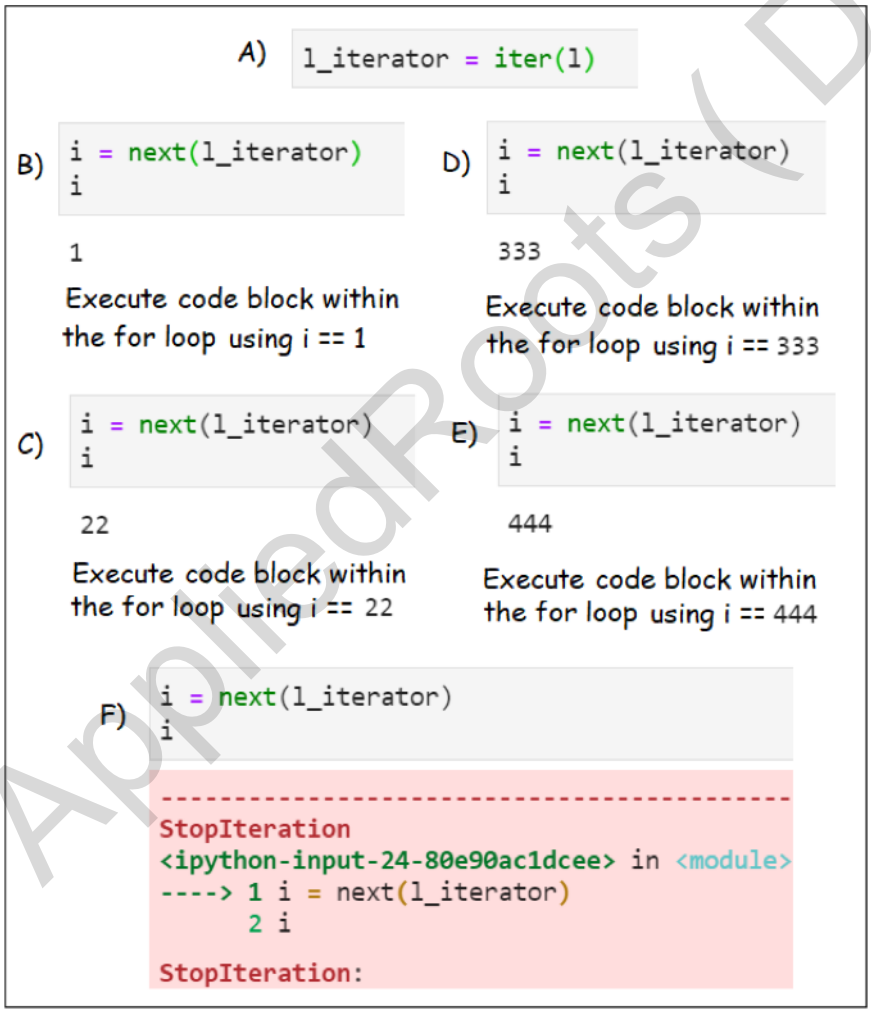
Code which is to be executed for every value in the sequence contained within the list l

```
[22, 444]
[1, 333]
```

When the for loop shown above is executed, the following steps happen under the hood:

1. An iterator object is first created by applying the iter function on list l.
2. The next function is then called on the iterator object.
3. The object returned by the next function is assigned to the variable name i.
4. The value assigned to i is then used within the code block belonging to the for loop.
5. Steps 2, 3 & 4 are repeated until the sequence embedded within the iterator object is exhausted and python raises a StopIteration exception.
6. When python raises a StopIteration exception, the execution flow within the for loop is stopped and the python interpreter exits the for loop.
7. After exiting the for loop, the interpreter continues executing any other code that exists outside the for loop.

The steps discussed above are expressed in the image image shown below:



Thus the **for** loop uses the **iter** and next **functions** under the hood to create looped executions and it makes use of the **StopIteration** exception to stop the looped execution and return to the code that is outside the **for** loop.

THE WHILE LOOP:

Apart from the **for** loop, python has one more form of iteration called the **while** loop. The basic form of a while loop is demonstrated in the function shown below:

```
def fn_sequence(max_val):

    l  = []
    val = 0

    while val <= max_val:

        l.append(val)
        val = val + 1

    return l
```

Condition

Code block belonging to the **while** statement

```
fn_sequence(5)
```

```
[0, 1, 2, 3, 4, 5]
```

The while loop uses the **while** keyword **followed** by a **conditional statement** as shown above. This forms the **while statement**. All indented code from the next line after the **colon** symbol, at the end of the **while statement**, belongs to the while statement.

The **while** statement executes the code block belonging to it, repetitively, as long as the **conditional statement** holds true (ie: **val < = max_val == True**). In other words, the **while** statement checks the validity of the **conditional** statement each time after executing the code block belonging to it. After successive iterations/executions of this code block, if the **conditional** statement becomes false (ie: **val < = max_val == False**), then the execution flow within the **while** loop is broken and the interpreter exits the loop. After exiting the **while** loop, the interpreter continues executing any other code that exists outside the loop.

The function shown below is another example of the usage of **while** loops. It creatings fibonacci sequences.

```
def fn_fibo(quantity):

    l_fibo = []
    a, b = 0, 1
    while len(l_fibo) < quantity:

        a, b = b, a + b
        l_fibo.append(a)

    return l_fibo
```

```
fn_fibo(10)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Note that each value in a fibonacci sequence is the sum of the previous two values, 0 & 1 being the first two values.

UNPACKING CONTAINERS USING THE STAR & DOUBLE STAR OPERATORS:

The **star** symbol apart from being used to represent the multiplication operator, can be used to unpack iterable sequence containers (ie: lists & tuples) into other sequence containers as shown below:

```
list_1, tup_1 = [1, 2, 3], (11, 22, 33)

l = [*list_1, *tup_1]
l
```

```
[1, 2, 3, 11, 22, 33]
```

The double star operator can be used to unpack dictionaries into any other dictionary as shown below:

```
dict_1 = dict(a = 1, b = 2)
dict_2 = dict(c = 11, d = 22)

d = {**dict_1, **dict_2}
d
```

```
{'a': 1, 'b': 2, 'c': 11, 'd': 22}
```

CREATING UNIQUE COMBINATIONS USING VALUES FROM MULTIPLE LISTS:

Consider the task of collecting all possible unique combinations of values that exist in a given collection of lists. The combinations should be from elements of one list with elements of other lists and not between elements within a list.

Let us start with a collection of two lists. We can collect unique combinations between them by using a nested for loop as shown below:

```
l1, l2 = [1,2,3], ['a', 'b', 'c']

combo = []
for v in l1:
    for u in l2:
        combo.append([v, u])

combo
```

```
[[1, 'a'],
 [1, 'b'],
 [1, 'c'],
 [2, 'a'],
 [2, 'b'],
 [2, 'c'],
 [3, 'a'],
 [3, 'b'],
 [3, 'c']]
```

As can be seen from the output of the code above, we have collected all possible combinations of the values between the given two lists.

Now let us consider the case of more than two lists. To do this we implement the following steps in python:

1. The collection of lists is first enclosed within a list called **listO_lists.**
2. We then take the first two lists belonging to **listO_lists** and perform the nested loop computation just described earlier. Thus getting a list that contains combinations of list_1 and list_2 in the same form as shown above (ie: output of the nested loop shown above).
3. We remove the two lists used in the previous step from **listO_lists.**
4. We add the new list created in step 2 to **listO_lists** as its first element (ie: at index = 0).
5. Repeat step 2 using this modified **listO_lists.**
6. Execute steps 2, 3, 4 & 5 repetitively until all the lists within **listo_lists** are exhausted.

The above steps are implemented by creating two functions - **fn_combo and fn_pop_n_collect** which perform the following tasks:

1. **fn_combo:** This function executes step 2 shown above.
1. **fn_pop_n_collect:** This function executes the steps 3, 4, 5 & 6 shown above.

The two functions are implemented in the manner shown below:

```
def fn_combo(l1, l2):

    combo = []
    for v in l1:
        for u in l2:
            combo.append([v, u])

    return combo
```

```
def fn_pop_n_collect(list0_lists):

    list0_lists = list0_lists.copy()
    while len(list0_lists) > 1:

        l1 = list0_lists.pop(0)
        l2 = list0_lists.pop(0)

        combo_list = fn_combo(l1, l2)
        list0_lists.insert(0, combo_list)

    return combo_list
```

Remove/pop/eject the first two lists from list0_lists and assign them to l1 & l2 — A

Apply fn_combo on l1 & l2 to get a list containing unique combinations of elements (ie: combo_list) — B

Insert combo list as the first element of list0_lists. — C

As can be seen from the code above, the function **fn_pop_n_collect** uses **fn_combo** within itself. It takes in **list0_lists** as its argument and uses **fn_combo** to create unique combinations. It uses a **while** loop with the condition **len(list0_lists) > 1** to repetitively perform the tasks **A, B & C** described in the image above.

We test the above two functions in the manner shown below:

```
list0_lists = [[1,2,3], ['a', 'b', 'c'], [22, 33]]

combos = fn_pop_n_collect(list0_lists)
combos
```

```
[[[1, 'a'], 22],
 [[1, 'a'], 33],
 [[1, 'b'], 22],
 [[1, 'b'], 33],
 [[1, 'c'], 22],
 [[1, 'c'], 33],
 [[2, 'a'], 22],
 [[2, 'a'], 33],
 [[2, 'b'], 22],
 [[2, 'b'], 33],
 [[2, 'c'], 22],
 [[2, 'c'], 33],
 [[3, 'a'], 22],
 [[3, 'a'], 33],
 [[3, 'b'], 22],
 [[3, 'b'], 33],
 [[3, 'c'], 22],
 [[3, 'c'], 33]]
```

As can be seen from the output above, the combinations are presented as nested lists. This is because:

1. **fn_combo** is designed to take in two simple lists (ie: list containing only values - not other lists) and return a list containing lists of various combinations possible between these two simple lists.
2. For the first iteration of the while loop in **fn_pop_n_collect, fn_combo** receives two simple lists as its arguments.
3. But from the second iteration onwards, the first list argument that **fn_combos** receives will be a compound list containing other lists and the second list argument will be a simple list. This causes the nested list output.

The nested output can be avoided by modifying fn_combos, by using the star unpacking operator to unpack the first list argument (l1) if it is a list. This is shown in the code below.

```
def fn_combo(l1, l2):

    combo = []
    for v in l1:
        for u in l2:
            if type(v) == list:
                combo.append([*v, u])
            else:
                combo.append([v, u])

    return combo
```

This statement executed after the first iteration

This statement is executed only for the first iteration

Now we get the output in the form that we desire:

```
listO_lists = [[1,2,3], ['a', 'b', 'c'], [22, 33]]

combos = fn_pop_n_collect(listO_lists)
combos
```

```
[[1, 'a', 22],
 [1, 'a', 33],
 [1, 'b', 22],
 [1, 'b', 33],
 [1, 'c', 22],
 [1, 'c', 33],
 [2, 'a', 22],
 [2, 'a', 33],
 [2, 'b', 22],
 [2, 'b', 33],
```

```
[2, 'c', 22],
[2, 'c', 33],
[3, 'a', 22],
[3, 'a', 33],
[3, 'b', 22],
[3, 'b', 33],
[3, 'c', 22],
[3, 'c', 33]]
```

This type of program is used for grid search in Machine Learning operations.

GENERATOR FUNCTIONS:

A **generator function** is a python **code construct** very similar to **functions**. **Generator functions** allow us to create **custom iterators**. In the image shown below compare the function for creating sequences **fn_sequence** (left) with the code shown on the right (**gen_sequence**).

```
def fn_sequence(max_val):

    l  = []
    val = 0

    while val <= max_val:

        l.append(val)
        val = val + 1

    return l
```

```
def gen_sequence(max_val):

    val = 0
    while val <= max_val:

        yield val
        val = val + 1
```

The yield statement

The code construct on the right side of the image above is called a **generator function**. It differs from a **function** in the way that it outputs its values. Unlike the normal function on the left, which **returns** a list of values when **called**, the generator on the right **yields** one value at a time, each time the **next** function is applied to it. This is shown below

A
```
g = gen_sequence(3)
```

B
```
i  = next(g)
i
```
0

C
```
i  = next(g)
i
```
1

D
```
i  = next(g)
i
```
2

E
```
i  = next(g)
i
```
3

F
```
i  = next(g)
i
```

```
---------------------------------------------
StopIteration
<ipython-input-77-68ae0c3d3f48> in <module>
----> 1 i  = next(g)
      2 i
StopIteration:
```

The main difference between functions and generators is that generators do not use the return statement, they use the yield statement instead.

When we call a generator function as shown below, python returns a generator object and this generator object is assigned to the variable name specified (ie: g).

```
g = gen_sequence(3)
```

When the **next** function is applied repetitively to the generator object assigned to variable name **g**, the following things happen:

1. When the next function is applied to **g** for the first time, the code within **gen_sequence** gets executed till it reaches the **yield** statement. When the yield statement is executed, it returns the value specified after the **yield keyword** and pauses the execution flow at that point.
2. The next time the **next** function is **called** on the generator object **g**, the code resumes execution from where it was stopped. In other words, any remaining code after the **yield statement** is executed and the **while** loop resumes till the yield statement is encountered again. At this point the value specified after the **yield keyword** is returned and the execution flow paused again.
3. Step 2 can be repeated until python raises the **StopIteration** error, when the **condition** specified after the **while** keyword no longer holds true.

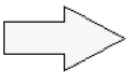
As can be inferred from the behaviour of generator objects when the next function is applied to them, **generator objects** are basically custom made **iterators** and so it can be iterated using a **for** loop just like any other iterator object. This is expressed in the code shown below:

```
g = gen_sequence(5)
for i in g:
    print(i)
```

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

Consider the function **fn_fibo** created earlier. We can create a generator function that generates fibonacci numbers as shown below.

```
def fn_fibo(quantity):

    l_fibo = []
    a, b = 0, 1
    while len(l_fibo) < quantity:

        a, b = b, a + b
        l_fibo.append(a)

    return l_fibo
```



```
def gen_fibo(quantity):

    count = 0
    a, b = 0, 1
    while count < quantity:

        a, b = b, a + b
        yield a
        count = count + 1
```

We can use the generator **gen_fibo** just created as shown below:

```
fibo = gen_fibo(6)
for i in fibo:
    print(i)
```

```
1
1
2
3
5
8
```

The usefulness of generators becomes more apparent when we need to iterate through large sequences. Say, we wanted to determine how many even and odd numbers exist within hundred fibonacci numbers. In the absence of generators, we would have to do the following:

1. Use a function like **fn_fibo** to create a sequence of hundred fibonacci numbers.
2. Iterate through the fibonacci sequence created above and use the **segregate and count** pattern to count the odd and even numbers as shown below.

```
l_even, l_odd = [], []

fibo_list = fn_fibo(6)
for i in fibo_list:

    if i%2 == 0:
        l_even.append(i)
    else:
        l_odd.append(i)
```

The method used above is feasible if the sequence being iterated through is small, but let us say we want to iterate through a fibonacci sequence that has a length in the order of millions of numbers. Then the list that **fn_fibo** returns will be huge and will have a significant memory footprint.

Generators on the other hand have no memory footprint. They do not contain any data, but instead they generate data one value at a time. This makes them extremely advantageous. One could design a custom iterator/generator (**gen_fibo**) using the format outlined in the above discussions and iterate over it as shown below, without needing to worry about the memory footprint.

```
count_even, count_odd = 0, 0

fibo_gen = gen_fibo(100000)
for i in fibo_gen:

    if i%2 == 0:
        count_even = count_even + 1
    else:
        count_odd = count_odd + 1

count_even, count_odd
```

```
(33333, 66667)
```

INBUILT GENERATORS/ITERATORS:

Python has some inbuilt iterators that make creating loops a lot more easier, They are:

ENUMERATE:

The enumerate iterator function takes in an iterable object (list/tuple/dict) and returns tuples containing the value being currently yielded and its count or index value as shown below.

```
l = [11,22,33]

for idx, i in enumerate(l):
    print(idx, i)
```

```
0 11
1 22
2 33
```

```
d = dict(a = 1, b = 22, c = 33)

for idx, i in enumerate(d):
    print(idx, i)
```

```
0 a
1 b
2 c
```

ZIP:

The zip iterator allows us to iterate through multiple equi-sized containers simultaneously as shown below:

```
l1, l2, l3 = [1 ,2 ,3], [11, 22, 33], ['a', 'b', 'c']

for i, j, k in zip(l1, l2, l3):
    print(i, j, k)
```

```
1 11 a
2 22 b
3 33 c
```

RANGE:

The range iterator function is an iterator that yields sequences based on the **start, end** and **step size** values fed to it as shown below.

```
start, end, step = 0, 10, 3

for i in range(start, end, step):
    print(i)
```

```
0
3
6
9
```

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

As can be seen from the code on the right in the image above, the range iterator uses default values of zero for the start argument and one for the step argument.

COMPREHENSIONS:

A comprehension is a high level declarative way of creating sequences in python. It enables one to perform an iteration and collect the required values from the iteration in a single step. Depending on the container in which we wish to collect the iterated objects, there are two kinds of comprehensions:

LIST COMPREHENSIONS:

The format for the list comprehension is described below on the right in comparison with a normal for loop.

```
l = []
for i in range(5):
    l.append(i)

l
```

```
[0, 1, 2, 3, 4]
```

```
l = [i for i in range(5)]
l
```

```
[0, 1, 2, 3, 4]
```

Note that for the normal loop we have to specify how the resultant list needs to be created, whereas in case of the list comprehension we declare what kind of list we want.

Conditions could be used within list comprehensions as shown below:

```
l = []
for i in range(10):
    if i%2 == 0:
        l.append(i)

l
```

```
[0, 2, 4, 6, 8]
```

```
l = [i for i in range(10) if i%2 == 0]
l
```

```
[0, 2, 4, 6, 8]
```

```
l = []
for i in range(10):
    if i%2 == 0:
        l.append(i)
    else:
        l.append('odd')

l
```

```
[0, 'odd', 2, 'odd', 4, 'odd', 6, 'odd', 8, 'odd']
```

```
l = [i if i%2 == 0 else 'odd' for i in range(10)]

l
```

```
[0, 'odd', 2, 'odd', 4, 'odd', 6, 'odd', 8, 'odd']
```

DICT COMPREHENSIONS:

Similar to list comprehensions we can also create dictionary comprehensions as shown below;

```
var = ['x', 'y', 'z']
x = [1, 2, 3]
y = [1.1, 2.2, 3.3]
z = [11, 22, 33]

d = {k:v for k, v in zip(var, [x, y, z])}
d
```

```
{'x': [1, 2, 3], 'y': [1.1, 2.2, 3.3], 'z': [11, 22, 33]}
```

GENERATOR COMPREHENSIONS:

Simple generators can be created by converting list comprehensions to generator comprehensions as shown below:

```
l = [i if i%2 == 0 else 'odd' for i in range(10)]

g = (i if i%2 == 0 else 'odd' for i in range(10))
```

Replace square brackets with curved brackets

```
next(g), next(g), next(g), next(g)
```

```
(0, 'odd', 2, 'odd')
```

The usefulness of creating such generators becomes more apparent in cases where the length of the sequence desired is very large and where the values the sequence contains could get large. This is expressed in the code shown below:

```
l = [i**3 for in in range(1000000000)]

g = (i**3 for in in range(1000000000))
```

If one uses the list **l** shown above, then its memory footprint is going to be quite large, but we can avoid this issue by using generator **g** instead.

This concludes the fifth chapter. In the next chapter we shall introduce the numpy **library** and explore multidimensional container objects called **ndarrays** and the concept of **vectorized operations.**

6. THE NUMPY LIBRARY

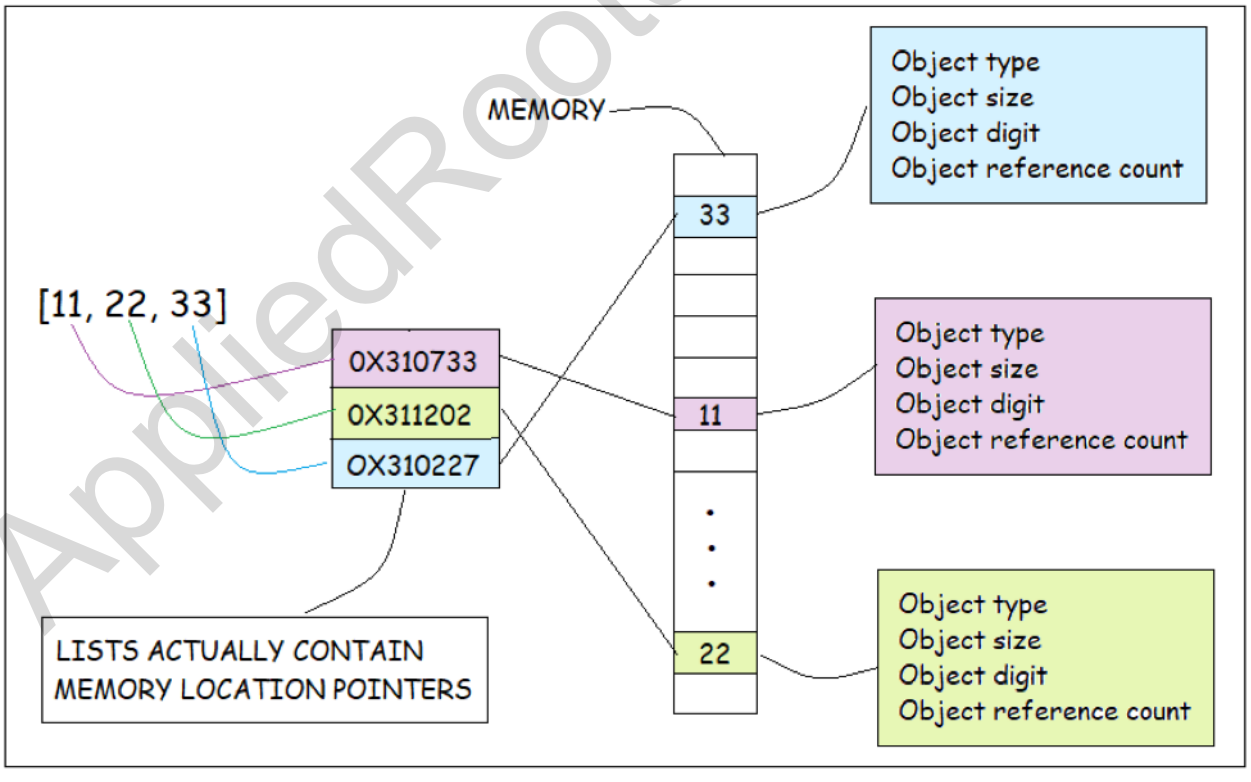
THE NUMPY LIBRARY

NumPy is one of the most important packages/libraries for numerical computing in python. The name NumPy basically is short for Numerical Python. At the core of numpy's functionality is the ndarray (n-dimensional array) data container object, which is capable of containing ordered multi - dimensional data (ie: Data containing multiple axes). Most of numpy's functionality is based on the ndarray class defined within it, which provides a comprehensive set of methods/ functions for manipulating multidimensional numerical data and for performing computations involving multiple instances of ndarrays.

LISTS Vs ARRAYS:

As discussed previously, lists are data container objects that are ordered, mutable and heterogeneous. This makes them extremely flexible and they provide high utility when one wants to just group a collection of objects together without needing to worry about the data types or when one wants to sequentially collect the output of some iterated computation.

The flexibility of lists comes at a price. They are not efficient in terms of numerical processing. Lists actually do not contain any object, but instead they contain mappings/pointers to objects that are stored at various locations within the memory of the computer.

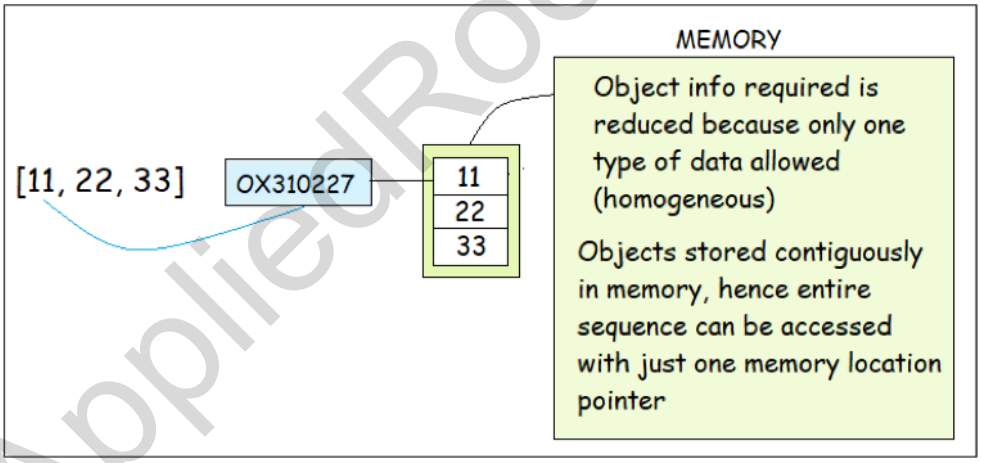


For the sake of simplicity, lets us consider the case where the objects within the list are just integers and floats. Each of these objects will contain the following information:

1. **Object type:** Type of variable
2. **Object size:** Number of digits
3. **Object digit:** The actual number
4. **Object reference count:** Count of the number of references to the object within the code executed till now.

Due to the **non-contiguous** nature of how objects belonging to a list are stored within the memory, they are not fast to iterate/compute through. Also, due to the amount of information each of those objects contain (as discussed above), they tend to use up more memory. In other words because the objects grouped within a list are stored in various locations in the memory, python has to jump around the memory space each time it has to retrieve an object within it. The situation becomes compounded when we consider lists that contain lists or any other container object within them, instead of just integers and floats.

Consider the case if lists were constrained to contain only one type of data (say floats) and that they were **contiguously stored** within the memory (ie: the values reside in the memory in the same sequence, as one continuous block of data). The resulting data type would be a one dimensional basic array. The image shown below describes how data is stored within an array data container object.



Structuring a data container object in the way described above makes it less flexible, but at the same time the values within the resulting data container become easily accessible, thus making computation faster and also since lesser information is required to define the container object, it requires lesser memory space.

Numpy's ndarray data container object, due to its "type" constraint and contiguous memory storage design, enables one to perform vectorized operations (ie: fast computations on all the objects contained within the array without needing to write loops). Vectorized operations are one of the key features of the numpy package. Vectorized operations will be further discussed in the "universal functions" subtopic later on in this chapter.

TOPICS DISCUSSED IN THIS CHAPTER:

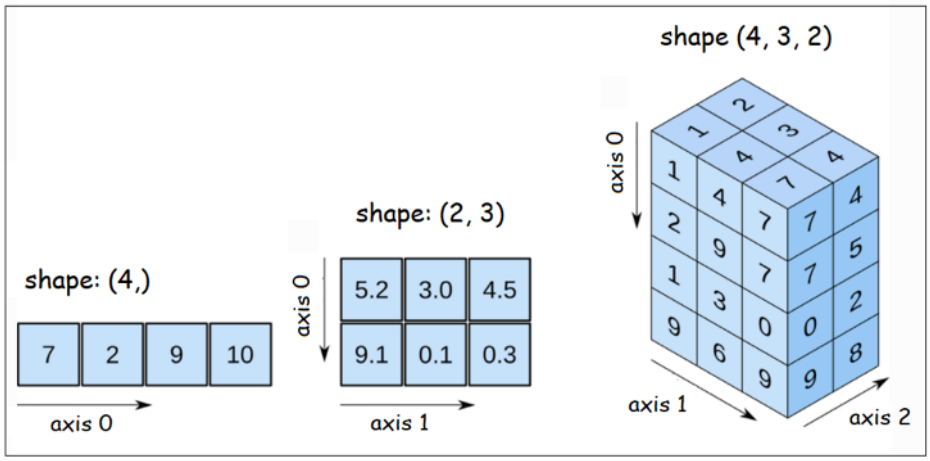
Numpy data objects & classes Creating arrays Array Indexing & Slicing Array attributes & methods	Universal Functions (Ufuncs) Unary Ufuncs Binary Ufuncs Broadcasting Linear Algebra

THE NUMPY OBJECT TYPES:

Numpy defines within itself, its own classes for value and container data types which are independent of and interconvertible with similar python data types. Shown below are some of the relevant value types defined and used within numpy.

TYPE	DESCRIPTION
int8, uint8	8 bits used used to represend signed & unsigned integers
int16, uint16	16 bits used used to represend signed & unsigned integers
int32, uint32	32 bits used used to represend signed & unsigned integers
int64, uint64	64 bits used used to represend signed & unsigned integers
float16	16 bits used used to represend floats
float32	32 bits used used to represend floats
float64	64 bits used used to represend floats
bool	True & False values
string_	1 bit per string character
object	Any python object that is not an int, float, str or bool

As discussed previously, Numpy basically uses one fundamental data container type for all its purposes - the ndarray or array as it is commonly called. An array can contain a homogeneous collection of any of the value types described in the table above. This homogeneous collection of values can be ordered along more than one dimensions as shown below:



IMPORTING LIBRARIES/MODULES:

To use the numpy library, we first have to "load"/import it into our current workspace using the **import** statement as shown below.

```
import numpy as np
```

When a library/package/module is imported, Python runs all of the code in the library/module's source file, inside the current global workspace of the user. Thus all the classes and functions defined within that library become available to the user. The name used to refer to the library/module can be changed using the "**as**" keyword followed by the preferred name as shown above (numpy **as** np). Now all the classes/functions defined within the numpy package/library can be accessed using the **np** keyword in the manner demonstrated in the examples that follow.

CREATING ARRAYS USING PREDEFINED VALUES:

Numpy arrays can be created using the numpy built in functions outlined in the table shown below. These built in numpy functions actually help create instances of the array class, based on various initial requirements of the user.

FUNCTION	DESCRIPTION
np.array	Converts input sequences (lists, tuples, other arrays) into an instance of **ndarray** data type
np.arange	Like python's **range** generator function, except that it returns an array of numpy data types
np.linspace	Creates an array containing the defined number of equispaced values within a given interval
np.ones	Creates an array containing only **ones** as per the dimensions specified
np.ones_like	Creates an array containing only **ones** having the same dimensionality as that of some other array
np.zeros	Creates an array containing only **zeroes** as per the dimensions specified
np.zeros_like	Creates an array containing only **zeroes** having the same dimensionality as that of some other array

1. CREATING AN ARRAY USING A PYTHON LIST:

Arrays can be created using python lists along with the **arange** as shown below:

```
l = [1, 2, 3]

a = np.array(l)
a
```

```
array([1, 2, 3])
```

```
type(a), type(a[0])
```

```
(numpy.ndarray, numpy.int32)
```

As can be seen from the above code, numpy by default uses the **int32** class while instantiating integer objects. If one wants to create an array of a specific data type, they can do so by specifying the preferred data type using the **dtype** keyword argument of the array function as shown below:

```
a1 = np.array(l, dtype = 'int64')
a1
```

```
array([1, 2, 3], dtype=int64)
```

Numpy arrays can contain only the same type of data, so if the list used to create the array has both integer and float type numerical data, the entire data is converted to float type as shown below:

```
l = [0.1, 2, 3]

a = np.array(l)
a
```

```
array([0.1, 2. , 3. ])
```

```
type(a[0])
```

```
numpy.float64
```

As can be seen from the above code, numpy by default uses the **float64** class while instantiating float objects. Multidimensional arrays can be created by using multiple equisized lists within a list as shown below:

```
l = [[1,2], [4,5], [6,7]]

a2 = np.array(l)
a2
```

```
array([[1, 2],
       [4, 5],
       [6, 7]])
```

2. CREATING ARRAYS USING THE ARANGE FUNCTION:

The **arange** function is similar to python's range generator function except that it creates an array of values of the specified range:

```
a = np.arange(5)
a
```

```
array([0, 1, 2, 3, 4])
```

```
a1 = np.arange(5, 20, 3)
a1
```

```
array([ 5,  8, 11, 14, 17])
```

3. CREATING ARRAYS USING THE LINSPACE FUNCTION:

The linspace function can be used to create the required number of equispaced values within a defined interval/range as shown below:

```
start, end, num_of = 22, 22.5, 3

a = np.linspace(start, end, num_of)
a
```

```
array([22.  , 22.25, 22.5 ])
```

```
a1 = np.linspace(0, 3, 5)
a1
```

```
array([0.  , 0.75, 1.5 , 2.25, 3.  ])
```

4. CREATING ARRAYS CONTAINING ONLY ONES:

The **ones** and **ones_like** functions can be used to create arrays containing only ones. This is done as shown below:

num of values

```
a = np.ones(5)
a
```

```
array([1., 1., 1., 1., 1.])
```

shape

```
a1 = np.ones([2, 4])
a1
```

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

As can be seen from the code on the right, multidimensional arrays containing ones can be created using the **ones function**, by defining the shape required within square brackets, as the function's argument.

The ones_like function creates arrays containing only ones, having the same dimensions as the reference array fed to it as its argument. This is demonstrated in the code shown below:

```
l = [1 ,2 ,3, 4]
a2 = np.ones_like(l)

a2
```

```
array([1, 1, 1, 1])
```

```
a3 = np.ones_like(a1)
a3
```

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

5. CREATING ARRAYS CONTAINING ONLY ZEROES:

The zeros and zeros_like functions are very much similar to the ones and ones_like functions just described above, except that it creates arrays containing only zeros:

```
a4 = np.zeros([3, 4])
a4
```

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
a5 = np.zeros_like(a4)
a5
```

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

CREATING ARRAYS USING RANDOM NUMBER GENERATION:

Arrays can be created using numpy's pseudorandom number generation capabilities. This is made possible through the **random** module existing within the numpy library.

1. RAND FUNCTION:

Arrays containing **random floats** between zero and one can be created using the **rand** function as shown below

```
a = np.random.rand(3)
a
```

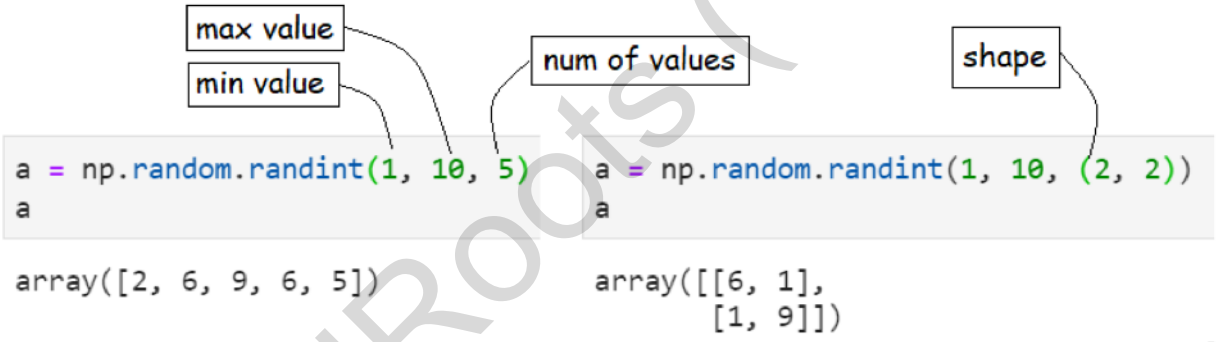
```
array([0.43615486, 0.51981458, 0.0604121 ])
```

```
a = np.random.rand(2, 2)
a
```

```
array([[0.09519936, 0.36363313],
       [0.56144661, 0.39619969]])
```

2. RANDINT FUNCTION:

Arrays containing **random integers** can be created using the **randint** function as shown below:



3. RANDN FUNCTION:

Arrays containing random numbers sampled from a standard **gaussian/normal distribution** can be created using the **randn function** as shown below.

```
a = np.random.randn(3)
a
```

```
array([-1.61786,  1.97005, -0.62602])
```

```
a = np.random.randn(2, 3)
a
```

```
array([[-1.0260609 , -0.87691149, -1.00449392],
       [ 0.80465142, -1.23976103,  0.3827204 ]])
```

Numpy's **random module** has the capability of producing arrays that contain random samples from a wide variety of **probability distributions**. We shall discuss this topic further in the chapter: **Matplotlib – Data Visualization**.

INDEXING AND SLICING:

Indexing numpy arrays is quite similar to the indexing protocol used for python lists (ie: values within an array are indexed starting from zero onwards). All the indexing/slicing rules for a one dimensional array are the same as that for python lists. This is demonstrated in the code below:

```
a = np.array([1, 2, 3, 11, 22])

a[0], a[3], a[-1]
```

```
(1, 11, 22)
```

```
a[2:-1]
```

```
array([ 3, 11])
```

```
a[2:]
```

```
array([ 3, 11, 22])
```

```
a[:-2]
```

```
array([1, 2, 3])
```

To understand indexing of arrays that have more than one dimension, consider the two dimensional array shown below.

```
a = np.random.randint(1, 100, [3, 4])
a
```

```
array([[40, 33, 39, 12],
       [ 2, 47, 37, 46],
       [69, 99, 13, 60]])
```

The above array can be perceived as an array containing three homogeneous objects (ie: three equized arrays containing integers). Using the same indexing rules as before we can access each of these objects individually as shown below:

```
a[0]
```

```
array([40, 33, 39, 12])
```

```
a[-1]
```

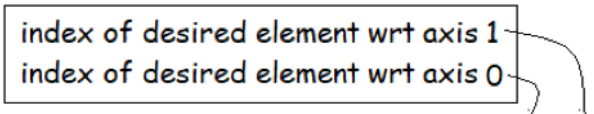
```
array([69, 99, 13, 60])
```

Following the same indexing logic, elements within the arrays that compose the multidimensional array can then be accessed as shown below:

```
a[0][0], a[0][-1], a[-1][0], a[-1][-1]
```

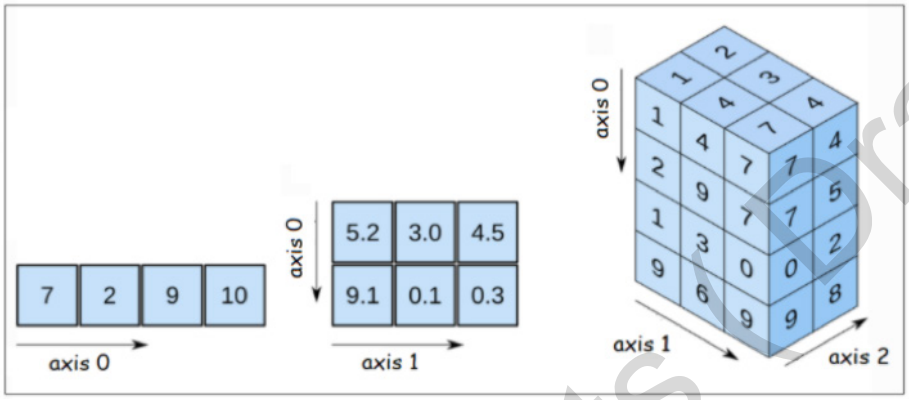
```
(40, 12, 69, 60)
```

Numpy simplifies the above indexing pattern by defining the "**position**" of each value in an array by using its index value it has with respect to each of the axes of the array. Thus each element in a two dimensional array will have two indexes associated with it, one for each axis as shown below.




```
a[0, 0], a[0, -1], a[-1, 0], a[-1, -1]
```

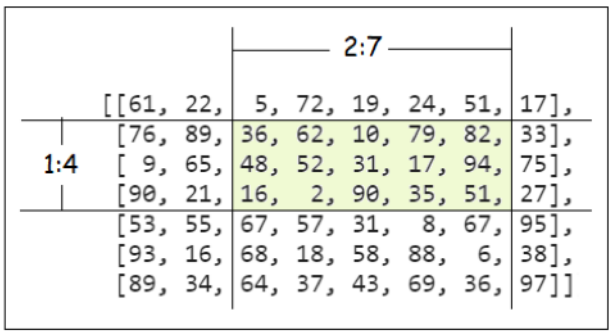
```
(40, 12, 69, 60)
```



Slicing of multidimensional arrays is done using a similar reasoning. Consider the two dimensional array shown below. To get the slice shown on the right side, we use the index ranges across the two axes that define the data slice desired (green region).

```
a = np.random.randint(1, 100, [7, 8])
a
```

```
array([[61, 22,  5, 72, 19, 24, 51, 17],
       [76, 89, 36, 62, 10, 79, 82, 33],
       [ 9, 65, 48, 52, 31, 17, 94, 75],
       [90, 21, 16,  2, 90, 35, 51, 27],
       [53, 55, 67, 57, 31,  8, 67, 95],
       [93, 16, 68, 18, 58, 88,  6, 38],
       [89, 34, 64, 37, 43, 69, 36, 97]])
```

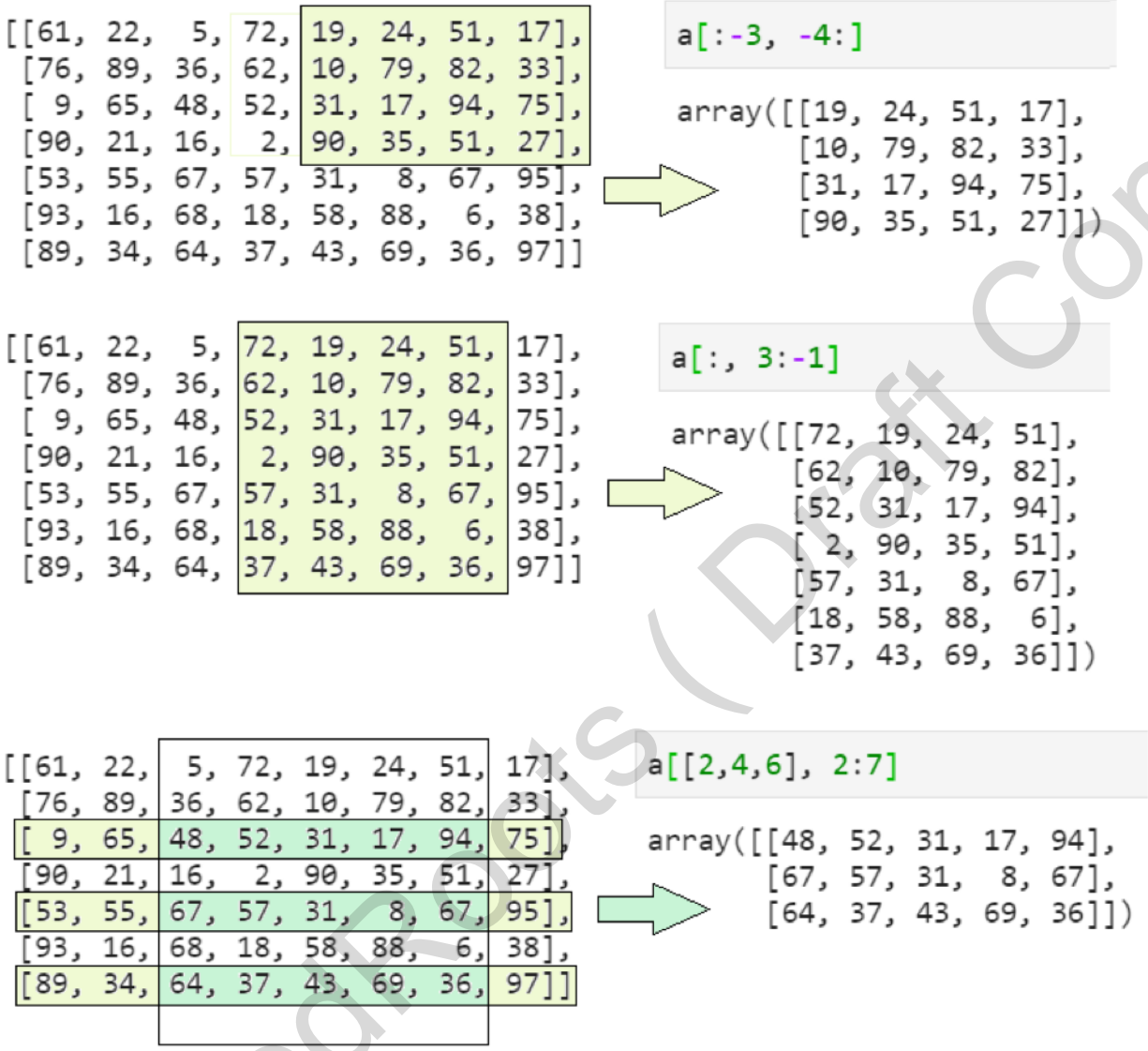


The values in the green region shown in the image above can then be got as shown below:

```
a[1:4, 2:7]
```

```
array([[36, 62, 10, 79, 82],
       [48, 52, 31, 17, 94],
       [16,  2, 90, 35, 51]])
```

Shown below are some more examples of two dimensional array slicing.



ARRAY ATTRIBUTES:

Numpy's array objects have a number of attributes that describe its state, some of the more relevant attributes are described below:

```
a = np.random.rand(3,4)
a
```

```
array([[0.4282672 , 0.62033711, 0.77885041, 0.6024426 ],
       [0.9979824 , 0.83907753, 0.98645608, 0.04216358],
       [0.5664346 , 0.94552892, 0.58755492, 0.00569665]])
```

1. DTYPE:

The **dtype** attribute gives is the data type of the elements stored within the array:

```
a.dtype
```

```
dtype('float64')
```

2. NDIM:

The **ndims** attribute gives is the number of axes/dimensions that the array consists of

```
a.ndim
```

```
2
```

3. SHAPE

The **shape** attribute gives is the shape of the array under consideration:

```
a.shape
```

```
(3, 4)
```

4. SIZE:

The **size** attribute gives us information about the number of elements contained within the array:

```
a.size
```

```
12
```

ARRAY METHODS:

Numpy's array object has a multitude of methods associated with it, some of the more relevant methods are described below:

1. RESHAPE & RESIZE:

The reshape and resize methods perform the same task of reshaping a given array into another configuration. The **reshape** method creates a copy of the original array and performs the reshaping operation on it. Hence the original array remains as is.

The **resize** method changes the shape of the original array into the new preferred shape. The code shown below demonstrates the reshape and resize methods.

```
a = np.random.randint(1, 100, 6)
a
```

```
array([68, 68, 88, 98, 79, 81])
```

```
a1 = a.reshape(2, 3)
a1
```

```
array([[68, 68, 88],
       [98, 79, 81]])
```

```
a.shape, a1.shape
```

```
((6,), (2, 3))
```

```
a.resize(2, 3)
a.shape
```

```
(2, 3)
```

```
a
```

```
array([[68, 68, 88],
       [98, 79, 81]])
```

The **reshape** method also allows negative values to be used to convert the given array into column arrays (array with one columns and n rows) or into row arrays (array with one row and n columns as shown in code below:

```
a.reshape(1, -1)
```

```
array([[68, 68, 88, 98, 79, 81]])
```

```
a.reshape(-1, 1)
```

```
array([[68],
       [68],
       [88],
       [98],
       [79],
       [81]])
```

2. FLATTEN:

The **flatten** method creates a copy of the original array and performs the flattening operation on it. Hence the original array remains as is.

```
a = np.random.randint(1, 100, [2,3])
a
```

```
array([[75, 64,  6],
       [20, 98, 51]])
```

```
a1 = a.flatten()
a1
```

```
array([75, 64,  6, 20, 98, 51])
```

```
a.shape, a1.shape
```

```
((2, 3), (6,))
```

3. SORT & ARGSORT:

The **sort** method modifies the array it is applied to by sorting the values contained within it as shown below:

```
a = np.random.randint(1, 100, 5)
a
```

```
array([35, 88, 39, 86, 83])
```

```
a.sort()
a
```

```
array([35, 39, 83, 86, 88])
```

The **argsort** method returns the indices that would sort the array it was applied to. This is demonstrated in the code shown below:

```
a = np.random.randint(1, 100, 5)
a
```

```
array([61, 21, 77, 43, 57])
```

```
i = a.argsort()
i
```

```
array([1, 3, 4, 0, 2], dtype=int64)
```

One can sort the elements in an array in descending order by using negative stepsizes, while slicing them (see list slicing using negative stepsizes). This is demonstrated in the code shown below:

```
a = np.arange(11, 100, 10)
a
```

```
array([11, 21, 31, 41, 51, 61, 71, 81, 91])
```

```
a.sort()

a1 = a[-1::-1]
a1
```

```
array([91, 81, 71, 61, 51, 41, 31, 21, 11])
```

4. NONZERO:

The **nonzero** method returns the indices of the elements in the array that are non-zero.

```
a = np.array([1, 2, 0, 0 ,3])
a.nonzero()
```

```
(array([0, 1, 4], dtype=int64),)
```

5. MAX & ARGMAX:

The **max** method returns the maximum value along any given axis. This is demonstrated in the code shown below:

```
a = np.random.randint(1, 100, [3, 5])
a
```

```
array([[95, 76, 26, 12, 86],
       [54,  9, 86, 88, 52],
       [27, 54, 11, 88, 30]])
```

```
a.max(), a.max(axis = 0), a.max(axis = 1)
```

```
(95, array([95, 76, 86, 88, 86]), array([95, 88, 88]))
```

The **argmax** method returns the indexes of maximum value along any given axis. This is demonstrated in the code shown below:

```
a.argmax(), a.argmax(axis = 0), a.argmax(axis = 1)
```

```
(0, array([0, 0, 1, 1, 0], dtype=int64), array([0, 3, 3], dtype=int64))
```

6. MIN & ARGMIN:

The **min** and **argmin** methods return the maximum value and the indexes of the maximum value along any given axis. This is demonstrated in the code shown below:

```
a.min(), a.min(axis = 0), a.min(axis = 1)
```

```
(9, array([27,  9, 11, 12, 30]), array([12,  9, 11]))
```

```
a.argmin(), a.argmin(axis = 0), a.argmin(axis = 1)
```

```
(6, array([2, 1, 2, 0, 2], dtype=int64), array([3, 1, 2], dtype=int64))
```

7. SUM & CUMSUN:

The **sum** method returns the sum of the array elements over any given axis. This is demonstrated in the code shown below:

```
a = np.random.randint(1, 100, [3, 5])
a
```

```
array([[16, 80,  5,  8, 94],
       [73, 91, 35, 46, 70],
       [48, 98, 62, 34, 85]])
```

```
a.sum(), a.sum(axis = 0), a.sum(axis = 1)
```

```
(845, array([137, 269, 102,  88, 249]), array([203, 315, 327]))
```

The **cumsum** method returns the cumulative sum of the array elements over any given axis. This is demonstrated in the code shown below:

```
a = np.random.randint(1, 100, [2, 2])
a
```

```
array([[88,  5],
       [62, 32]])
```

```
a.cumsum()
```

```
array([ 88,  93, 155, 187], dtype=int32)
```

```
a.cumsum(axis = 0)
```

```
array([[ 88,   5],
       [150,  37]], dtype=int32)
```

```
a.cumsum(axis = 1)
```

```
array([[88, 93],
       [62, 94]], dtype=int32)
```

8. PROD & CUMPROD:

The **prod** & **cumprod** methods returns the product and cumulative product of the array elements over any given axis. This is demonstrated in the code shown below:

```
a = np.random.randint(1, 100, [2, 3])
a
```

```
array([[ 4, 67, 22],
       [58, 33,  8]])
```

```
a.prod(), a.prod(axis = 0), a.prod(axis = 1)
```

```
(90279552, array([ 232, 2211,  176]), array([ 5896, 15312]))
```

```
a.cumprod()
```

```
array([      4,     268,    5896,  341968, 11284944, 90279552],
      dtype=int32)
```

```
a.cumprod(axis = 0)
```

```
array([[   4,   67,   22],
       [ 232, 2211,  176]], dtype=int32)
```

```
a.cumprod(axis = 1)
```

```
array([[    4,   268,  5896],
       [   58,  1914, 15312]], dtype=int32)
```

9. MEAN, VAR & STD:

The **mean**,, **var** and **std** methods return the mean, median, variance and standard deviation of the array elements along any given axis.

```
a = np.random.randint(1, 100, [2, 3])
a
```

```
array([[19, 76, 51],
       [71, 73, 24]])
```

```
a.mean(), a.mean(axis = 0), a.mean(axis = 1
```

```
(52.333333333333336,
 array([45. , 74.5, 37.5]),
 array([48.66666667, 56.        ]))
```

```
a.var(), a.var(axis = 0), a.var(axis = 1)
```

```
(541.8888888888888,
 array([676.  ,   2.25, 182.25]),
 array([544.22222222, 512.66666667]))
```

```
a.std(), a.std(axis = 0), a.std(axis = 1)
```

```
(23.278507015891048,
 array([26. ,  1.5, 13.5]),
 array([23.32857094, 22.6421436 ]))
```

UNIVERSAL FUNCTIONS OR UFUNCS (INBUILT NUMPY FUNCTIONS):

Consider a **for** loop over a list. At every iteration, when handling the objects within the loop python has to do the following:

1. Use the memory pointer of that specific object within the list to locate the object within the computer's memory.
2. Check the specific object's **type** (ie: which class it belongs to) and determine if the operations that are going to be performed are compatible with the object's type or not.

This can prove to be time consuming and slow. The slowness becomes more apparent as the size of the list increases.

Since arrays are **homogeneous** and **contiguously** stored in memory, numpy does not need to do **type checking** on every object within the array at every iteration nor does it have to jump around in the memory to access these objects. Since the **type** is predetermined and **memory** contiguous, numpy is able to convert looping operations into optimized (concurrent) **compiled** code. Thus the code does not need to be **interpreted** at each loop. This process of converting element wise looping operations into precompiled code is referred to as **vectorization**. The outcome of this can be a tremendous speedup relative to the sequential looping computation performed in Python.

Universal functions (or ufuncs as they are usually called) are inbuilt vectorized numpy functions that cover most of the mathematical operations one would use in computations involving arrays. Ufuncs can be of two types depending on the number of arrays used within the ufunc - **unary** ufuncs and **binary** ufuncs. Unary ufuncs are functions designed to perform computations using the elements present within a single array, whereas binary ufuncs are designed for computations involving two arrays.

UNARY UFUNCS:

Numpy comes with an exhaustive set of unary ufuncs, shown in the table below are some of the more relevant ones:

Unary ufuncs	Description
np.abs	Compute absolute value of each element in array
np.sqrt	Compute square root of each element in array
np.square	Compute square of each element in array
np.exp	Compute exponent e**x of each element in array
np.log	Compute log to the base **e** of each element in array
np.log2	Compute log to the base 2 of each element in array
np.log10	Compute log to the base 10 of each element in array

np.ceil	Compute the smallest integer greater than or equal to a given number for every element in the array
np.floor	Compute the largest integer lesser than or equal to a given number for every element in the array
np.sin, np.sinh	Sin and hyperbolic sin of each element in the array
np.cos, np.cosh	Cos and hyperbolic cos of each element in the array
np.tan, np.tanh	Tan and hyperbolic tan of each element in the array

Most of the ufuncs shown above are quite obvious in their usage. Shown below are examples of some of the more obvious ones:

```
a = np.random.randint(-100, 100, [2, 3])
a
```

```
array([[-74, -58, -57],
       [-94,  43, -82]])
```

```
np.abs(a)
```

```
array([[74, 58, 57],
       [94, 43, 82]])
```

```
np.square(a)
```

```
array([[5476, 3364, 3249],
       [8836, 1849, 6724]], dtype=int32)
```

```
a = np.array([[1.5, 2.22, 11.95]])
a
```

```
array([[ 1.5 ,  2.22, 11.95]])
```

```
np.ceil(a)
```

```
array([[ 2.,  3., 12.]])
```

```
np.floor(a)
```

```
array([[ 1.,  2., 11.]])
```

BINARY UFUNCS:

Binary ufuncs are basically ibuilt vectorized numpy functions that can be applied between two arrays of the same size and object type. The ufuncs is applied across corresponding elements of each array. Some of the relevant binary ufuncs are described in the table shown below.

Binary ufunc	Operator	Description
np.add	" + "	Perform addition between corresponding elements of two arrays
np.subract	" - "	Perform suntraction between corresponding elements of two arrays
np.multiply	" * "	Perform multiplication between corresponding elements of two arrays
np.divide	" / "	Perform division between corresponding elements of two arrays

np.floor_divide	" // "	Perform floor division between corresponding elements of two arrays
np.power	" ** "	Raise elements in first array to powers indicated in corresponding cells of the second array
np.mod	" % "	Perform modulus operation between corresponding elements of two arrays

The code shown below demonstrates some of the binary ufuncs described in the table above:

```
a1 = np.random.randint(-100, 100, [2, 3])
a2 = np.random.randint(-10, 50, [2, 3])
```

```
a1
```

```
array([[ 18, -83,  40],
       [ 71, -34, -93]])
```

```
a2
```

```
array([[11, 34, 14],
       [29, -2, 16]])
```

```
a1 + a2
```

```
array([[ 29, -49,  54],
       [100, -36, -77]])
```

```
a1 / a2
```

```
array([[ 1.63636364, -2.44117647,  2.85714286],
       [ 2.44827586, 17.        , -5.8125    ]])
```

```
a1 % a2
```

```
array([[ 7, 19, 12],
       [13,  0,  3]], dtype=int32)
```

Apart from the arithmetic ufuncs described above, numpy has ufuncs for comparison operators too. Shown below are some examples of comparison ufuncs:

```
a1 >= a2
```

```
array([[ True, False,  True],
       [ True, False, False]])
```

```
a1 == a2
```

```
array([[False, False, False],
       [False, False, False]])
```

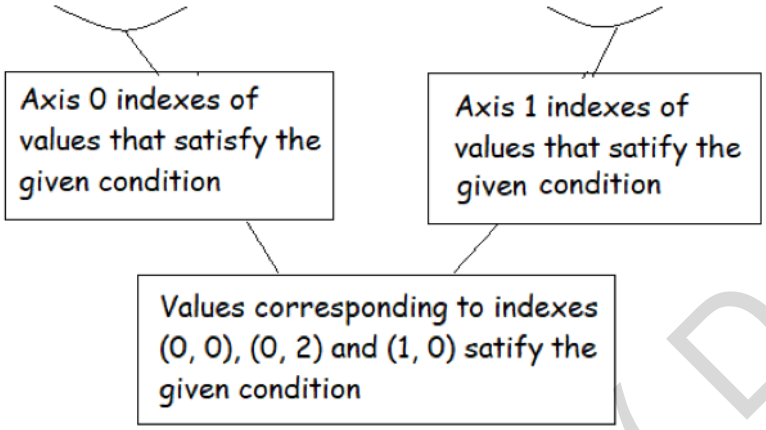
Note how numpy returns an array containing boolean values that indicate which corresponding values of each array satisfy the condition imposed.

THE WHERE UFUNC:

The **where** ufunc applies a **comparison** operation between each corresponding element of two equal sized arrays and returns the indexes of the elements that satisfy that condition. This is demonstrated in the code shown below.

```
np.where(a1 >= a2)
```

```
(array([0, 0, 1], dtype=int64), array([0, 2, 0], dtype=int64))
```



BROADCASTING:

Broadcasting is the technique of modifying two arrays of unequal sizes such that a binary ufunc can be applied between them. Consider the example shown below:

```
a1 = np.array([[11, 69, 2], [6, 21, 73], [1, 3, 0]])
a2 = np.array([3])

a1, a2
```

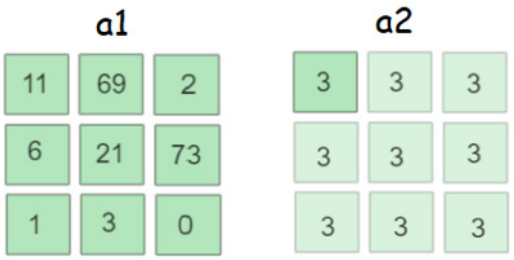
```
(array([[11, 69,  2],
        [ 6, 21, 73],
        [ 1,  3,  0]]),
 array([3]))
```

Consider the operation between the two unequal arrays shown below (**shape(3, 3) and shape(1, 1)**).

```
a1 - a2
```

```
array([[ 8, 66, -1],
       [ 3, 18, 70],
       [-2,  0, -3]])
```

What is actually happening in the above computation is that numpy expands the array **a2** such that it has the same dimensionality of the bigger array. It then fills the newly created cells with the same value contained in the origin array (ie: 3). This value **broadcasting** is demonstrated in the image shown below:



Consider the operation shown below, where an array is multiplied with a single number. Internally numpy considers the number as an array having **shape(1,1)** and then performs the same steps as shown above and delivers the result shown.

```
a1 * 2
```

```
array([[ 22, 138,   4],
       [ 12,  42, 146],
       [  2,   6,   0]])
```

Consider another operation between the two unequal arrays **a1** having **shape(3, 3)** and **a3** having **shape(1, 3)** as shown below.

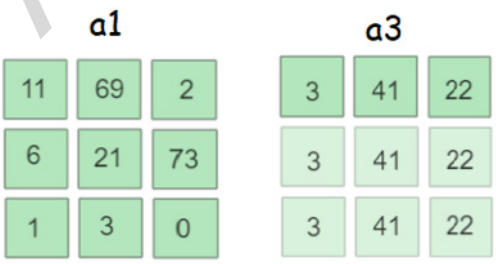
```
a3 = np.array([3, 41, 22])
a3
```

```
array([ 3, 41, 22])
```

```
a4 - a3
```

```
array([[  8, -30, -11],
       [  3, -35, -16],
       [ -2, -40, -21]])
```

In this case, numpy expands the smaller array in the manner shown below and then performs the specified binary ufunc operation (ie: a1 - a3).



Finally consider the example of operation between the two unequal arrays **a1** having **shape(3,1)** and **a3** having **shape(1, 3)** as shown below. Numpy expands both arrays in the manner shown below and then performs the specified binary ufunc operation (ie: a1 - a3).

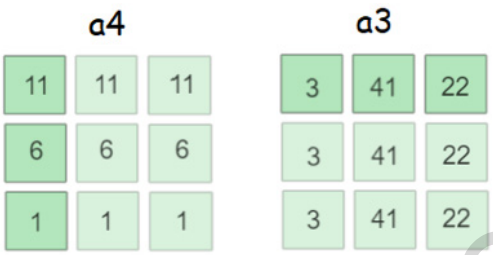
```
a4 = np.array([11, 6, 1]).reshape(-1, 1)
a4
```

```
array([[11],
       [ 6],
       [ 1]])
```

```
a4 - a3
```

```
array([[  8, -30, -11],
       [  3, -35, -16],
       [ -2, -40, -21]])
```

In this case, numpy expands the both arrays in the manner shown below and then performs the specified binary ufunc operation (ie: a4 - a3).



BROADCASTING RULES:

Basically when numpy comes across operations between two arrays of unequal dimensionality it does the following:

1. It increases the **dimensionality** of the smaller array by creating dimensions of unit size for each missing dimension. Thus resulting in two equi-dimensional arrays. This is seen in the first example discussed above.
2. If the size of the dimensions of the two equi-dimensional arrays are not the same, then numpy increases the size of the smaller dimensions on both arrays until both arrays have the same shape. This is seen in the second and third example discussed above.
3. Broadcasting is possible between two arrays of different shapes only if one of the following conditions are satisfied:
   a. One of the arrays is a single element array.
   a. Both arrays have one non similar axis with the same size.

If the above conditions are not met during a binary operation, numpy raises an error.

LINEAR ALGEBRA:

Linear Algebraic operations like dot products, matrix multiplication/transformations, decompositions and other matrix operations can be performed using numpy's **linalg** module. Shown in the table below are some of numpy's most relevant **unary** linear algebraic functions (operations involving only one array).

Function	Description
np.transpose	returns the transpose of a 2D array
np.diag	returns the diagonal elements of a 2D array
np.trace	returns the sum of the diagonal elements
np.linalg.norm	returns the norms of an array along specfied axis
np.linalg.det	returns the determinant of a square array/matrix
np.linalg.eig	returns the eigen values of a square array/matrix
np.linalg.inv	returns the inverse of a square array/matrix
np.linalg.svd	returns the sigular value decompositions of an array/matrix

Shown below is a demonstration of some of the functions described above:

```
a = np.random.rand(3, 3)
a
```

```
array([[0.07959984, 0.62597973, 0.10506718],
       [0.04516355, 0.1813188 , 0.57601288],
       [0.17633413, 0.89324822, 0.66046405]])
```

```
np.diag(a)
```

```
array([0.07959984, 0.1813188 , 0.66046405])
```

```
np.trace(a)
```

```
0.9213826867072751
```

The **transpose** function can also be applied using the **dot T** operation as shown below. This operation converts all rows of a 2D array into columns as shown below.

```
np.transpose(a)
```

```
array([[0.07959984, 0.04516355, 0.17633413],
       [0.62597973, 0.1813188 , 0.89324822],
       [0.10506718, 0.57601288, 0.66046405]])
```

```
a.T
```

```
array([[0.07959984, 0.04516355, 0.17633413],
       [0.62597973, 0.1813188 , 0.89324822],
       [0.10506718, 0.57601288, 0.66046405]])
```

The norm function returns the norm of an array along any axis specified. Norms are a way to quantify the magnitude of a one dimensional array (ie: vector). There are more than one measures for the norm of a vector. Two norm types are relevant for our purposes, they are the L1 and L2 norms. The L1 norm is the sum of the absolute values of each element of the vector, whereas the L2 norm is the square root of the sum of the squares of each element in the vector. The norm desired can be specified using the second argument of the norm function as shown below.

L2 NORMS:

```
np.linalg.norm(a, 2)
```

```
1.353247043281335
```

```
np.linalg.norm(a, 2, axis = 0)
```

```
array([0.19866959, 1.10572127, 0.88263397])
```

```
np.linalg.norm(a, 2, axis = 1)
```

```
array([0.63970765, 0.60556345, 1.1248106 ])
```

L1 NORMS:

```
np.linalg.norm(a, 1)
```

```
1.7005467550755617
```

```
np.linalg.norm(a, 1, axis = 0)
```

```
array([0.30109753, 1.70054676, 1.34154411])
```

```
np.linalg.norm(a, 1, axis = 1)
```

```
array([0.81064675, 0.80249523, 1.73004641])
```

THE DOT PRODUCT:

The dot product is the most fundamental linear algebraic operation between two equal sized vectors (ie: one dimensional array). The dot product can be deconstructed into two sequential operations:

1. Multiplication operation between the two one dimensional arrays.
2. Summation of the values of the vector resulting from step 1.

The two steps described above are implemented in the code shown below:

```
a1 = np.random.rand(3)
a2 = np.random.rand(3)

a1, a2
```

```
(array([0.71180534, 0.10773413, 0.32782862]),
 array([0.1974285 , 0.12348605, 0.01204303]))
```

```
a_dot = (a1 * a2).sum()
a_dot
```

```
0.1577823767189268
```

Dot products can be directly computed using the **dot** function described in the table shown earlier. Also, the **"@"** symbol is utilized in numpy as the dot product operator. This is demonstrated below:

```
np.dot(a1, a2)
```

```
0.1577823767189268
```

```
a1 @ a2
```

```
0.1577823767189268
```

MATRIX MULTIPLICATION:

Matrix multiplication is a linear algebraic operation between two 2D arrays (ie: Matrices), which results in another matrix. Each value at any **index(i, j)** of this resultant matrix is the outcome of the dot product of the ith row of the first matrix and the jth column of the second matrix. Matrix multiplication is only possible if the number of columns of the first matrix is the same as the number of rows of the second matrix.

The code shown below demonstrates matrix multiplication between two compatible matrices.

```
m1 = np.random.randint(1, 100, [2, 3])
m2 = np.random.randint(50, 75, [3, 5])
```

```
m1
```

```
array([[ 9, 17, 65],
       [73, 67, 30]])
```

```
m2
```

```
array([[55, 53, 66, 62, 60],
       [72, 67, 72, 69, 59],
       [55, 56, 74, 63, 59]])
```

```
m_prod = m1 @ m2
m_prod
```

```
array([[ 5294,  5256,  6628,  5826,  5378],
       [10489, 10038, 11862, 11039, 10103]])
```

Each cell above is the dot product of the row of the first matrix corresponding to the first index of the cell and the column of the second matrix corresponding to the second index of the cell. This is demonstrated in the code shown below, which computes the values of cells (0, 0) and (-1, -1), using the actual corresponding vectors from each matrix.

```
m1[0] @ m2[:, 0]
```

```
5294
```

```
m1[-1] @ m2[:, -1]
```

```
10103
```

COSINE SIMILARITY:

The cosine similarity is a measure of how similar two vectors are. The cosine similarity can be defined as the dot product of the two vectors divided by the product of their L2 norms. Shown in the code below, is the cosine similarity similarity computed for two sample vectors. Note that the cosine similarity of a vector with itself is one.

```
def fn_cos_simi(v1, v2):

    dot_prod = v1 @ v2
    norm_v1 = np.linalg.norm(v1, 2)
    norm_v2 = np.linalg.norm(v2, 2)
    cos_simi = dot_prod/(norm_v1 * norm_v2)

    return cos_simi
```

```
v1 = np.random.randint(1, 100, 3)
v2 = np.random.randint(10, 50, 3)

v1, v2
```

```
(array([34, 45, 76]), array([34, 20, 23]))
```

```
cos_simi =  fn_cos_simi(v1, v2)
cos_simi
```

```
0.8802503197580952
```

```
cos_simi =  fn_cos_simi(v1, v1)
cos_simi
```

```
0.9999999999999998
```

```
cos_simi =  fn_cos_simi(v2, v2)
cos_simi
```

```
1.0
```

CONCATENATION OF ARRAYS:

Array can be concatenated using the numpy functions described below:

```
m1 = np.random.randint(1, 100, [2, 3])
v1 = np.random.randint(10, 50, [1, 3])
v2 = np.random.randint(10, 50, [3, 1])
```

```
m1
```

```
array([[77, 51, 32],
       [33, 82, 27]])
```

```
v1
```

```
array([[14, 25, 12]])
```

```
v2
```

```
array([[20],
       [12],
       [27]])
```

1. ROW WISE CONCATENATION:

To perform row concatenation we can use the **concatenate** function with axis specified as **zero** or use the **vstack** (vertical stack) function as shown below.

```
m2 = np.concatenate((m1, v1), axis = 0)
m2
```

```
array([[77, 51, 32],
       [33, 82, 27],
       [14, 25, 12]])
```

```
np.vstack((m1, v1))
```

```
array([[77, 51, 32],
       [33, 82, 27],
       [14, 25, 12]])
```

2. COLUMN WISE CONCATENATION:

To perform column concatenation we can use the **concatenate** function with axis specified as **one** or use the **hstack** (horizontal stack) function as shown below.

```
np.concatenate((m2, v2), axis = 1)
```

```
array([[77, 51, 32, 20],
       [33, 82, 27, 12],
       [14, 25, 12, 27]])
```

```
np.hstack((m2, v2))
```

```
array([[77, 51, 32, 20],
       [33, 82, 27, 12],
       [14, 25, 12, 27]])
```

SAVING AND LOADING NUMPY DATA:

Numpy has two functions **save** and **load** for efficiently saving and loading array data to disk. Arrays are saved in a file with **.npy** extension. The code shown below demonstrates the save and load functions.

```
a = np.random.randint(1, 100, [3, 5])
a
```

```
array([[28, 47, 88, 44, 26],
       [95, 64, 62, 94, 21],
       [18, 73, 82, 99, 10]])
```

```
name = 'array_a.npy'
np.save(name, a)
```

```
a = np.load('array_a.npy')
a
```

```
array([[28, 47, 88, 44, 26],
       [95, 64, 62, 94, 21],
       [18, 73, 82, 99, 10]])
```

This concludes the sixth chapter. In the next chapter we shall explore the **Pandas** Library.

7. THE PANDAS LIBRARY

THE PANDAS LIBRARY

In the chapter we explored the Numpy library which consisted of the **ndarray** object as its main data container object. **Pandas** is a python library built on top of Numpy and it significantly adopts from Numpy's array based computing model. While Numpy is best suited for working with homogeneous array based numerical data, Pandas is designed for working with heterogeneous tabular data. It allows one to perform powerful tabular data based operations similar to those used in Database frameworks and spreadsheet programs.

PANDAS OBJECTS:

There are three fundamental objects in pandas, they are:

1. The Series Object
2. The Dataframe Object
3. The Index object

1. THE SERIES OBJECT:

A Series is a one dimensional data container object containing a sequence of **homogeneous** values. The Series object is similar to a one dimensional array except that the sequence of values contained within a Series object is associated with an **explicitly** defined **index**. This is different from Numpy's array object which has an **implicitly** defined index, which we used to access the values within it. This explicit index definition gives the Series object additional flexibility in terms of the data types used to represent the index values. The Series index can be composed of values of any data types, including **strings**. Thus the values within a Series object can be accessed via the **labels** (generally strings) within its explicitly defined index or via its implicitly defined array integer index. This will become more apparent further along the chapter.

THE SERIES OBJECT AS A GENERALIZED NUMPY ARRAY:

A pandas series can be created from a homogeneous list or an array as shown below:

```
import numpy as np
import pandas as pd
```

```
a = np.random.randint(1, 100, 5)
s = pd.Series(a)

s
```

```
0    35
1     6
2    92
3    19
4    11
dtype: int32
```

As can be seen from the output of the code shown above, the Series object **explicitly** shows an **index** on the left and the **values** on the right. Since an index was not specified during the creation of the above Series object, pandas creates a default index, whose values will be the same as the implicit index values of an array or list containing the same sequence as used above (ie: 0 to n-1 for sequence of length n).

The values and index of a Series object can accessed using its value and index class attributes as shown below:

```
s.values
```

```
array([35,  6, 92, 19, 11])
```

```
s.index
```

```
RangeIndex(start=0, stop=5, step=1)
```

As mentioned before, the index values of a Series object need be just integers. The index of a Series object can be specified using the index keyword argument while creating the Series object. This is demonstrated in the code shown below:

```
i1= ['a', 'b', 'c', 'd', 'e']
s1 = pd.Series(a, index = i1)
s1
```

```
a    35
b     6
c    92
d    19
e    11
dtype: int32
```

The values of a Series object can be accessed via its **explicit** index or via its **implicit** integer based index. This is demonstrated in the code shown below:

```
s1[1 : -2]
```

```
b     6
c    92
dtype: int32
```

```
s1['b' : 'd']
```

```
b     6
c    92
d    19
dtype: int32
```

Note the behaviour of the Series object when sliced using **integer** based indexing and **label** based indexing. The label based slicing shown on the right side includes the last element of the slice specified in its output. The integer based slicing is the same as the normal python indexing/slicing protocol.

THE SERIES OBJECT AS A HOMOGENEOUS DICTIONARY:

A Series object can also be viewed as a dictionary with keys and values constrained to contain only homogeneous data types. In other words all the **keys** have to be of the same **type** and similarly all the **values** also have to be of the same **type**. We can create a Series object using a dictionary that satisfies the constraints required. This is demonstrated in the code shown below:

```
people = ['bob', 'sam', 'john', 'sally', 'kim']
weights = [70, 97, 63, 50, 65]

d = {k:v for k, v in zip(people, weights)}
s = pd.Series(d)
s
```

```
bob      70
sam      97
john     63
sally    50
kim      65
dtype: int64
```

Now, just as in a dictionary object we can access values using keys as shown earlier, but unlike a dictionary, Series supports array based slicing operations as shown below:

```
s['sam':'kim']
```

```
sam      97
john     63
sally    50
kim      65
dtype: int64
```

Most of the **attributes** and **methods** associated with numpy arrays are also applicable to the Series object. This is demonstrated in the code shown below:

```
s.max(), s.min(), s.mean(), s.var(), s.std()
```

```
(97, 50, 69.0, 299.5, 17.30606829987678)
```

```
s.sum(), s.prod()
```

```
(345, 1390252500)
```

```
s.cumsum()
```

```
bob       70
sam      167
john     230
sally    280
kim      345
dtype: int64
```

```
s.cumprod()
```

```
bob              70
sam            6790
john         427770
sally      21388500
kim      1390252500
dtype: int64
```

2. THE DATAFRAME OBJECT:

The DataFrame object basically represents a rectangular table of data, which contains an ordered collection of homogeneous columns. Though the elements within a single column have to be homogeneous, each column can independently have its own data type. The DataFrame objects have two indexes, one to represent the rows within the rectangular table of data and one for the columns. Generally the rows represent multiple observations of some same datagroup (ex: courses offered by Universities) and the columns represent common **features/Characteristics** recorded about each individual row (ex: subject category, employment status after courses, aggregated earnings of students, gender distribution, etc).

DataFrames can be thought of as a collection of Series objects sharing the same index. Consider the dictionary shown below. It expresses the age, weight and height features of a set of individuals.

```
d1 = dict(sam = [60, 52, 5.9],
          jill = [55,  46, 5.4],
          kate = [19, 60, 5.8],
          jack = [36, 89, 6.3])

d1
```

```
{'sam': [60, 52, 5.9],
 'jill': [55, 46, 5.4],
 'kate': [19, 60, 5.8],
 'jack': [36, 89, 6.3]}
```

Each key of the above dict corresponds to a list data container containing a set of features. The same data can be expressed with respect to its features by using the features as keys instead of the individuals. This is demonstrated in the code shown below.

```
d2 = dict(age = [60, 55, 19, 36],
       weight = [52, 46, 60, 89],
       height = [5.9, 5.4, 5.8, 6.3])


d2
```

```
{'age': [60, 55, 19, 36],
 'weight': [52, 46, 60, 89],
 'height': [5.9, 5.4, 5.8, 6.3]}
```

Note that for the dictionary shown above, the values within the container objects that correspond to its keys are heterogeneous with respect to each other (integers, floats and stings). But the values corresponding to each feature are homogeneous by themselves (ex: all weights are integers).

A DataFrame object could be considered as a data container that combines the two perspectives described above into a single two dimensional structure consisting of rows and columns. The rows of this data container represent the perspective expressed by the dictionary **d1** and its columns represent the perspective expressed by the dictionary **d2**. Thus the DataFrame object has two indexes, one pertaining to the individuals (rows) and the other pertaining to the **features** (columns). This is expressed in the image shown below.

	age	weight	height
sam	60	52	5.9
jill	55	46	5.4
kate	19	60	5.8
jack	36	89	6.3

Each row in the DataFrame is a Series object which shares its index with all other rows and each column is also a Series object which shares its index with all other columns. The values within the DataFrame shown above can be expressed and used as a two dimensional numpy array because each of these Series objects can be converted to their corresponding one dimensional array representations via its **"values" attribute**. All numpy array **attributes, functions and methods** are applicable to the DataFrame as a whole and to its individual rows or columns.

DataFrame objects can be created using a dictionary as shown below. Note that Pandas creates a default index that explicitly expresses numpy's implicit integer indexes, when no index is specified during its instantiation. This is demonstrated in the code on the left side.

```
df = pd.DataFrame(d2)
df
```

	age	weight	height
0	60	52	5.9
1	55	46	5.4
2	19	60	5.8
3	36	89	6.3

```
df = pd.DataFrame(d2, index = d1.keys())
df
```

	age	weight	height
sam	60	52	5.9
jill	55	46	5.4
kate	19	60	5.8
jack	36	89	6.3

One can specify the index of a DataFrame object by using the **"index" keyword argument** during its instantiation as shown in the code on the right side above.

DataFrames can also be constructed from two dimensional numpy arrays. The column indexes in this case can be specified using the "columns" keyword argument as shown below.

```
a1 = np.array([[60, 52, 5.9],
               [55, 46, 5.4],
               [19, 60, 5.8],
               [32, 89, 6.3]])
a1
```

```
array([[60. , 52. ,  5.9],
       [55. , 46. ,  5.4],
       [19. , 60. ,  5.8],
       [32. , 89. ,  6.3]])
```

```
cols = ['age', 'weight', 'height']
idx = ['sam', 'jill', 'kate', 'jack']

df = pd.DataFrame(a1, columns = cols, index = idx)
df
```

	age	weight	height
sam	60.0	52.0	5.9
jill	55.0	46.0	5.4
kate	19.0	60.0	5.8
jack	32.0	89.0	6.3

3. THE INDEX OBJECT:

Both Series and DataFrame objects have explicit indexes which one could use to access and slice the data contained within them. These indexes are data objects in their own right. One could consider them as **immutable** arrays or **set-like** data containers on which **set operations** can be performed. One could create an index object as shown below.

```
idx_1 = pd.Index([1, 2, 4, 6, 7, 9])
idx_1
```

```
Int64Index([1, 2, 4, 6, 7, 9], dtype='int64')
```

Since index objects are immutable their content cannot be altered. This is demonstrated by the outcome shown below.

```
idx_1[2] = 77
```

```
---------------------------------------------------
----> 1 idx_1[2] = 77

TypeError: Index does not support mutable operations
```

This immutability of index objects makes it possible to share indexes between multiple DataFrames without the danger of accidentally modifying them (the indexes) while performing operations over these dataframes.t

Set operations can be performed using index objects as shown below. This allows for choosing values across datasets based on set operation performed on their index objects. Set operations on index objects are demonstrated in the code shown below:

```
idx_2 = pd.Index([1, 2, 3, 5, 8, 10])
idx_2
```

```
Int64Index([1, 2, 3, 5, 8, 10], dtype='int64')
```

```
idx_1 | idx_2
```

```
Int64Index([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype='int64')
```

```
idx_1 & idx_2
```

```
Int64Index([1, 2], dtype='int64')
```

```
idx_1 ^ idx_2
```

```
Int64Index([3, 4, 5, 6, 7, 8, 9, 10], dtype='int64')
```

Consider the situation where the index objects **idx_1 and idx_2** in the code above, belong to two DataFrames. Then one could create a third dataframe that is composed of specific rows from both or any of these DataFrames, by using only those rows that correspond to the resultant index obtained from **set** operations on **idx_1 and idx_2.** The same can be done with respect to column indexes of two DataFrames. This concept will be explored further along in this chapter while discussing **join/merge** operations between DataFrames.

READING AND WRITING TABULAR DATA USING PANDAS:

The pandas library contains a number of inbuilt functions for reading various formats of tabular data as a DataFrame object.

Fucntions	Description
pd.read_csv	Load csv (comma separated values) file as DataFrame
pd.read_table	Load tsv (tab separated values) file as DataFrame
pd.read_excel	Load Excel file as DataFrame
pd.read_html	Load table found in html file as DataFrame
pd.read_json	Load tabular data represented as json file as DataFrame
pd,read_hdf5	Load tabular data stored as hdf5 as DataFrame

For most of the purposes in this book we shall be using the **csv** files format. The csv file format is basically a text file containing tabular data as "**comma separated values**" as shown below. Csv files have the '**.csv**' file extension that distinguishes it from other files.

```
,age,weight,height
sam,60.0,52.0,5.9
jill,55.0,46.0,5.4
kate,19.0,60.0,5.8
jack,32.0,89.0,6.3
```

Each row of the table contains **values** separated by commas. The first row of the csv file is usually used to represent the column names of the tabular dataset and it starts with a comma. A basic text file containing values in the format shown above and saved using the **.csv** file extension becomes a csv file.

Consider the DataFrame **df** created earlier. It can be **saved** to disk as a **csv** file using the pandas "**to_csv**" function as shown below.

```
df
```

	age	weight	height
sam	60.0	52.0	5.9
jill	55.0	46.0	5.4
kate	19.0	60.0	5.8
jack	32.0	89.0	6.3

```
name = "sample_table.csv"

df.to_csv(name)
```

Csv files can be read into the current "working space", as a DataFrame object using the **read_csv** function as shown below. The **index_col** keyword argument allows one to specify which column in the csv file represents the index.

```
name = "sample_table.csv"

df = pd.read_csv(name, index_col = 0)
df
```

	age	weight	height
sam	60.0	52.0	5.9
jill	55.0	46.0	5.4
kate	19.0	60.0	5.8
jack	32.0	89.0	6.3

BASIC DATAFRAME ATTRIBUTES & METHODS FOR DATA EXPLORATION:

As discussed before, most of the **attributes** and **methods** defined within Numpy's **array** data container class, are also applicable to Pandas' **Series** and **DataFrame** objects. Apart from these, DataFrames have certain unique attributes and methods associated with them that make them very useful when exploring, analysing and organising heterogeneous column labelled tabular data.

For the sake of demonstrating DataFrame operations, we shall be using the "automobiles dataset", which is available as a **csv** file. This dataset was created in the mid 1980s and it describes various features of the cars available at that time (ex: manufacturer, car type, fuel type, number of cylinders, etc) along with their prices.

We shall explore the functionality of DataFrames, by emulating how one would actually go about exploring the content of a DataFrame and explain relevant attributes, methods and functions of DataFrame objects as and when they are applied during the exploration process.

We first load the data into a DataFrame using the read_csv function and assign it to the variable name **df_cars** as shown below.

```
import numpy as np
import pandas as pd
```

```
df_cars = pd.read_csv('cars.csv')
```

Demonstrated below are some of the most fundamental attributes of the DataFrame.

```
df_cars.shape
```

```
(205, 12)
```

```
df_cars.index
```

```
RangeIndex(start=0, stop=205, step=1)
```

```
df_cars.columns
```

```
Index(['make', 'fuel', 'type', 'drive_wheels', 'eng_loc', 'eng_type', 'n_cyl',
       'hp', 'peak_rpm', 'city_mpg', 'highway_mpg', 'price'],
      dtype='object')
```

THE HEAD, TAIL AND SAMPLE METHODS:

We can "peek" at the DataFrame's content using the **head, tail** and **sample** methods as shown below. The head and tail methods return the first **n** and last **n** rows of the dataset respectively, where **n** is defined by the integer **argument** fed to these methods as shown below. If the number of rows required is not defined, these methods return by default, the first and last five rows respectively.

```
df_cars.head(10)
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	city_mpg	hway_mpg	price
0	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	21	27	5118.0
1	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	21	27	5151.0
2	alfa-romero	gas	hatchback	rwd	front	ohcv	6	154.0	5000.0	19	26	5195.0
3	audi	gas	sedan	fwd	front	ohc	4	102.0	5500.0	24	30	5348.0
4	audi	gas	sedan	4wd	front	ohc	5	115.0	5500.0	18	22	5389.0
5	audi	gas	sedan	fwd	front	ohc	5	110.0	5500.0	19	25	5399.0
6	audi	gas	sedan	fwd	front	ohc	5	110.0	5500.0	19	25	5499.0
7	audi	gas	wagon	fwd	front	ohc	5	110.0	5500.0	19	25	5572.0
8	audi	gas	sedan	fwd	front	ohc	5	140.0	5500.0	17	20	5572.0
9	audi	gas	hatchback	4wd	front	ohc	5	160.0	5500.0	16	22	6095.0

```
df_cars.tail(10)
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	city_mpg	hway_mpg	price
195	volvo	gas	wagon	rwd	front	ohc	4	114.0	5400.0	23	28	36000.0
196	volvo	gas	sedan	rwd	front	ohc	4	114.0	5400.0	24	28	36880.0
197	volvo	gas	wagon	rwd	front	ohc	4	114.0	5400.0	24	28	37028.0
198	volvo	gas	sedan	rwd	front	ohc	4	162.0	5100.0	17	22	40960.0
199	volvo	gas	wagon	rwd	front	ohc	4	162.0	5100.0	17	22	41315.0
200	volvo	gas	sedan	rwd	front	ohc	4	114.0	5400.0	23	28	45400.0
201	volvo	gas	sedan	rwd	front	ohc	4	160.0	5300.0	19	25	NaN
202	volvo	gas	sedan	rwd	front	ohcv	6	134.0	5500.0	18	23	NaN
203	volvo	diesel	sedan	rwd	front	ohc	6	106.0	4800.0	26	27	NaN
204	volvo	gas	sedan	rwd	front	ohc	4	114.0	5400.0	19	25	NaN

The **sample** method returns randomly sampled rows of the data, the quantity of which is defined by the integer **argument** fed to it. This is demonstrated in the code shown below.

```
df_cars.sample(3)
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	city_mpg	hway_mpg	price
91	nissan	gas	sedan	fwd	front	ohc	4	69.0	5200.0	31	37	9895.0
77	mitsubishi	gas	hatchback	fwd	front	ohc	4	68.0	5500.0	31	38	8921.0
187	volkswagen	diesel	sedan	fwd	front	ohc	4	68.0	4500.0	37	42	30760.0

THE INFO METHOD:

The **info** method demonstrated below, gives us information of the data types and the number of **non null** values present in each column.

```
df_cars.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 12 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   make          205 non-null    object
 1   fuel          205 non-null    object
 2   type          205 non-null    object
 3   drive_wheels  205 non-null    object
 4   eng_loc       205 non-null    object
 5   eng_type      205 non-null    object
 6   n_cyl         205 non-null    int64
 7   hp            203 non-null    float64
 8   peak_rpm      203 non-null    float64
 9   city_mpg      205 non-null    int64
 10  highway_mpg   205 non-null    int64
 11  price         201 non-null    float64
dtypes: float64(3), int64(3), object(6)
memory usage: 19.3+ KB
```

As can be seen from the output of the above code, three columns - **hp, peak_rpm** and **price** contain a lesser number of non-null values than the total length of the dataset. This simply means that there are some values missing in these columns. **Missing** or null values are represented by the python's **None** object or numpy's **np.nan** object. We shall discuss missing value representations and operations further on in the chapter under **"Handling missing values"** topic.

THE DESCRIBE METHOD:

The **describe** methods gives us a statistical summary of all the columns of the DataFrame that contain numerical values. The statistics returned by this method include the count. mean, standard deviation, minimum, maximum, 25th, 50th and 75th percentiles values numeric column. This is demonstrated in the code shown below. Note that the statistical summary is returned as a DataFrame object.

```
df_cars.describe()
```

	n_cyl	hp	peak_rpm	city_mpg	highway_mpg	price
count	205.000000	203.000000	203.000000	205.000000	205.000000	201.000000
mean	4.380488	104.256158	5125.369458	25.219512	30.751220	13207.129353
std	1.080854	39.714369	479.334560	6.542142	6.886443	7947.066342
min	2.000000	48.000000	4150.000000	13.000000	16.000000	5118.000000
25%	4.000000	70.000000	4800.000000	19.000000	25.000000	7775.000000
50%	4.000000	95.000000	5200.000000	24.000000	30.000000	10295.000000
75%	4.000000	116.000000	5500.000000	30.000000	34.000000	16500.000000
max	12.000000	288.000000	6600.000000	49.000000	54.000000	45400.000000

DATAFRAME COLUMN ATTRIBUTES:

Pandas converts each **column name** of the tabular data it represents, as **attributes** of the DataFrame **object** and hence column values of a DataFrame can be accessed by using the **dot** operator followed by the column's name. This is demonstrated in the code shown below. Note that this method returns the columns in the form of a Series object instead of an array. The numpy array underlying these Series objects can be accessed using the **values** method if required.

```
df_cars.make
```

```
0      alfa-romero
1      alfa-romero
2      alfa-romero
3             audi
4             audi
          ...
200          volvo
201          volvo
202          volvo
203          volvo
204          volvo
Name: make, Length: 205, dtype: object
```

```
df_cars.price
```

```
0      13495.0
1      16500.0
2      16500.0
3      13950.0
4      17450.0
        ...
200    16845.0
201    19045.0
202    21485.0
203    22470.0
204    22625.0
Name: price, Length: 205, dtype: float64
```

THE UNIQUE METHOD:

The number of unique values within any column of a DataFrame can be obtained by using the "unique" method on its columns. This is demonstrated in the code shown below.

```
df_cars.n_cyl.unique()
```

```
array([ 4,  6,  5,  3, 12,  2,  8], dtype=int64)
```

```
df_cars.make.unique()
```

```
array(['alfa-romero', 'audi', 'bmw', 'chevrolet', 'dodge', 'honda',
       'isuzu', 'jaguar', 'mazda', 'mercedes-benz', 'mercury',
       'mitsubishi', 'nissan', 'peugot', 'plymouth', 'porsche', 'renault',
       'saab', 'subaru', 'toyota', 'volkswagen', 'volvo'], dtype=object)
```

THE VALUE_COUNTS METHOD:

The **distribution** of the values contained in a DataFrame column, can be obtained by applying the **value_counts** method to the DataFrame column. This is demonstrated in the code shown below. Note that it makes sense to use this method only on categorical/non numeric data.

```
df_cars.make.value_counts()
```

```
toyota          32
nissan          18
mazda           17
mitsubishi      13
honda           13
subaru          12
volkswagen      12
peugot          11
volvo           11
dodge            9
mercedes-benz    8
bmw              8
plymouth         7
audi             7
saab             6
porsche          5
isuzu            4
alfa-romero      3
jaguar           3
chevrolet        3
renault          2
mercury          1
Name: make, dtype: int64
```

```
df_cars.n_cyl.value_counts()
```

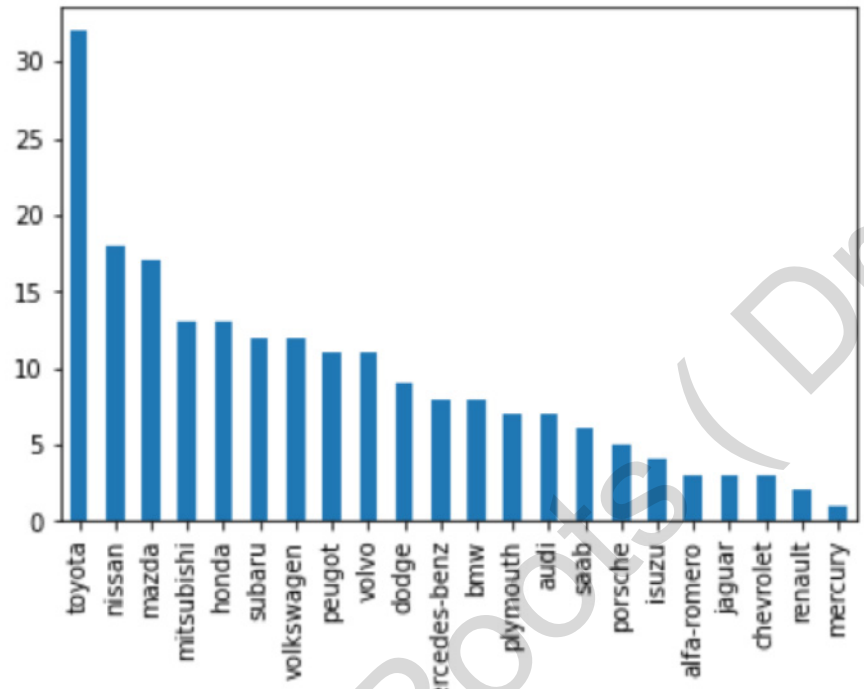
```
4     159
6      24
5      11
8       5
2       4
12      1
3       1
Name: n_cyl, dtype: int64
```

The **value_counts** method returns a Series object, whose **index** represents the **unique values** of the column and whose **values** represent the occurrence counts of the values represented by the index. Note that the Series object is returned sorted in descending order.

THE PLOT METHOD:

The Pandas plot method returns visualizations/plots of the values contained within a Series object or pair of series objects. This helps the user to get a more visual/intuitive understanding of the data contained within DataFrames. The plot method takes in a keyword argument kind, which determines the kind of plot that will be returned.

```
df_cars.make.value_counts().plot(kind = 'bar')
```



Shown above is a **bar** plot, which gives us a clear visual representation of the number of cars belonging to the various automobile manufactures used in the data set. If one prefers a horizontal version of the same bar plot, the **barh** keyword argument should be used instead of **bar**.

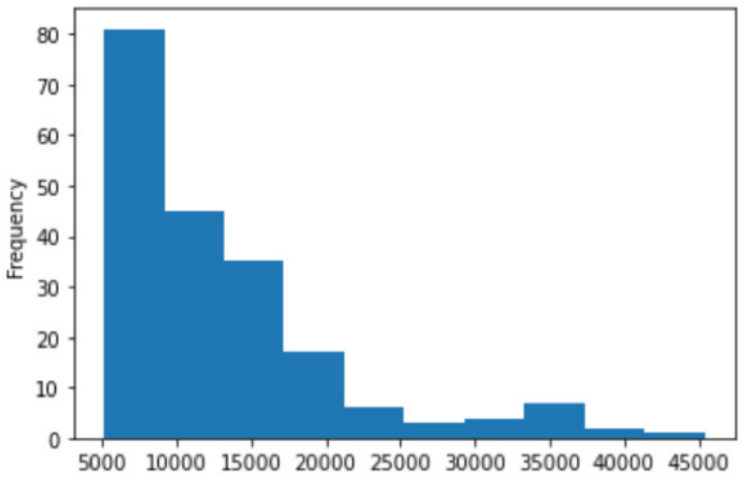
Pandas uses the **Matplotlib** library to create these plots via methods defined within the Series/ DataFrame classes. The Matplotlib library is an extensive python plotting library, which we will be discussing further on in the book under the chapter "The Matplotlib Library".

Some of the relevant plot types available to be used with pandas, apart from the bar pot described above are: Line plots, histograms, Kernel density estimation (kde) plots, and scatter plots.

Shown below are some more examples of plot methods in Pandas.

HISTOGRAM:

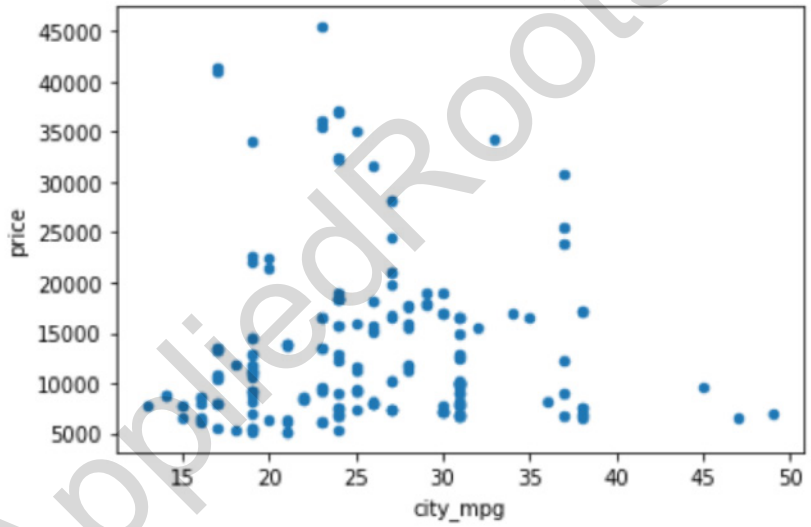
```
df_cars.price.plot(kind = 'hist')
```



From the histogram shown above, we understand that most of the cars lie within the price range of 500 to 15000..

SCATTER PLOT:

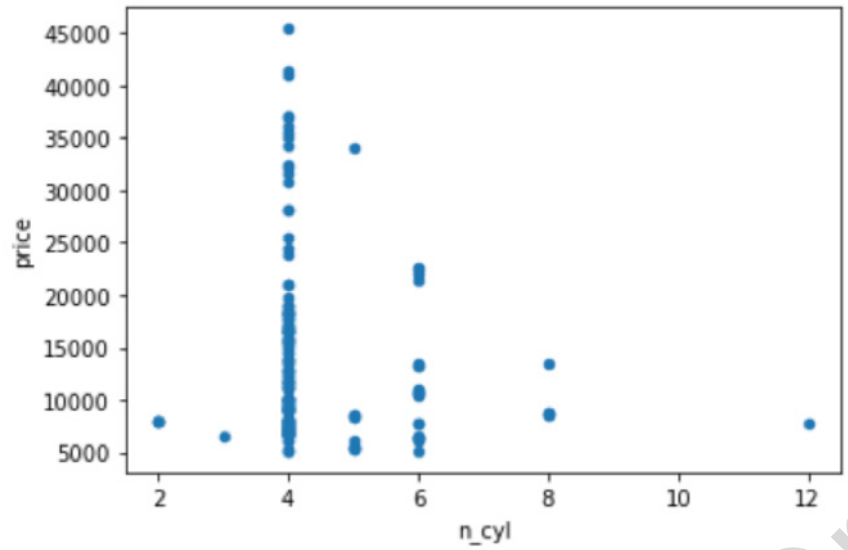
```
df_cars.plot(x = 'city_mpg', y = 'price', kind = 'scatter')
```



Scatter plots are used when one wants to understand the relationship between values in two chosen columns of the DataFrame. In the plot above we see that most of the vehicles have a mileage less than 35 and have their prices below 20000.

Shown below is another scatter plot between number of cylinders and price.

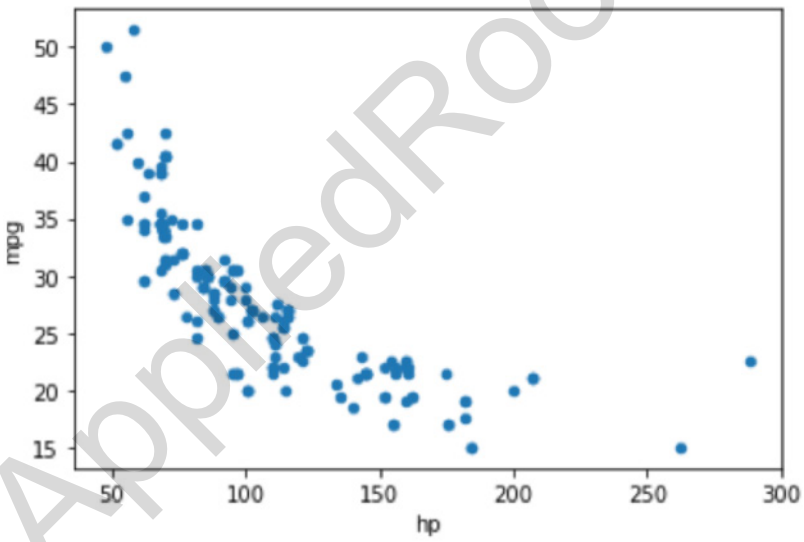
```
df_cars.plot(x = 'n_cyl', y = 'price', kind = 'scatter')
```



From the scatter plot shown above one can say that most of the cars are either 4 cylinder or 6 cylinder and most of the high priced cars are 4 cylinder cars.

From the scatter plot shown below one can say that the mileage of a car generally reduces as its horsepower rating increases.

```
df_cars.plot(x = 'hp', y = 'mpg', kind = 'scatter')
```



THE DROP METHOD:

The Drop method removes rows or columns from a DataFrame by specifying label names and corresponding axis of the rows/columns to be dropped as shown below. Note that we have also used the **head** method to limit the returned output to just the first five rows.

```
df_cars.drop(['drive', 'peak_rpm'], axis = 1).head()
```

	make	fuel	type	eng_loc	eng_type	n_cyl	hp	city_mpg	hway_mpg	price
0	alfa-romero	gas	convertible	front	dohc	4	111.0	21	27	5118.0
1	alfa-romero	gas	convertible	front	dohc	4	111.0	21	27	5151.0
2	alfa-romero	gas	hatchback	front	ohcv	6	154.0	19	26	5195.0
3	audi	gas	sedan	front	ohc	4	102.0	24	30	5348.0
4	audi	gas	sedan	front	ohc	5	115.0	18	22	5389.0

In the code shown above, we drop columns labelled "drive" and "peak_rpm" . Note that we specify the **axis** keyword argument as **one**, to drop **columns**. The default value for **axis** is set to **zero** and hence the drop method by default drops rows. This is demonstrated in the code shown below.

```
df_cars.drop([1 ,2, 5, 7]).head()
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	city_mpg	hway_mpg	price
0	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	21	27	5118.0
3	audi	gas	sedan	fwd	front	ohc	4	102.0	5500.0	24	30	5348.0
4	audi	gas	sedan	4wd	front	ohc	5	115.0	5500.0	18	22	5389.0
6	audi	gas	sedan	fwd	front	ohc	5	110.0	5500.0	19	25	5499.0
8	audi	gas	sedan	fwd	front	ohc	5	140.0	5500.0	17	20	5572.0

The drop method does not actually drop the rows/columns specified, but actually returns a "view" of the DataFrame with desired rows/columns dropped. If one desires to permanently drop columns/rows from a DataFrame, the inplace keyword argument should be changed from its default value of **False** to **True**.

THE SORT_VALUES METHOD:

The **sort_values** method allows one to sort a DataFrame based on the values of any one of its numerical columns. Shown in the code below, we sort the dataset based on the "price" column.

```
df_cars.sort_values('price', ascending = False).head()
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	city_mpg	hway_mpg	price
200	volvo	gas	sedan	rwd	front	ohc	4	114.0	5400.0	23	28	45400.0
199	volvo	gas	wagon	rwd	front	ohc	4	162.0	5100.0	17	22	41315.0
198	volvo	gas	sedan	rwd	front	ohc	4	162.0	5100.0	17	22	40960.0
197	volvo	gas	wagon	rwd	front	ohc	4	114.0	5400.0	24	28	37028.0
196	volvo	gas	sedan	rwd	front	ohc	4	114.0	5400.0	24	28	36880.0

The **False** boolean value assigned to the **ascending** keyword argument as shown above, sorts the dataframe in descending order. This keyword argument by default is assigned the **True** boolean value and thus the **sort_values** method sorts the DataFrame in ascending order of the column specified.

THE ASSIGN METHOD:

The assign method is used to create a new DataFrame containing the same values as the original DataFrame to which it was applied to, but with new columns assigned to it. This is demonstrated in the code below. Note that the new column is added as last column in the new DataFrame.

```
df_cars2 = df_cars.assign(mpg = avg_mpg)
df_cars2.head()
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	city_mpg	hway_mpg	price	mpg
0	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	21	27	5118.0	24.0
1	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	21	27	5151.0	24.0
2	alfa-romero	gas	hatchback	rwd	front	ohcv	6	154.0	5000.0	19	26	5195.0	22.5
3	audi	gas	sedan	fwd	front	ohc	4	102.0	5500.0	24	30	5348.0	27.0
4	audi	gas	sedan	4wd	front	ohc	5	115.0	5500.0	18	22	5389.0	20.0

New columns can be directly inserted into the original DataFrame by assigning them to new column **labels** as shown below. This is similar to assigning a new **key value** pair to a dictionary. Here too, the new column is added as the last column of the DataFrame.

```
df_cars['mpg'] = avg_mpg
df_cars.head()
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	city_mpg	hway_mpg	price	mpg
0	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	21	27	5118.0	24.0
1	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	21	27	5151.0	24.0
2	alfa-romero	gas	hatchback	rwd	front	ohcv	6	154.0	5000.0	19	26	5195.0	22.5
3	audi	gas	sedan	fwd	front	ohc	4	102.0	5500.0	24	30	5348.0	27.0
4	audi	gas	sedan	4wd	front	ohc	5	115.0	5500.0	18	22	5389.0	20.0

THE INSERT METHOD:

The **insert** method allows one to insert a new column into a DataFrame at a specific index location. Let's say that we want to drop the **city_mpg** and **hway_mpg** columns from **df_cars** and keep only the **mpg** column. Also, we want to reposition the **mpg** column as the second last column. We do this as shown in the code below.

```
df_cars.drop(['city_mpg', 'hway_mpg', 'mpg'], axis = 1, inplace = True)
loc = df_cars.shape[1] - 1

df_cars.insert(loc = loc, column = 'mpg', value = avg_mpg)
df_cars.head()
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	mpg	price
0	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	24.0	5118.0
1	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	24.0	5151.0
2	alfa-romero	gas	hatchback	rwd	front	ohcv	6	154.0	5000.0	22.5	5195.0
3	audi	gas	sedan	fwd	front	ohc	4	102.0	5500.0	27.0	5348.0
4	audi	gas	sedan	4wd	front	ohc	5	115.0	5500.0	20.0	5389.0

As can be seen from above, the insert method has three main keyword arguments (or **kwargs** for short). The **loc** kwarg lets one specify the location where one wants to insert the new column. The **column** kwarg lets one specify the column name and the **value** kwarg specifies the new column values (Series/array/list) that need to be added.

THE GROUPBY METHOD:

The **Groupby** method is used for aggregating the **numerical** columns of a DataFrame with respect to some other column containing **categorical** values. Consider the Data Frame **df_cars**, the first six of its columns contain categorical values and the rest are numerical in nature. Suppose one wants to do a **mean aggregate** on the numerical values of the DataFrame with respect to the "**drive**" column, this can be done as shown below.

```
df_drive = df_cars.groupby('drive').mean()
df_drive
```

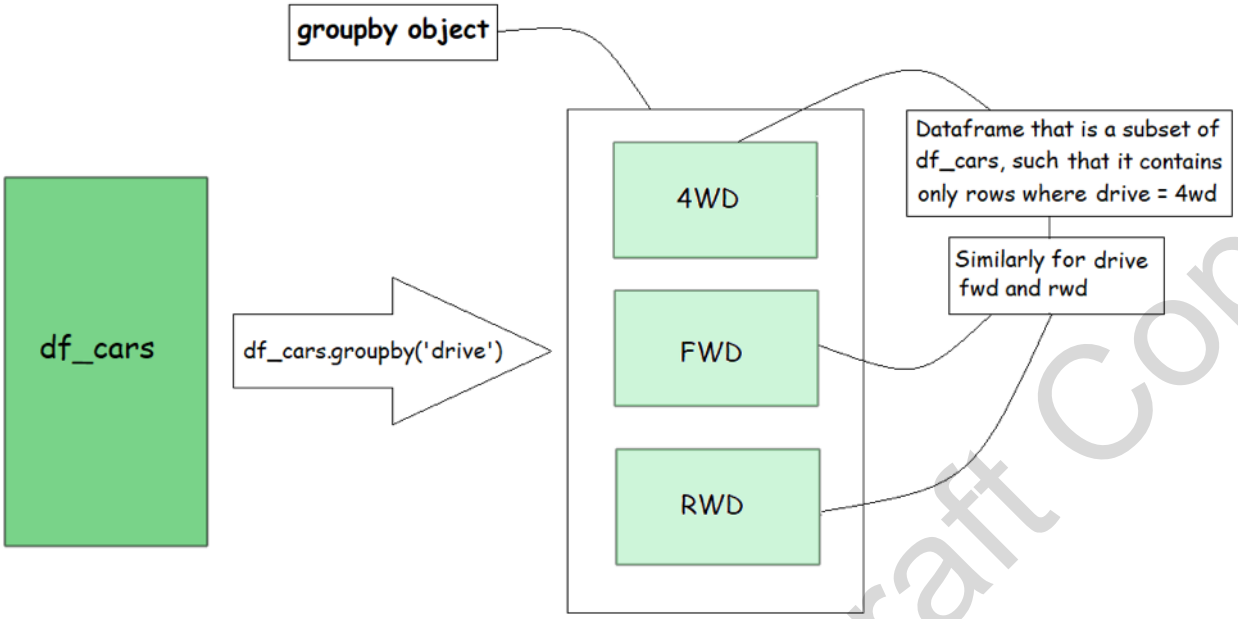
drive	n_cyl	hp	peak_rpm	mpg	price
4wd	4.222222	95.333333	4911.111111	25.166667	13825.444444
fwd	4.083333	85.974576	5176.271186	31.270833	12642.933333
rwd	4.868421	133.697368	5071.710526	23.131579	14070.166667

As can be seen from the output shown above, a DataFrame is returned, whose index is composed of the **unique** values of the column specified in the method's **argument** (ie: **drive**). Every row corresponding to these unique categorical values represented in the index, contains the statistical aggregate specified by the **method** applied after the groupby method (ie: mean).

Thus from the DataFrame **df_drive** returned above, we know the mean values for each of the numerical columns with respect to the number of cylinders of a vehicle. Thus the average **number of cylinders, hp, peak rpm, miles per gallon** and **price** for rear wheel drive cars are 4.8, 133.69, 5071.71, 23.13 and 14070.17 respectively.

THE GROUPBY OBJECT:

The code shown earlier contains two methods applied in succession: the **groupy** method and the **mean** method. The groupby method basically creates a **groupby object**. The groupby object is a special pandas object that internally contains multiple DataFrames as shown in the image below. Each DataFrames contained within the groupby object, consists of only those rows that correspond to one of the categories contained within the column chosen, in this case **drive**(ie: df_cars.groupby(**drive**)).



The groupby object does not display any values. This is demonstrated in the code shown below. It relies on a **second aggregation method** like count, sum, mean, std, var, min or max to be applied after it to return any result.

```
drive_grpby = df_cars.groupby('drive')

drive_grpby
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001F645BD0610>
```

When any of the aggregation methods mentioned above is applied to a groupby object, pandas internally applies that method individually to each of the DataFrames contained within it, thus resulting in one row per DataFrame that contains the aggregates of each column. Each of these resulting rows is then collected within another DataFrame and then returned. This is demonstrated by the code shown below.

```
df_drive_grpby = drive_grpby.mean()
df_drive_grpby
```

drive	n_cyl	hp	peak_rpm	mpg	price
4wd	4.222222	95.333333	4911.111111	25.166667	13825.444444
fwd	4.083333	85.974576	5176.271186	31.270833	12642.933333
rwd	4.868421	133.697368	5071.710526	23.131579	14070.166667

Shown below is another example of the groupby operation, this time with reference to the 'type' column.

```
df_drive = df_cars.groupby('type').mean()
df_drive
```

type	n_cyl	hp	peak_rpm	mpg	price
convertible	5.000000	131.666667	5158.333333	23.250000	13902.000000
hardtop	5.125000	142.250000	5031.250000	24.437500	13634.625000
hatchback	4.114286	101.333333	5232.608696	29.242857	11516.828571
sedan	4.510417	103.104167	5081.770833	28.078125	13459.793478
wagon	4.240000	97.750000	5014.583333	26.380000	16706.600000

Note that the indexes of the DataFrames returned by a groupby operation have **names**. The name of such indexes can be accessed using the name attribute of index objects. This demonstrated as shown below.

```
df_drive.index.name
```

```
'type'
```

Dataframes with no index names (actually index names are by default assigned the **None** value) can be named and DataFrames with already existing index names can be renamed, by directly assigning the desired name to the name attribute of the index object. This is demonstrated in the code below.

```
df_drive.index.name = 'shape'

df_drive.head()
```

shape	n_cyl	hp	peak_rpm	mpg	price
convertible	5.000000	131.666667	5158.333333	23.250000	13902.000000
hardtop	5.125000	142.250000	5031.250000	24.437500	13634.625000
hatchback	4.114286	101.333333	5232.608696	29.242857	11516.828571
sedan	4.510417	103.104167	5081.770833	28.078125	13459.793478
wagon	4.240000	97.750000	5014.583333	26.380000	16706.600000

HANDLING MISSING DATA (None, nan & NaN):

Real world data is rarely clean and organized, and it is quite common for them to have values missing. The ability to be able to work with missing values is therefore quite important. Pandas makes use of special representations and data handling protocols that makes working with missing data as easy as possible.

In statistical parlance, missing values are referred to as **NA** values (not available). Pandas relies on Numpy's Internal representation of NA values while handling missing data. **Numpy** represents NA values using an abstract float value called nan which is basically an acronym for "Not A Number". This demonstrated in the code below.

```
n = np.nan

n, type(n)
```

```
(nan, float)
```

Any arithmetic numpy operation involving a nan results in another nan. This is demonstrated below.

```
n + 1000000, n*66
```

```
(nan, nan)
```

Thus any form of aggregating method applied to an array containing n nan value will result in a nan. This is demonstrated in the code shown below.

```
a = np.array([11, 2, np.nan, 5])
a.sum()
```

```
nan
```

Python natively uses the None value to represent missing values. Numpy arrays can contain None values, but cannot perform computations using them in the way described above for nan values. This is demonstrated in the code below.

```
a1 = np.array([11, 2, None, 5])
a1.sum()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-220-68a11aaf8f27> in <module>
      1 a1 = np.array([11, 2, None, 5])
----> 2 a1.sum()

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Pandas relies on Numpy's **nan** value to handle and perform operations on missing data. Pandas also accepts python's **None** values, but it internally converts them to numpy's **nan** values. Pandas represents **nan** as **NaN** within its container objects. This is demonstrated in the code shown below.

```
l = [11, 2, None, np.nan, 5, 722]
s = pd.Series(l)
s
```

```
0     11.0
1      2.0
2      NaN
3      NaN
4      5.0
5    722.0
dtype: float64
```

```
s[2], s[3]
```

```
(nan, nan)
```

```
type(s[2]), type(s[3])
```

```
(numpy.float64, numpy.float64)
```

np.nan values are abstract float values

One of the protocols that pandas uses while dealing with NaN values is that it excludes them during any aggregation operation. This makes it easier to statistically inspect data even in the presence of missing values. This is demonstrated in the code shown below.

```
s.mean(), s.std()
```

```
(185.0, 358.0195525386847)
```

```
s.cumsum()
```

```
0     11.0
1     13.0
2      NaN
3      NaN
4     18.0
5    740.0
dtype: float64
```

```
s.cumprod()
```

```
0       11.0
1       22.0
2        NaN
3        NaN
4      110.0
5    79420.0
dtype: float64
```

It was because of this protocol implemented within Pandas, that we were able to explore and perform statistical aggregations on the DataFrame df_cars, which contained missing values.

ISNA, DROPNA & FILLNA METHODS:

Isna, **notna**, **dropna** and **fillna** are Pandas methods pandas used for handling missing values. The **isna** and **notna** methods are used for detecting missing values as shown below.

```
s
```

```
0     11.0
1      2.0
2      NaN
3      NaN
4      5.0
5    722.0
dtype: float64
```

```
s.isna()
```

```
0    False
1    False
2     True
3     True
4    False
5    False
dtype: bool
```

```
s.notna()
```

```
0     True
1     True
2    False
3    False
4     True
5     True
dtype: bool
```

Indexes corresponding to missing value can be got by using Numpy's where function as shown below.

```
null_s = s.isna()

np.where(null_s)
```

```
(array([2, 3], dtype=int64),)
```

The **isna** method when applied to DataFrames returns another DataFrame with the same index and size, but filled with boolean values instead. All **True** values within this returned DataFrame represent cells in the original Dataframe where values are missing. This is demonstrated in the code shown below.

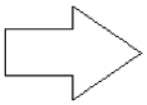
```
df_cars.isna().sample(5)
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	mpg	price
134	False	False	False	False	False	False	False	False	False	False	False
34	False	False	False	False	False	False	False	False	False	False	False
196	False	False	False	False	False	False	False	False	False	False	False
179	False	False	False	False	False	False	False	False	False	False	False
125	False	False	False	False	False	False	False	False	False	False	False

To find out which columns contain missing values we can apply the sum method across the axis = 0 as shown below (this works because True == 1 and False == 0).

```
null_cols = df_cars.isna().sum()
null_cols
```

```
make         0
fuel         0
type         0
drive        0
eng_loc      0
eng_type     0
n_cyl        0
hp           2
peak_rpm     2
mpg          0
price        4
dtype: int64
```



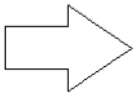
```
np.where(null_cols)
```

```
(array([ 7,  8, 10], dtype=int64),)
```

To find out which rows contain missing values we can apply the sum method across the axis = 1 as shown below.

```
null_rows = df_cars.isna().sum(axis = 1)
null_rows
```

```
0      0
1      0
2      0
3      0
4      0
      ..
200    0
201    1
202    1
203    1
204    1
Length: 205, dtype: int64
```



```
np.where(null_rows)
```

```
(array([130, 131, 201, 202, 203, 204], dtype=int64),)
```

The **dropna** method is used for dropping rows or columns in a DataFrame that contain missing values. This is demonstrated in the code shown below. The code on the right drops rows containing missing values and the code on the right drops columns.

```
df_cars.dropna().shape
```

```
(199, 11)
```

```
df_cars.dropna(axis = 1).shape
```

```
(205, 8)
```

Note that the dropna method does not actually change the original DataFrame but actually returns a "view" of the original DataFrame, with the desired rows/columns dropped. To actually drop the columns/rows from the original DataFrame one has to change the **inplace** keyword argument to True.

The **fillna** method is used to fill missing values in Series and DataFrames with values of our choice. This is demonstrated for Series objects in the code shown below.

```
l = [3.5, 4, 3.3, np.nan, np.nan, 2.75]
s = pd.Series(l)
s
```

```
0    3.50
1    4.00
2    3.30
3     NaN
4     NaN
5    2.75
dtype: float64
```

```
s.fillna(0)
```

```
0    3.50
1    4.00
2    3.30
3    0.00
4    0.00
5    2.75
dtype: float64
```

```
s.fillna(s.mean())
```

```
0    3.5000
1    4.0000
2    3.3000
3    3.3875
4    3.3875
5    2.7500
dtype: float64
```

In the case of DataFrames, the **fillna** method can be used as shown below.

```
df
```

	a	b	c
0	1.0	2	NaN
1	22.0	5	77.0
2	NaN	9	10.0

```
df.fillna(0)
```

	a	b	c
0	1.0	2	0.0
1	22.0	5	77.0
2	0.0	9	10.0

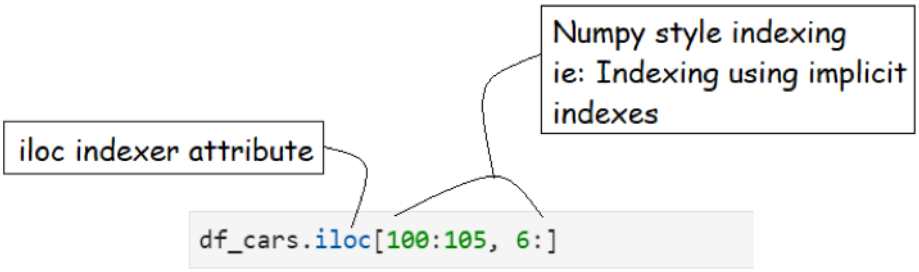
In the case that one wants to specify separate "**fill**" values for each column, it can be done using a dictionary which specifies the fill values for each of the columns containing missing values. This is demonstrated in the code shown below.

```
c_mean = df.c.mean()
df.fillna({'a': 15, 'c': c_mean})
```

	a	b	c
0	1.0	2	43.5
1	22.0	5	77.0
2	15.0	9	10.0

DATA SELECTION AND SLICING:

Pandas main provides the **iloc** and **loc indexer attributes** for data selection and slicing. The **iloc** indexer attribute allows one to use the underlying implicit numpy indexes for data selection/slicing. This is demonstrated in the code shown below.




```
df_cars.iloc[100:105, 6:]
```

	n_cyl	hp	peak_rpm	mpg	price
100	4	97.0	5200.0	30.5	10295.0
101	6	152.0	5200.0	19.5	10345.0
102	6	152.0	5200.0	19.5	10595.0
103	6	152.0	5200.0	22.0	10698.0
104	6	160.0	5200.0	22.0	10795.0

The loc indexer attribute allows one to use the DataFrame's explicitly labeled indexes for data selection/slicing. This is demonstrated in the code shown below.


```
df_cars.loc[100:105, 'n_cyl':]
```

	n_cyl	hp	peak_rpm	mpg	price
100	4	97.0	5200.0	30.5	10295.0
101	6	152.0	5200.0	19.5	10345.0
102	6	152.0	5200.0	19.5	10595.0
103	6	152.0	5200.0	22.0	10698.0
104	6	160.0	5200.0	22.0	10795.0
105	6	200.0	5200.0	20.0	10898.0

CONDITIONAL SLICING:

DataFrames can be conditionally sliced based on values contained in its columns as shown below.

```
cond = df_cars.make == 'alfa-romero'
df_cars[cond]
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	mpg	price
0	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	24.0	5118.0
1	alfa-romero	gas	convertible	rwd	front	dohc	4	111.0	5000.0	24.0	5151.0
2	alfa-romero	gas	hatchback	rwd	front	ohcv	6	154.0	5000.0	22.5	5195.0

```
cond1 = df_cars.price <= 6650
cond2 = df_cars.mpg >= df_cars.mpg.mean()

df_cars[cond1 & cond2]
```

	make	fuel	type	drive	eng_loc	eng_type	n_cyl	hp	peak_rpm	mpg	price
18	chevrolet	gas	hatchback	fwd	front	l	3	48.0	5100.0	50.0	6529.0
19	chevrolet	gas	hatchback	fwd	front	ohc	4	70.0	5400.0	40.5	6575.0
20	chevrolet	gas	sedan	fwd	front	ohc	4	70.0	5400.0	40.5	6649.0

COMBINING DATAFRAMES:

Sometimes it is required to unify/combine two DataFrames containing different information about some common set of objects, so as to better understand the data. The process of merging and joining of tabular data, is an intricate subject by itself, requiring one to be fluent with the rules of r**elational algebra.**

For the purposes of our discussion we limit ourselves to the two basic kinds of joins as described below.

INNER JOINS:

Inner joins basically lets one combine rows that share data. It basically requires that the two DataFrames share a column that describe the same feature about the data. This common column (generally the index) is referred to as 'key'. Given a key, an inner join will only consider those rows that are common to both the DataFrames (intersection of the keys values of both datasets) and combine them together in one DataFrame.

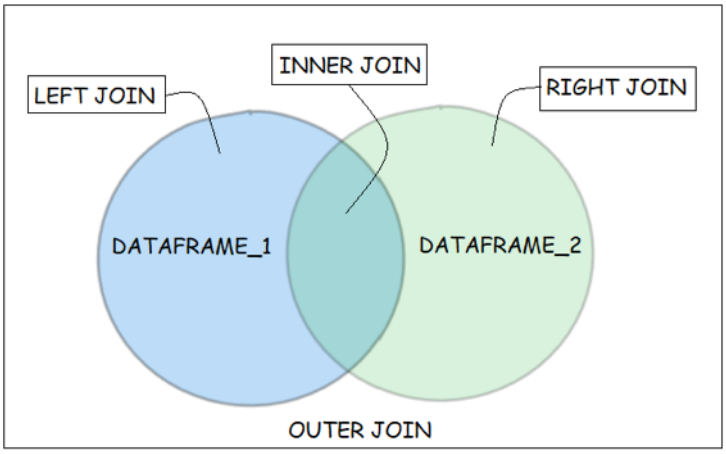
OUTER JOINS:

Outer joins basically lets one combine rows of two DataFrames based on:

1. All the rows in any one of the DataFrames (left and right outer joins)
2. The union of the rows of both DataFrames (Outer join)

It does not require that the two DataFrames have common rows. Outer joins combine DataFrames based on the specifications defined in points 1 and 2 described above. It fills cells of rows that are not common to both with NaN values.

The set diagram shown below expresses the concept of joins further.



We shall describe join operation in Pandas using the two simplistic DataFrames shown below.

df1

	F1	F2
A	0	11
B	1	22
C	2	33
D	3	44
E	4	55
F	5	66

df2

	F3	F4
A	0	111
B	1	222
Y	2	333
Z	3	444

Inner joins and outer joins can be applied across these two DataFrames using the merge, join and concat functions as shown below. Note that all of these join operations are performed with respect to the index of the DataFrames. One can also join using some common column as the join 'key', by using the keyword arguments of these functions to specify it.

INNER JOIN

```
df3 = pd.merge(df1, df2,
               left_index=True,
               right_index=True)
df3
```

	F1	F2	F3	F4
A	0	11	0	111
B	1	22	1	222

LEFT JOIN

```
df4 = df1.join(df2)
df4
```

	F1	F2	F3	F4
A	0	11	0.0	111.0
B	1	22	1.0	222.0
C	2	33	NaN	NaN
D	3	44	NaN	NaN
E	4	55	NaN	NaN
F	5	66	NaN	NaN

OUTER JOIN

```
df5 = pd.concat([df1, df2],
                axis=1)
df5
```

	F1	F2	F3	F4
A	0.0	11.0	0.0	111.0
B	1.0	22.0	1.0	222.0
C	2.0	33.0	NaN	NaN
D	3.0	44.0	NaN	NaN
E	4.0	55.0	NaN	NaN
F	5.0	66.0	NaN	NaN
Y	NaN	NaN	2.0	333.0
Z	NaN	NaN	3.0	444.0

The concat function is also very convenient to use in situations where one needs to combine multiple DataFrames which share the same columns. This is shown in the code below.

```
df1 = pd.DataFrame({'A':range(5),
                    'B':[11,22,33,44,55]})
df1
```

	A	B
0	0	11
1	1	22
2	2	33
3	3	44
4	4	55

```
df2 = pd.DataFrame({'A':range(2),
                    'B':[111,223]})
df2
```

	A	B
0	0	111
1	1	223

```
df3 = pd.DataFrame({'A':[5],
                    'B':[1111]})
df3
```

	A	B
0	5	1111

```
df4 = pd.concat([df1, df2, df3])
df4
```

	A	B
0	0	11
1	1	22
2	2	33
3	3	44
4	4	55
0	0	111
1	1	223
0	5	1111

```
df4 = pd.concat([df1, df2, df3],
                ignore_index = True)
df4
```

	A	B
0	0	11
1	1	22
2	2	33
3	3	44
4	4	55
5	0	111
6	1	223
7	5	1111

The **ignore_index** keyword argument creates a new index for the resulting DataFrame if set to **True**, otherwise by default, the original index values are preserved.

UFUNCS

As discussed earlier Pandas inherits much of Numpy's unary and binary ufunc functionality and hence all the performant element wise operations that are possible using numpy arrays can also be performed using pandas.

Incase of binary ufunc operations, the operations are carried as per the labelled column and row indexes of the DataFrames. In other words elementwise operations are performed across cells that share the same indexes and for indexes that are not common to both DataFrames, the operation results in a NaN value. This is demonstrated in the code shown below.

```
df1
```

	A	B
0	10	11
1	12	22
2	12	33
3	13	44
4	15	55

```
df2
```

	A	B	C
0	0	1	111
1	1	2	222
2	2	3	333
3	3	4	444
5	4	5	555

```
df1 + df2
```

	A	B	C
0	10.0	12.0	NaN
1	13.0	24.0	NaN
2	14.0	36.0	NaN
3	16.0	48.0	NaN
4	NaN	NaN	NaN
5	NaN	NaN	NaN

APPLY FN

Apart from using the inbuilt functions on a DataFrame, pandas also allows one to define custom functions and apply them across the DataFrame using the apply function. This is demonstrated in the code shown below.

```
df1
```

	A	B
0	10	11
1	12	22
2	12	33
3	13	44
4	15	55

```
def example_fn(x):return (x**2)/2

df1 = df1.apply(example_fn)
df1
```

	A	B
0	50.0	60.5
1	72.0	242.0
2	72.0	544.5
3	84.5	968.0
4	112.5	1512.5

FEATURE SELECTION (CORRELATION):

Most often in Statistics and Data Science, we deal with datasets that contain '**features**' and '**labels**' . The labels refer to values that one must be able to predict given a set of **features**. Consider the example of the dataset shown below.

```
df = pd.read_csv('df_boston_housing.csv')
df.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	5.33	36.2

FEATURES

LABELS/TARGET VARIABLE (VALUE TO BE PREDICTED)

The dataset contains 506 entries consisting of 12 **features**, which were collected by the U.S Census Service concerning housing in the area of Boston in 1978. The target variable/labels in this dataset are the median values of owner-occupied homes in units of 1000 dollars. Given below is a description of the features and labels:

```
- CRIM     per capita crime rate by town
- ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS    proportion of non-retail business acres per town
- CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX      nitric oxides concentration (parts per 10 million)
- RM       average number of rooms per dwelling
- AGE      proportion of owner-occupied units built prior to 1940
- DIS      weighted distances to five Boston employment centres
- RAD      index of accessibility to radial highways
- TAX      full-value property-tax rate per $10,000
- PTRATIO  pupil-teacher ratio by town
- LSTAT    % lower status of the population
- MEDV     Median value of owner-occupied homes in $1000's
```

Each '**feature**' column in the data shown above, represents some characteristic about the subject (in this case houses in Boston) being analysed and each row represents a **sample** (house) described in terms of those characteristics. The '**labels**' column represents some value of importance that corresponds to the features of each of these samples (Median Value of house or MEDV).

One of the main tasks in statistics and machine learning is to build predictive models that can predict the labels to a close approximation given a set of feature values. For instance, given a house that has similar features as that represented in the first row in the data shown above, the model should be able to predict a value that is close to the value represented by its label. In other words, we analyse data containing **features** and **labels** and then formulate models that mathematically represent the **correlations** detected between the features and the labels. This model is then used to **predict** the labels for **new** data of the same kind.

Sometimes the number of features contained in the dataset can be too large to be feasible for analysis. Imagine a dataset containing hundreds of features, in such cases we would like to reduce the number of features based on some criteria, such that we end up with a dataset with a smaller, more manageable set of features, that are more important than the rest. This is called **feature selection.**

For the sake of convenience we will use the dataset shown above to demonstrate how feature selection is done using correlation. We rename the columns of the DataFrame using more generic names as shown below:

```
df.columns = ['f' + str(i) for i in range(df.shape[1]-1)]+['labels']

df.head()
```

	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	labels
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	5.33	36.2

PANDAS CORRELATION METHOD:

The inbuilt **correlation** method for DataFrames, computes the correlation of every column in the DataFrame with every other column and presents the results in the form of a correlation matrix as shown below.

```
df_corr = df.corr().abs()
df_corr
```

	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	labels
f0	1.000000	0.200469	0.406583	0.055892	0.420972	0.219247	0.352734	0.379670	0.625505	0.582764	0.289946	0.455621	0.388305
f1	0.200469	1.000000	0.533828	0.042697	0.516604	0.311991	0.569537	0.664408	0.311948	0.314563	0.391679	0.412995	0.360445
f2	0.406583	0.533828	1.000000	0.062938	0.763651	0.391676	0.644779	0.708027	0.595129	0.720760	0.383248	0.603800	0.483725
f3	0.055892	0.042697	0.062938	1.000000	0.091203	0.091251	0.086518	0.099176	0.007368	0.035587	0.121515	0.053929	0.175260
f4	0.420972	0.516604	0.763651	0.091203	1.000000	0.302188	0.731470	0.769230	0.611441	0.668023	0.188933	0.590879	0.427321
f5	0.219247	0.311991	0.391676	0.091251	0.302188	1.000000	0.240265	0.205246	0.209847	0.292048	0.355501	0.613808	0.695360
f6	0.352734	0.569537	0.644779	0.086518	0.731470	0.240265	1.000000	0.747881	0.456022	0.506456	0.261515	0.602339	0.376955
f7	0.379670	0.664408	0.708027	0.099176	0.769230	0.205246	0.747881	1.000000	0.494588	0.534432	0.232471	0.496996	0.249929
f8	0.625505	0.311948	0.595129	0.007368	0.611441	0.209847	0.456022	0.494588	1.000000	0.910228	0.464741	0.488676	0.381626
f9	0.582764	0.314563	0.720760	0.035587	0.668023	0.292048	0.506456	0.534432	0.910228	1.000000	0.460853	0.543993	0.468536
f10	0.289946	0.391679	0.383248	0.121515	0.188933	0.355501	0.261515	0.232471	0.464741	0.460853	1.000000	0.374044	0.507787
f11	0.455621	0.412995	0.603800	0.053929	0.590879	0.613808	0.602339	0.496996	0.488676	0.543993	0.374044	1.000000	0.737663
labels	0.388305	0.360445	0.483725	0.175260	0.427321	0.695360	0.376955	0.249929	0.381626	0.468536	0.507787	0.737663	1.000000

The information presented in the above matrix would become more clear when we used a heatmap visualization as shown below. In the visualization cells in the DataFrame containing greater values will have darker shades.

```
df_corr.style.background_gradient()
```

	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	labels
f0	1.000000	0.200469	0.406583	0.055892	0.420972	0.219247	0.352734	0.379670	0.625505	0.582764	0.289946	0.455621	0.388305
f1	0.200469	1.000000	0.533828	0.042697	0.516604	0.311991	0.569537	0.664408	0.311948	0.314563	0.391679	0.412995	0.360445
f2	0.406583	0.533828	1.000000	0.062938	0.763651	0.391676	0.644779	0.708027	0.595129	0.720760	0.383248	0.603800	0.483725
f3	0.055892	0.042697	0.062938	1.000000	0.091203	0.091251	0.086518	0.099176	0.007368	0.035587	0.121515	0.053929	0.175260
f4	0.420972	0.516604	0.763651	0.091203	1.000000	0.302188	0.731470	0.769230	0.611441	0.668023	0.188933	0.590879	0.427321
f5	0.219247	0.311991	0.391676	0.091251	0.302188	1.000000	0.240265	0.205246	0.209847	0.292048	0.355501	0.613808	0.695360
f6	0.352734	0.569537	0.644779	0.086518	0.731470	0.240265	1.000000	0.747881	0.456022	0.506456	0.261515	0.602339	0.376955
f7	0.379670	0.664408	0.708027	0.099176	0.769230	0.205246	0.747881	1.000000	0.494588	0.534432	0.232471	0.496996	0.249929
f8	0.625505	0.311948	0.595129	0.007368	0.611441	0.209847	0.456022	0.494588	1.000000	0.910228	0.464741	0.488676	0.381626
f9	0.582764	0.314563	0.720760	0.035587	0.668023	0.292048	0.506456	0.534432	0.910228	1.000000	0.460853	0.543993	0.468536
f10	0.289946	0.391679	0.383248	0.121515	0.188933	0.355501	0.261515	0.232471	0.464741	0.460853	1.000000	0.374044	0.507787
f11	0.455621	0.412995	0.603800	0.053929	0.590879	0.613808	0.602339	0.496996	0.488676	0.543993	0.374044	1.000000	0.737663
labels	0.388305	0.360445	0.483725	0.175260	0.427321	0.695360	0.376955	0.249929	0.381626	0.468536	0.507787	0.737663	1.000000

Each cell in the correlation matrix represents the correlation value between the columns represented by its index. In other words the cell corresponding to index : (f4, f7) will contain the correlation value between columns f4 and f7 respectively and similarly for all other columns. As can be seen from the heat mapped DataFrame shown above, the diagonal of the correlation matrix has the darkest shade (highest correlation values of ones) since they correspond to the correlations of all the columns with themselves. Also note that the correlation matrix is symmetric with respect to its diagonal.

When doing feature selection based on correlation we have the following two objectives:

1. We want to choose those features (columns) that are more correlated to the labels, since we will be ultimately using these features to predict the labels. Hence we drop those features that have a correlation lesser than some specified threshold.
2. We want to select those features that are not too correlated with each other, since a high degree of correlation between two features would mean that both of them are similar and only one of them would actually be actually necessary.

We achieve the objective mentioned above as follows:

1. We first create a copy of the DataFrame df_corr as shown below.

```
df_corr1 = df_corr.copy().abs()
```

2. The "labels" column of the df_corr1 shown above, contains the correlations of all the columns with respect to the labels. We first sort the labels in descending order as shown in the code below.

```
ser_label_corr = df_corr1.labels.sort_values(ascending = False)
ser_label_corr
```

```
labels    1.000000
f11       0.737663
f5        0.695360
f10       0.507787
f2        0.483725
f9        0.468536
f4        0.427321
f0        0.388305
f8        0.381626
f6        0.376955
f1        0.360445
f7        0.249929
f3        0.175260
Name: labels, dtype: float64
```

The **series** object **ser_label_corr** contains the correlations of all the features with the labels column as shown above. This Series object also contains the value corresponding to the correlation of the labels with itself. We drop this value as shown below.

```
ser_label_corr = ser_label_corr.drop('labels')
ser_label_corr
```

```
f11    0.737663
f5     0.695360
f10    0.507787
f2     0.483725
f9     0.468536
f4     0.427321
f0     0.388305
f8     0.381626
f6     0.376955
f1     0.360445
f7     0.249929
f3     0.175260
Name: labels, dtype: float64
```

3. We then drop all features that have label correlation values less than some specified threshold (say 0.3). This step corresponds to Objective 1 mentioned earlier. This is done as shown below.

```
cond = df_corr1.labels >= 0.3
df_corr1 = df_corr1[cond]
df_corr1 = df_corr1.loc[df_corr1.index.values, df_corr1.index.values]

ser_label_corr = df_corr1.labels.sort_values(ascending = False).drop('labels')
```

```
df_corr1
```

	f0	f1	f2	f4	f5	f6	f8	f9	f10	f11	labels
f0	1.000000	0.200469	0.406583	0.420972	0.219247	0.352734	0.625505	0.582764	0.289946	0.455621	0.388305
f1	0.200469	1.000000	0.533828	0.516604	0.311991	0.569537	0.311948	0.314563	0.391679	0.412995	0.360445
f2	0.406583	0.533828	1.000000	0.763651	0.391676	0.644779	0.595129	0.720760	0.383248	0.603800	0.483725
f4	0.420972	0.516604	0.763651	1.000000	0.302188	0.731470	0.611441	0.668023	0.188933	0.590879	0.427321
f5	0.219247	0.311991	0.391676	0.302188	1.000000	0.240265	0.209847	0.292048	0.355501	0.613808	0.695360
f6	0.352734	0.569537	0.644779	0.731470	0.240265	1.000000	0.456022	0.506456	0.261515	0.602339	0.376955
f8	0.625505	0.311948	0.595129	0.611441	0.209847	0.456022	1.000000	0.910228	0.464741	0.488676	0.381626
f9	0.582764	0.314563	0.720760	0.668023	0.292048	0.506456	0.910228	1.000000	0.460853	0.543993	0.468536
f10	0.289946	0.391679	0.383248	0.188933	0.355501	0.261515	0.464741	0.460853	1.000000	0.374044	0.507787
f11	0.455621	0.412995	0.603800	0.590879	0.613808	0.602339	0.488676	0.543993	0.374044	1.000000	0.737663
labels	0.388305	0.360445	0.483725	0.427321	0.695360	0.376955	0.381626	0.468536	0.507787	0.737663	1.000000

```
ser_label_corr
```

```
f11    0.737663
f5     0.695360
f10    0.507787
f2     0.483725
f9     0.468536
f4     0.427321
f0     0.388305
f8     0.381626
f6     0.376955
f1     0.360445
Name: labels, dtype: float64
```

As can be seen from the outputs above the code drops all rows and columns from df_corr1 that have label correlations less than 0.3 and also modifies **ser_label_corr** to contain only the chosen features.

4. We then isolate the **top feature** represented in **ser_label_corr** (ie: f11), from the other top features using the function **fn_top_features** shown below.

```
def fn_top_features(ser_label_corr):

    list0_top_feats = list(ser_label_corr.index)
    top_feat = list0_top_feats.pop(0)

    return top_feat, list0_top_feats
```

```
filtered_feats = []
```

```
top_feat, list0_other_top_feats = fn_top_features(ser_label_corr)
filtered_feats.append(top_feat)

print(list0_other_top_feats)
print(filtered_feats)
```

```
['f5', 'f10', 'f2', 'f9', 'f4', 'f0', 'f8', 'f6', 'f1']
['f11']
```

As can be seen from the code shown above, we also collect the top feature in the list **filtered_feats** and the other top features in **list list0_other_top_feats.**

5. We then drop all features in **list0_other_top_feats** that have a correlation higher than some specified threshold (say 0.6) with the top feature. This is done using the code shown below.

```
thresh_feat = 0.6
for f in list0_other_top_feats:

    if df_corr1.loc[top_feat, f] >= thresh_feat:

        df_corr1 = df_corr1.drop(f, axis = 0)
        df_corr1 = df_corr1.drop(f, axis = 1)

df_corr1 = df_corr1.drop(top_feat, axis = 0)
df_corr1 = df_corr1.drop(top_feat, axis = 1)

print(df_corr.shape, df_corr1.shape)
df_corr1
```

Drop rows & columns from df_corr1 that have corr >= 0.6

Drop **top_feat** row & column from df_corr1

```
(13, 13) (7, 7)
```

	f0	f1	f4	f8	f9	f10	labels
f0	1.000000	0.200469	0.420972	0.625505	0.582764	0.289946	0.388305
f1	0.200469	1.000000	0.516604	0.311948	0.314563	0.391679	0.360445
f4	0.420972	0.516604	1.000000	0.611441	0.668023	0.188933	0.427321
f8	0.625505	0.311948	0.611441	1.000000	0.910228	0.464741	0.381626
f9	0.582764	0.314563	0.668023	0.910228	1.000000	0.460853	0.468536
f10	0.289946	0.391679	0.188933	0.464741	0.460853	1.000000	0.507787
labels	0.388305	0.360445	0.427321	0.381626	0.468536	0.507787	1.000000

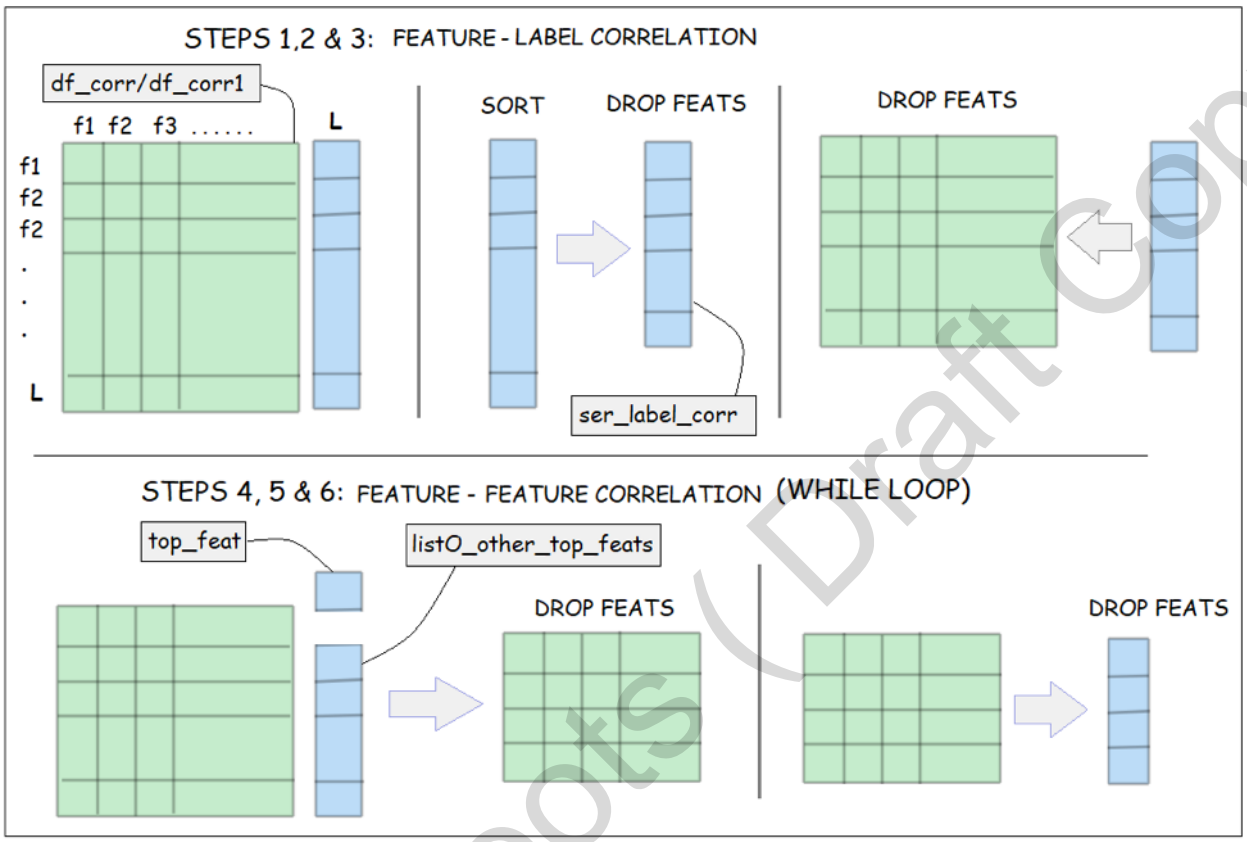
As can be seen from the output shown above, the size of **df_corr** is reduced from (13, 13) to (7, 7).

6. We repeat step 2 to 5 described above till all the values in **list0_other_top_feats** are exhausted. This is done by organizing the code shown in the steps above into a set of three functions shown below (including **fn_top_features** described earlier)..

```
def fn_drop_corr_feats(df_corr, top_feat,
                       list0_other_top_feats, thresh = 0.7):

    for f in list0_other_top_feats:

        if df_corr.loc[top_feat, f] >= thresh:

            df_corr = df_corr.drop(f, axis = 0)
            df_corr = df_corr.drop(f, axis = 1)

    df_corr = df_corr.drop(top_feat, axis = 0)
    df_corr = df_corr.drop(top_feat, axis = 1)

    return df_corr
```

```
def fn_feat_select(df_corr, thresh_label = 0.3, thresh_feat = 0.4, verbose = False):

    df_corr1 = df_corr.copy().abs()
    cond = df_corr1.labels >= thresh_label
    df_corr1 = df_corr1[cond]
    df_corr1 = df_corr1.loc[df_corr1.index.values, df_corr1.index.values]
    filtered_feats = []
    ser_label_corr = df_corr1.labels.sort_values(ascending = False).drop('labels')

    if verbose == True:
        print('Following feats have corr >', thresh_label, 'with the labels:')
        print(ser_label_corr.index.values)
        print()

    while len(ser_label_corr) > 0 :

        top_feat, list0_other_top_feats = fn_top_features(ser_label_corr)
        filtered_feats.append(top_feat)

        if verbose == True:
            print('Checking corr of',  top_feat, 'with', list0_other_top_feats)

        df_corr1 = fn_drop_corr_feats(df_corr1, top_feat, list0_other_top_feats, thresh = thresh_feat)
        ser_label_corr = df_corr1.labels.sort_values(ascending = False).drop('labels')

    return filtered_feats
```

The function **fn_feat_select** shown above uses two other functions - **fn_top_features** and **fn_drop_corr_feats** to repeat the steps 2 to 5 described earlier. It incorporates a **while** loop conditioned on the length of **ser_label_corr** to do this.



Demonstrated below is the use of the code shown above to do feature selection using thresholds of 0.45 for feature - label correlation (**thresh_label**) and 0.70 for feature - feature correlation (**thresh_feat**). The verbose kwarg lets one inspect the internal computation status while the function is running.

```
df_corr = df.corr().abs()
best_feats = fn_feat_select(df_corr, thresh_label = 0.45,
                                     thresh_feat = 0.70,
                                         verbose = True)
best_feats
```

```
Following feats have corr > 0.45 with the labels:
['f11' 'f5' 'f10' 'f2' 'f9']

Checking corr of f11 with ['f5', 'f10', 'f2', 'f9']
Checking corr of f5 with ['f10', 'f2', 'f9']
Checking corr of f10 with ['f2', 'f9']
Checking corr of f2 with ['f9']
['f11', 'f5', 'f10', 'f2']
```

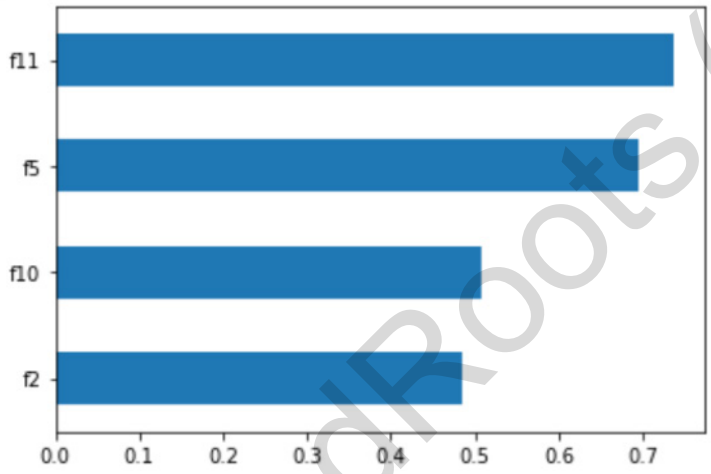
We can slice the DataFrame containing the original correlation matrix (ie: df_corr) using the "filtered features" resulting from the feature selection and check the correlations using a heatmap as shown below.

```
df_corr.loc[best_feats, best_feats].style.background_gradient()
```

	f11	f5	f10	f2
f11	1.000000	0.613808	0.374044	0.603800
f5	0.613808	1.000000	0.355501	0.391676
f10	0.374044	0.355501	1.000000	0.383248
f2	0.603800	0.391676	0.383248	1.000000

We can check for correlations of these features with respect to the labels using a bar plot as shown below

```
df_corr.loc[best_feats].labels.sort_values().plot(kind = 'barh')
```



We can infer from both plots shown above, that the features selected are contained within the thresholds specified.

This concludes the seventh chapter. Next chapter we shall explore the matplotlib library which deals with data visualization and plotting.

8. MATPLOTLIB

MATPLOTLIB

Visualization of Data is one of the easiest ways to get insight into data and forms the basis of exploratory data analysis. Matplotlib is a widely used python data visualization library that offers the user two basic interfaces for visualizing data: A **high level imperative interface** and an **object oriented interface**. The high level imperative interface allows one to quickly create visualizations/plots without needing to specify the procedure for implementing the plots. The object oriented interface uses **methods** associated with two main inbuilt object classes **(figure & axes)** that form the basis for describing/specifying the features of the plot we want. The object oriented interface is more **low level** and it gives one the flexibility to create plots as per one's own specificity.

The pandas **plot** method used in the previous chapter, basically uses matplotlib under the hood to create the visualizations demonstrated.

TYPES OF VISUALIZATIONS/PLOTS:

Data visualization in a vast topic by itself, and there are many ways to visualize data, but at a fundamental level there are basically two types of data we want to analyse - **Numeric** and **Categorical** and there are two main basic types of analysis - **Univariate** and **Multivariate** analysis.

Univariate analysis deals with analysing a single **Random Variable** (ie: single feature or column of Tabular data) and Multivariate analysis deals with analysing relationships between two (**Bivariate** analysis) or more Random Variables.

Under **univariate** analysis we will explore the following plot types:

1. Histograms
2. KDE plots
3. Box plots
4. Violin plots
5. Bar plots

Under **multivariate** analysis we will explore the following plot types:

1. Line plots
2. Scatter plots

HISTOGRAMS & BOXPLOTS:

A histogram is used to visually represent the distribution of numerical data. To construct a histogram we divide the entire range of values of the Random Variable into a series of intervals (bins) and then **count** how many values fall into each interval. The bins are specified as consecutive, non-overlapping intervals. The count of the values within each bin is represented as rectangles whose height is proportional to the counts.

To use matplotlib we first import it and all other required libraries as shown below.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

For demonstration purposes we will be using the 'Australia weather" dataset, which contains ten years of weather observations from various locations in Australia.

```
df = pd.read_csv('aus_weather.csv').dropna()

print(df.shape)
df.head()
```

```
(56420, 12)
```

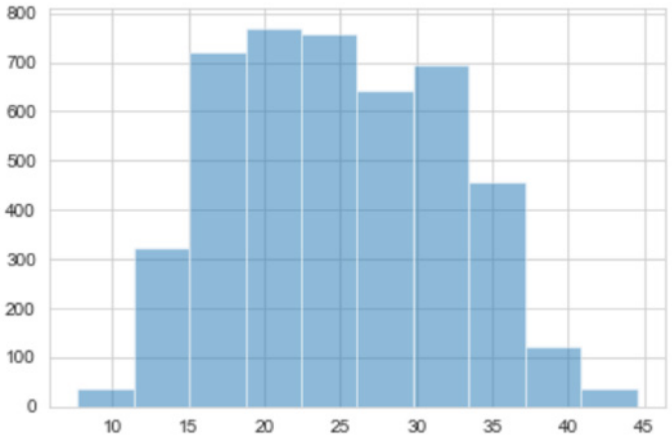
	year	month	day	location	mintemp	maxtemp	rainfall	sunshine	humidity	cloud	windspeed	winddir
0	2009	1	1	Cobar	17.9	35.2	0.0	12.3	16.5	3.5	48.0	SSW
1	2009	1	2	Cobar	18.4	28.9	0.0	13.0	19.0	1.0	37.0	S
2	2009	1	4	Cobar	19.4	37.6	0.0	10.6	32.0	3.5	46.0	NNE
3	2009	1	5	Cobar	21.9	38.4	0.0	12.2	29.5	3.0	31.0	WNW
4	2009	1	6	Cobar	24.2	41.0	0.0	8.4	17.0	3.5	35.0	WNW

```
df.describe().round()
```

	year	month	day	mintemp	maxtemp	rainfall	sunshine	humidity	cloud	windspeed
count	56420.0	56420.0	56420.0	56420.0	56420.0	56420.0	56420.0	56420.0	56420.0	56420.0
mean	2012.0	6.0	16.0	13.0	24.0	2.0	8.0	58.0	4.0	41.0
std	2.0	3.0	9.0	6.0	7.0	7.0	4.0	18.0	2.0	13.0
min	2007.0	1.0	1.0	-7.0	4.0	0.0	0.0	0.0	0.0	9.0
25%	2010.0	3.0	8.0	9.0	19.0	0.0	5.0	47.0	2.0	31.0
50%	2012.0	6.0	16.0	13.0	24.0	0.0	9.0	59.0	4.0	39.0
75%	2014.0	9.0	23.0	18.0	30.0	1.0	11.0	70.0	6.0	48.0
max	2017.0	12.0	31.0	31.0	48.0	206.0	14.0	100.0	8.0	124.0

We gain insight into the distributions of any of the numerical random variables (ie: columns/features) by plotting its histogram using the **hist** function. This is demonstrated on the maxtemp feature as shown below.

```
df_2016 = df[df.year == 2016]
rv1 = df_2016.maxtemp.values

plt.hist(rv1, bins = 10, alpha = 0.5)
plt.draw()
```

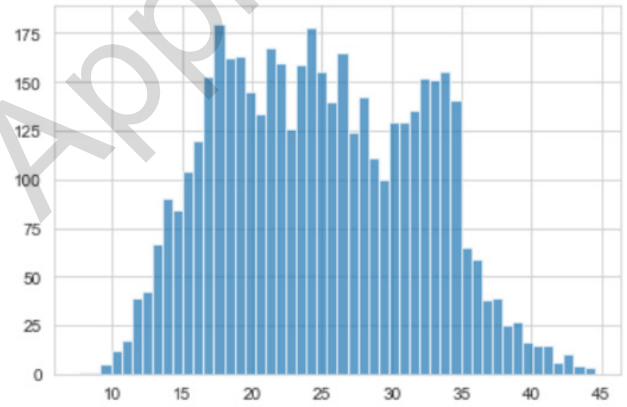


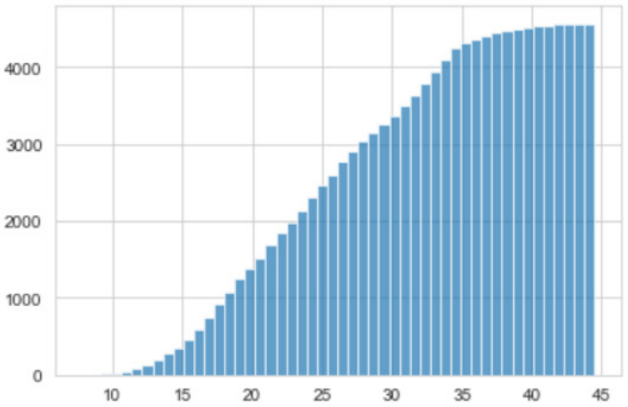
Shown above is the visualization of the **distribution** of maximum temperature in the year of 2016. The **bins** keyword argument (**kwarg**) gives us the option of deciding the granularity of the plot. The more the number of bins the more detailed the representation becomes. The **alpha** kwarg allows one to set the degree of opacity of the colors of the plot. The value of alpha can be any value between 0 and 1, where 0 represents transparent.

Shown below is the same plot, but with the number of bins increased.

```
plt.hist(rv1, bins = 50, alpha = 0.7)

plt.draw()
```

```
plt.hist(rv1, bins = 50, alpha = 0.7,
         cumulative = True)

plt.draw()
```

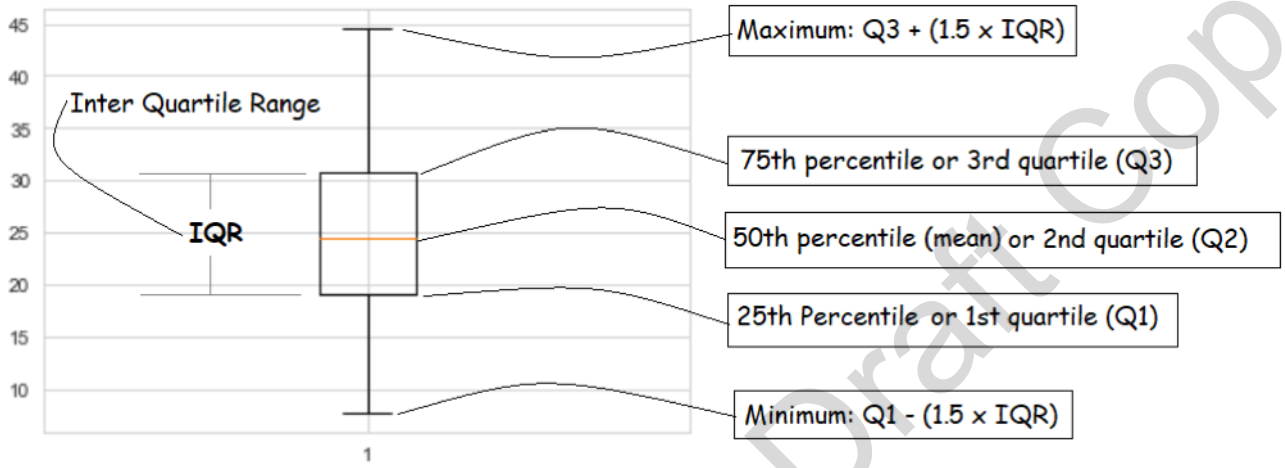




Note that the cumulative keyword argument of the hist function causes it to plot a cumulative distribution frequency (CDF) plot if set to True.

The same information can be visualized in the form of a box plot using the boxplot function as shown below.

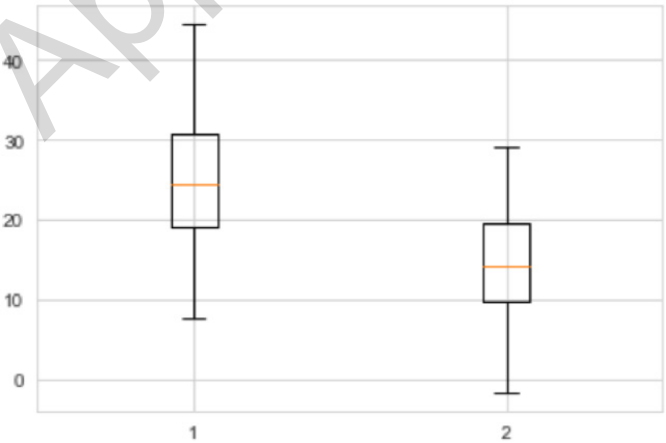
```
plt.boxplot(rv1)
plt.draw()
```



As can be seen from the image above, box plots represent the distribution in terms of the percentiles that make up its **Inter Quartile Range**. The interquartile range represents the interval within which the **middle 50%** of the data is encompassed. The minimum and maximum thresholds defined above are basically used to indicate **outliers**. Any point lying outside these thresholds can be considered as an outlier.

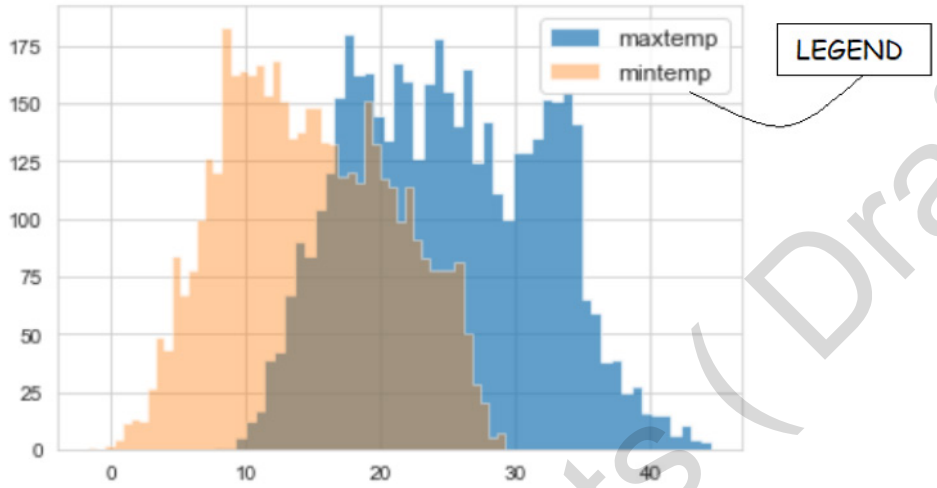
Comparisons can be made between two or more distributions by passing a list of Random Variables to the boxplot function instead of just one Random Variable. This is demonstrated in the code below.

```
rv2 = df_2016.mintemp.values

plt.boxplot([rv1, rv2])
plt.draw()
```



One can also plot more than one distribution using the histogram plots as shown below.

```
kwarg1 = dict(alpha = 0.7, label = 'maxtemp', histtype = 'stepfilled')
kwarg2 = dict(alpha = 0.4, label = 'mintemp', histtype = 'stepfilled')

plt.hist(rv1, bins = 50, **kwarg1)
plt.hist(rv2, bins = 50, **kwarg2)

plt.legend(fontsize = 12)
plt.draw()
```



By sequentially executing histogram functions as shown above, we can plot more than one histogram in a single plot (this generally applies to any other matplotlib plotting function).

Note the following details about the plot shown above:

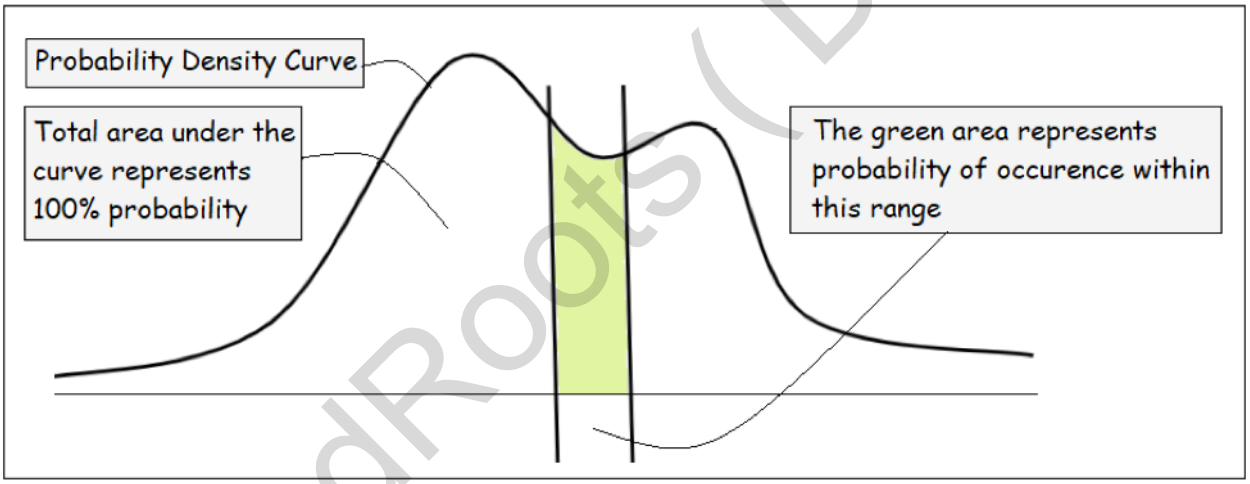
1. How the kwargs are fed to the hist function by using the **double star** operator for dict unpacking. This makes the code more neat. It is used instead of writing all the kwargs inside the function's curved bracket.
2. How the **alpha** kwarg values have been used such that the hist function executed second is more transparent that the one that precedes it.
3. How the **histtype** kwarg is used to define how we want the histogram to be plotted - as **bars** or **steps** and if steps whether to be filled with color (**stepfilled**) or not (**step**).
4. How the **label** kwarg is used to create **legends** (category identification incase of multiple features being compared) for the plot. Legends are only displayed if the **legend** function is executed after the plotting function.
5. How the **legend** function is used after all the plotting functions. The legend function lets us specify the attributes of the legend for all the plots created prior to it. Here we set the fontsize to 12, by defining it as such against the function's **fontsize** kwarg.

Matplotlib provides "**plot attribute specification functions**" that allow one to define customspecifications to the "**blank canvas**" details of the plot - like title, legend, font size, text orientation of the values displayed on the plot's axes and so on. Some of the relevant functions are : **xlabel, ylabel, xticks, yticks, title** and **legend**, All of these functions will be explored and used further along in the chapter.

Note that these "blank canvas" functions are only used **after** some plotting function (**hist** in the previous case) and they modify the default values of the plot attributes they represent (**legend** in the previous case) to those specified by the programmer.

KDE PLOTS:

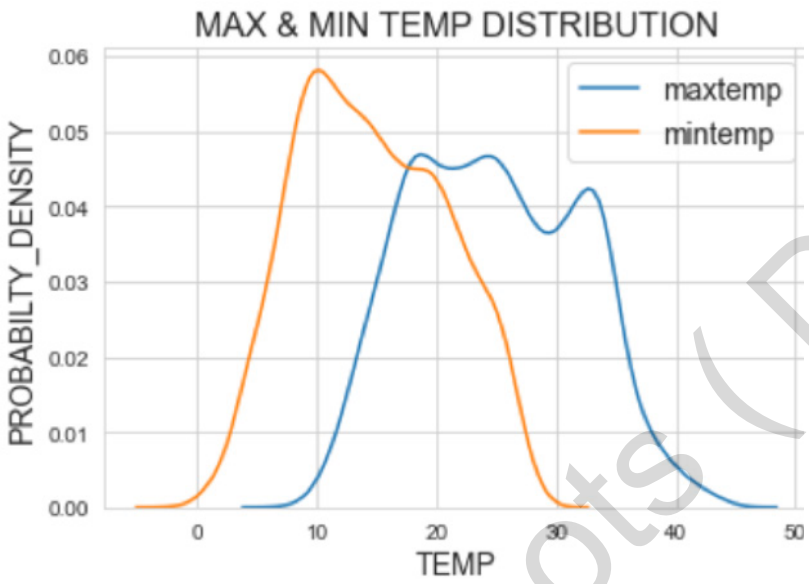
KDE plots are plots that describe distributions in terms of their Probability Density Curve. The Probability Density Curve of a Random Variable is a curve whose areas under the curve (AUC) represents the probability of occurrence of the values of the Random Variable. This is expressed in the image below.



KDE stands for Kernel Density Estimation, which is a method for estimating the probability density curve of a Random Variable. A function for plotting KDE plots is not directly available in matplotlib. One can use the **seaborn** visualization package for this purpose. The seaborn library is built on top of matplotlib and has some more refinements when compared to matplotlib. Seaborn has a function called the **kdeplot** function specifically defined for creating KDE plots. This is demonstrated in the code shown below.

```
import seaborn as sns
```

```
sns.kdeplot(rv1, label = 'maxtemp')
sns.kdeplot(rv2, label = 'mintemp')

plt.title('MAX & MIN TEMP DISTRIBUTION', fontsize = 16)
plt.xlabel('TEMP', fontsize = 14)
plt.ylabel('PROBABILTY_DENSITY', fontsize = 14)
plt.legend(fontsize = 14)
plt.show()
```



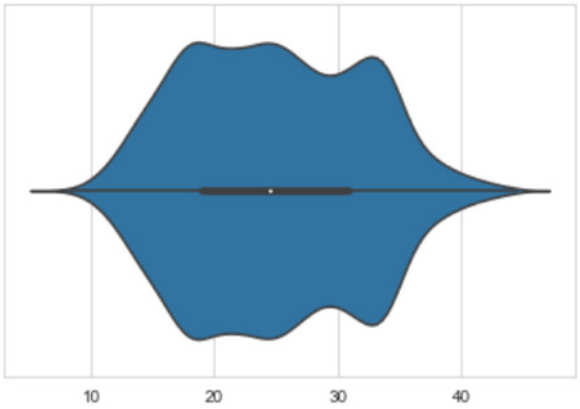
Note the following aspects of the code/plot shown above:

1. How the title, **xlabel, & ylabel** functions are used to name the plot and its axes.
2. How the font sizes of the title, axes labels and legend, have been suitably enlarged using the **fontsize** kwarg.

VIOLIN PLOTS:

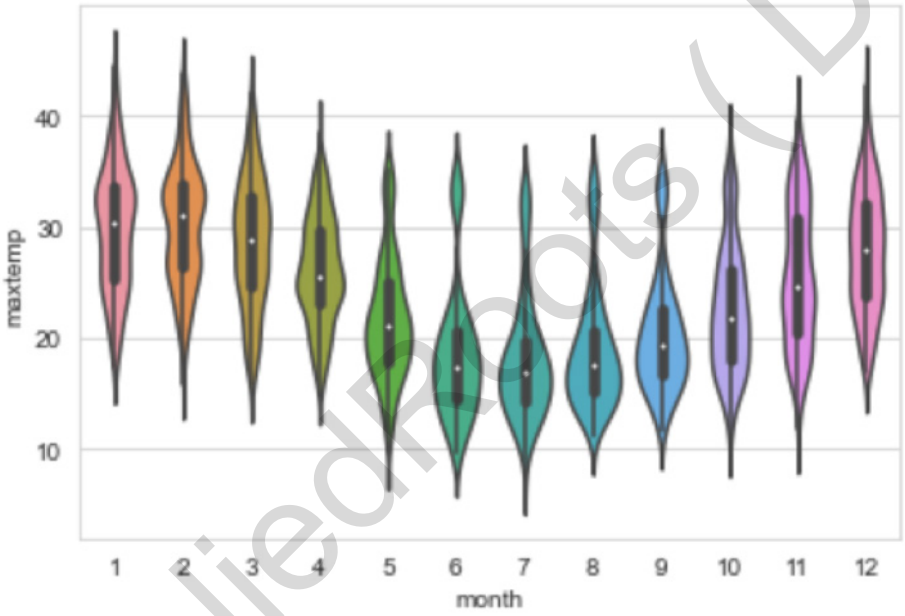
THe violin plot is basically a slight modification of the KDE plot, where the Probability Density Curve is represented as shown below. The Top half of the violin plot is the same as a normal KDE plot, the bottom half is the "reflection" of the top half. Violin plots are plotted using the seaborn library as shown below.

```
sns.violinplot(x = rv1)
plt.show()
```



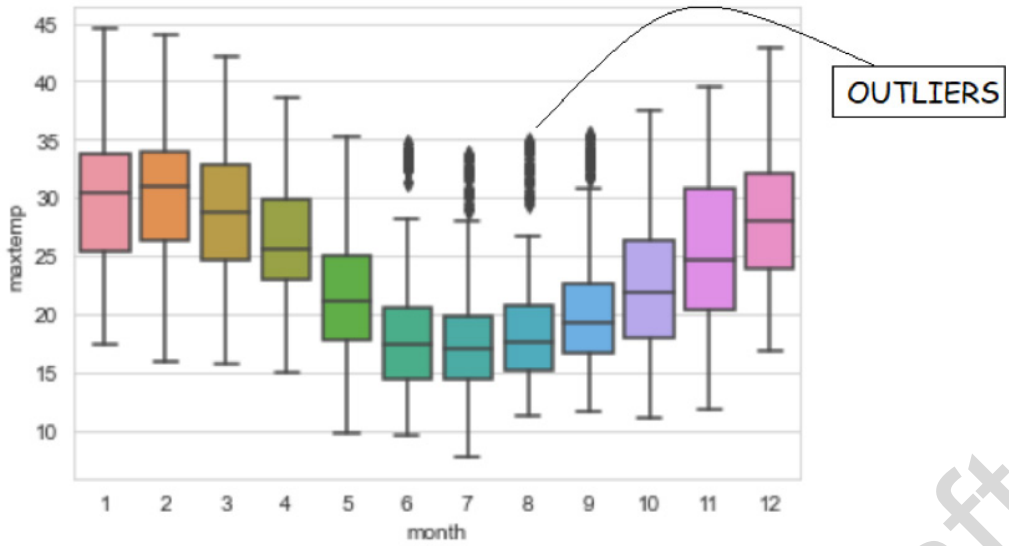
Violin plots are more useful when comparing the Probability Distributions of multiple Random Variables. This is demonstrated in the code shown below.

```
sns.violinplot(x = 'month', y = 'maxtemp', data = df_2016)
plt.show()
```

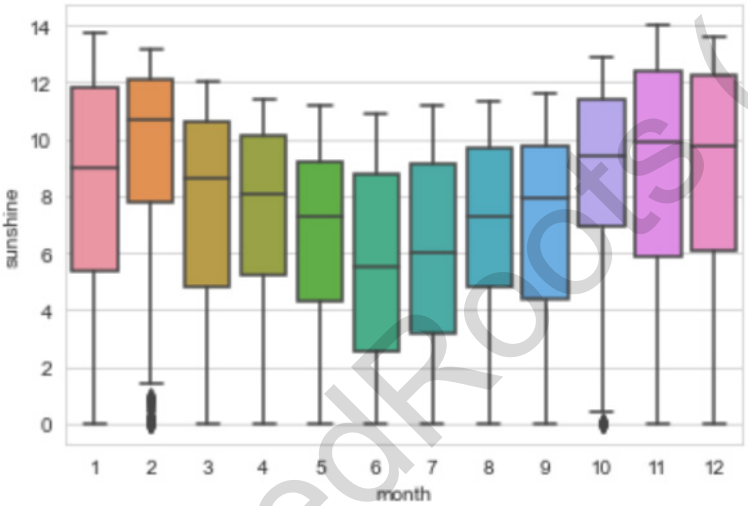


Note how seaborn makes it easier to plot comparative violin plots, by plotting them with respect to the DataFrame in question (ie: df_2016). The seaborn library also makes it convenient to plot comparative boxplots as shown below.

```
sns.boxplot(x = 'month', y = 'maxtemp', data = df_2016)
plt.show()
```




```
sns.boxplot(x = 'month', y = 'sunshine', data = df_2016)
plt.show()
```

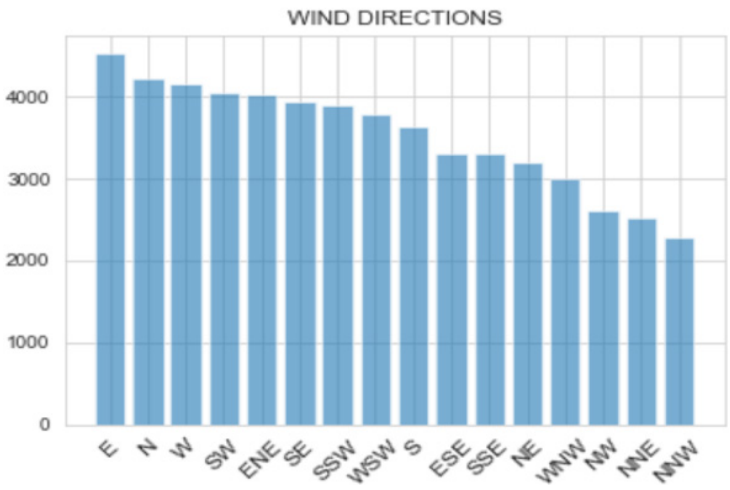


Note the correlation between the **maxtemp** and **sunshine** Random Variables (ie: features) as expressed in the two plots shown above. Also note how **outliers** are indicated outside the max and min thresholds in the box plots above.

BAR PLOTS:

A bar plot or graph represents categorical data with rectangular bars with heights or lengths proportional to the values that they represent as shown in the plot below.

```
s_wind_dir = df.winddir.value_counts()
x = s_wind_dir.index
y = s_wind_dir.values
plt.bar(x, y, alpha = 0.6)
plt.title('WIND DIRECTIONS')
plt.xticks(fontsize = 12, rotation = 45)
plt.show()
```

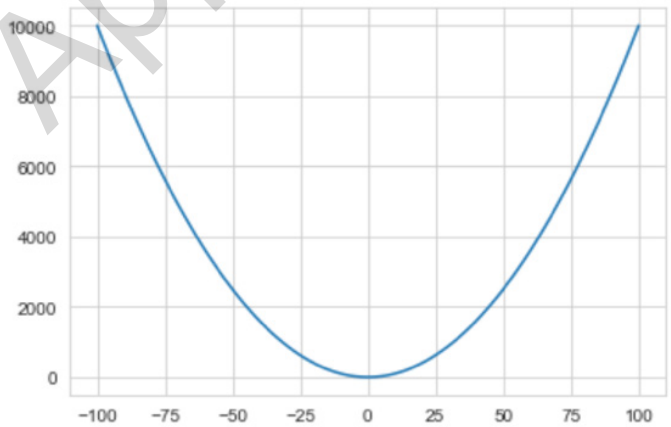


The plot above indicates that winds mostly blow in the east, north and west directions in Australia. Note the usage of the **xticks** function after the **bar** plotting function in the code shown above. The **xticks** and **yticks** functions are used to specify the attributes of the values displayed on the x & y axis respectively. In the above plot, we change the orientation of the text displayed on the x axis by using the **rotation** kwarg. This avoids the text from overlapping with each other, as they would have, if the text were aligned horizontally (default setting) and not at 45 degrees as specified.

LINE PLOTS:

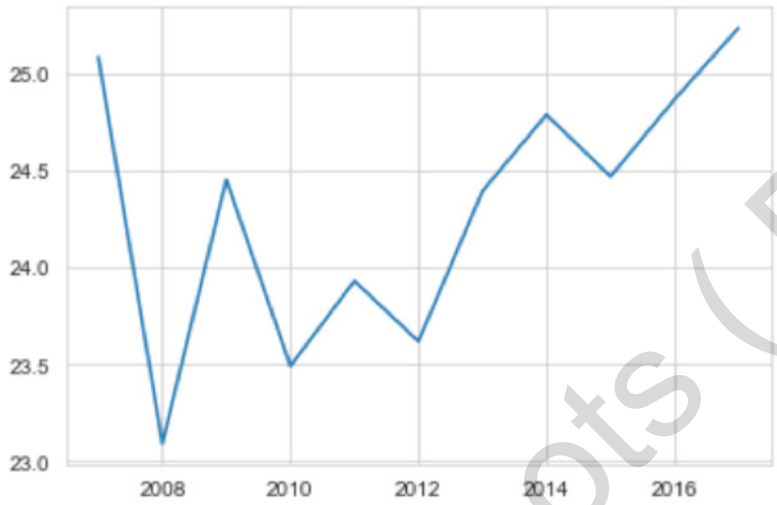
Line plots are used to visualize the relationship between two (or more) Random Variables where the y axis represents Random Variables that are a function of (or represented with respect to) the variable plotted along the x axis as demonstrated in the code shown below. We use the **plot** function to plot line plots.

```
x = np.linspace(-100, 100, 100)
y = x**2

plt.plot(x, y)
plt.show()
```



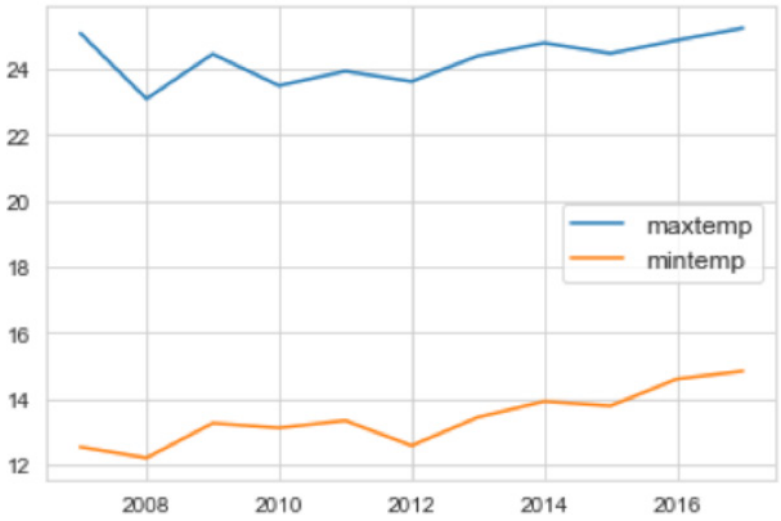
Line plots have a directional (or progressive) nature and they require that the values of the Random Variable represented along the x axis be sorted. Consider the example of the line plot shown below. The plot represents the variation in the mean maximum temperature in Australia between the years 2007 to 2017.

```
s_yrly_max_temp = df.groupby('year').mean().maxtemp
x = s_yrly_max_temp.index
y = s_yrly_max_temp.values

plt.plot(x, y)
plt.show()
```



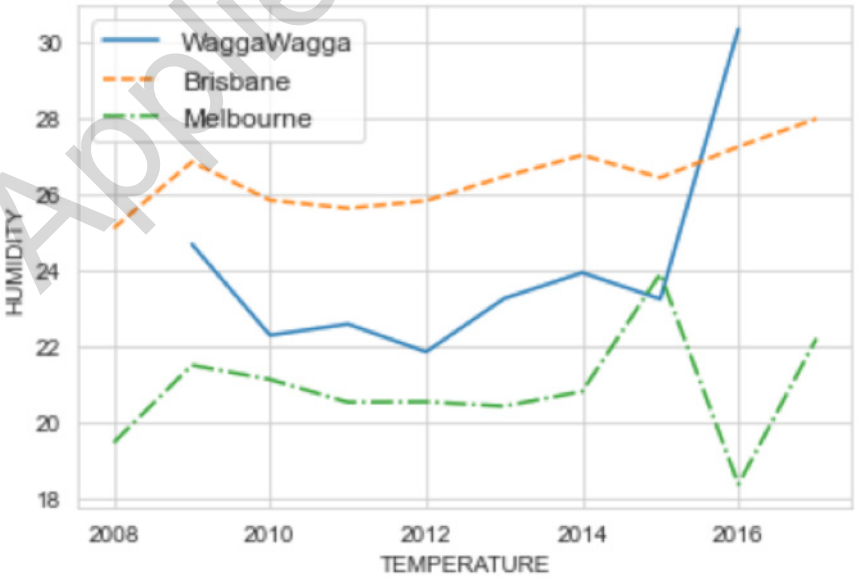
Multiple line plots can be plotted within the same plot by sequentially executing multiple plot functions as shown below.

```
ss_yrly_max_temp = df.groupby('year').mean().maxtemp
s_yrly_min_temp = df.groupby('year').mean().mintemp

x1 = s_yrly_max_temp.index
y1 = s_yrly_max_temp.values

x2 = s_yrly_min_temp.index
y2 = s_yrly_min_temp.values

plt.plot(x1, y1, label = 'maxtemp')
plt.plot(x2, y2, label = 'mintemp')
plt.legend(fontsize = 12)
plt.show()
```



Show below is another example of multiple line plots within a single plot. It depicts the changes in mean maximum temperature between the years 2007 to 2017, for three different locations in Australia. Note how the **linestyle** kwarg of the **plot** function is used to define how the lines in the line plots are plotted.

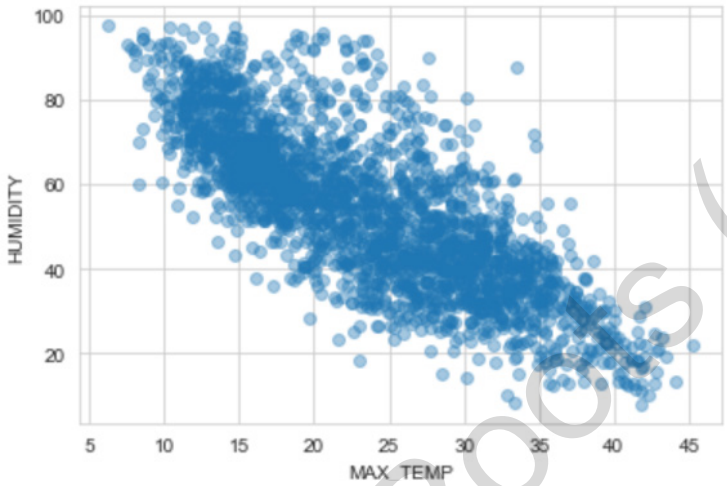
```
s1 = df[df.location == 'WaggaWagga'].groupby('year').mean().maxtemp
s2 = df[df.location == 'Brisbane'].groupby('year').mean().maxtemp
s3 = df[df.location == 'Melbourne'].groupby('year').mean().maxtemp

plt.plot(s1.index, s1.values, label = 'WaggaWagga', linestyle = '-')
plt.plot(s2.index, s2.values, label = 'Brisbane', linestyle = '--')
plt.plot(s3.index, s3.values, label = 'Melbourne', linestyle = '-.')

plt.xlabel('TEMPERATURE')
plt.ylabel('HUMIDITY')
plt.legend(fontsize = 12)
plt.show()
```



SCATTER PLOTS:

Scatter plots use dots to represent the positions of pairs of values formed between two numeric Random Variables of the same size (values of the same index form a pair). **Scatter** plots are created using the scatter function.Shown below is the scatter plot between **maxtemp** and **humidity Random Variables** for the location named WaggaWagga in Australia.

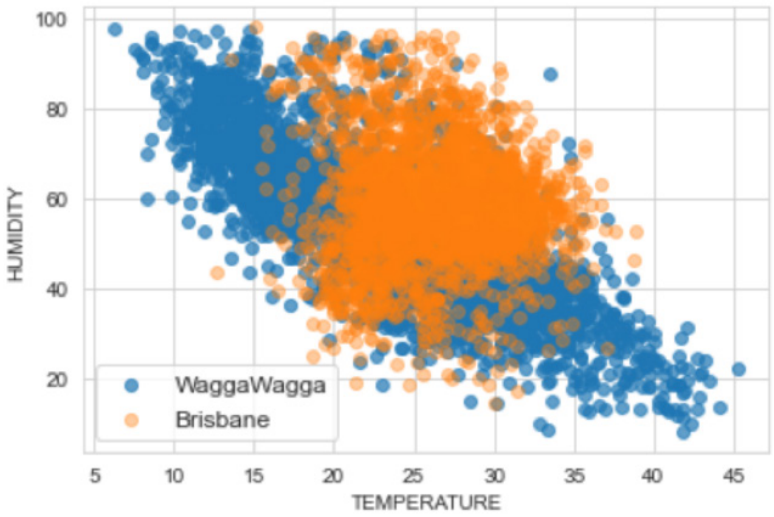
```
x1 = df[df.location == 'WaggaWagga'].maxtemp
y1 = df[df.location == 'WaggaWagga'].humidity

plt.scatter(x1, y1, alpha = 0.4)
plt.xlabel('MAX_TEMP')
plt.ylabel('HUMIDITY')
plt.show()
```



As can be seen from the scatter plot above, there seems to be an inverse linear relationship between **humidity** and **max_temp** (humidity generally decreases with increase in max_temp).

Scatter plots are ideal for comparing two categorical variables based on the relationship between two numeric random variables related to them. This is accomplished by sequentially executing multiple **scatter** functions as shown below.

```
x1 = df[df.location == 'WaggaWagga'].maxtemp
y1 = df[df.location == 'WaggaWagga'].humidity
x2 = df[df.location == 'Brisbane'].maxtemp
y2 = df[df.location == 'Brisbane'].humidity

plt.scatter(x1, y1, alpha = 0.7, label = 'WaggaWagga')
plt.scatter(x2, y2, alpha = 0.4, label = 'Brisbane')
plt.xlabel('TEMPERATURE')
plt.ylabel('HUMIDITY')
plt.legend(fontsize = 12)
plt.show()
```



SUBPLOTS (Object Oriented Interface/Paradigm):

All the previous examples implemented so far in this chapter used the **imperative** style for plotting. Using this style/technique is convenient when we want to create single plots. When one wants to create more complex plots that contain multiple subplots expressed in grid style - as rows and columns, then the object oriented style of plotting is more convenient.

The main function here is the **subplots function**. The subplots function has two arguments: number of **rows** and number of **columns** as shown below. Executing the **subplots** function results in the following outcomes:

1. Two objects are returned by the function: the first object is the matplotlib **Figure** object (assigned to variable name "figure" in the example shown below) and the second object is a numpy **array** containing matplotlib **AxesSubplot** objects (assigned to variable name "axes").
2. An empty grid of subplots is created/plotted having the specified rows and columns.

```
n_rows, n_columns = 2, 2

figure, axes = plt.subplots(n_rows, n_columns)
```

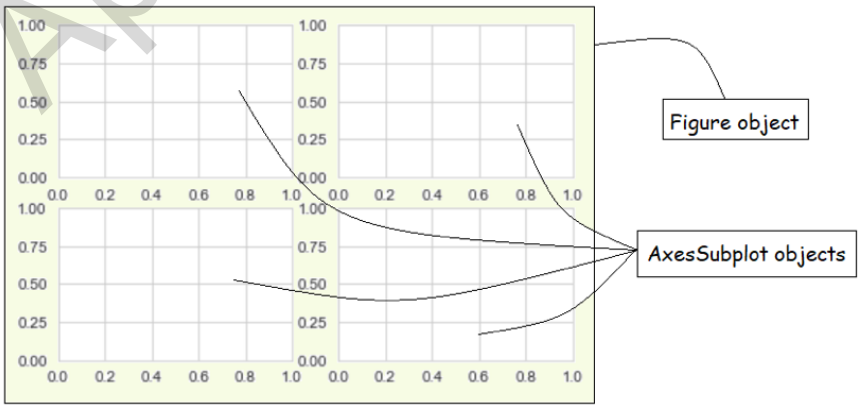


FIGURE & AXES OBJECTS:

The **figure** object which is an instance of class **plt.Figure**, can be thought of as a container that contains all other objects representing axes, graphics, legend and labels. The **AxesSubplot** objects contained within the **array** named axes, are instances of class **plt.AxesSubplot** and they represent each individual subplot. This is demonstrated in the code shown below.

```
type(axes), axes.shape, type(axes[0,0])
```

```
(numpy.ndarray, (2, 2), matplotlib.axes._subplots.AxesSubplot)
```

```
axes
```

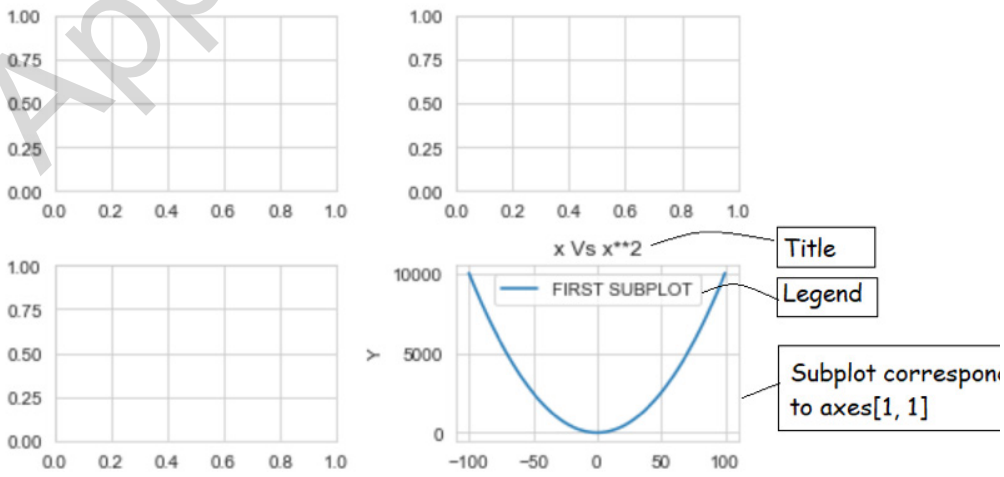
```
array([[<AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>]], dtype=object)
```

As can be seen from the output above, the axes array contains four **AxesSubplot** instances, arranged as a 2 x 2 matrix. Now if one chooses to create a plot within any of the four subplots shown above, it can be done by specifically using the AxesSubplot object that corresponds to the **index** of the desired subplot. This is demonstrated in the code shown below.

```
x = np.linspace(-100, 100, 100)
y = x**2

n_rows, n_columns = 2, 2
figure, axes = plt.subplots(n_rows, n_columns)

axes[1,1].plot(x, y, label = 'FIRST SUBPLOT')

axes[1,1].set_title('x Vs x**2')
axes[1,1].set_xlabel('X')
axes[1,1].set_ylabel('Y')
axes[1,1].legend()

figure.tight_layout()
```

Methods applied to AxesSubplot (or axes) object(s)

Method applied to figure object



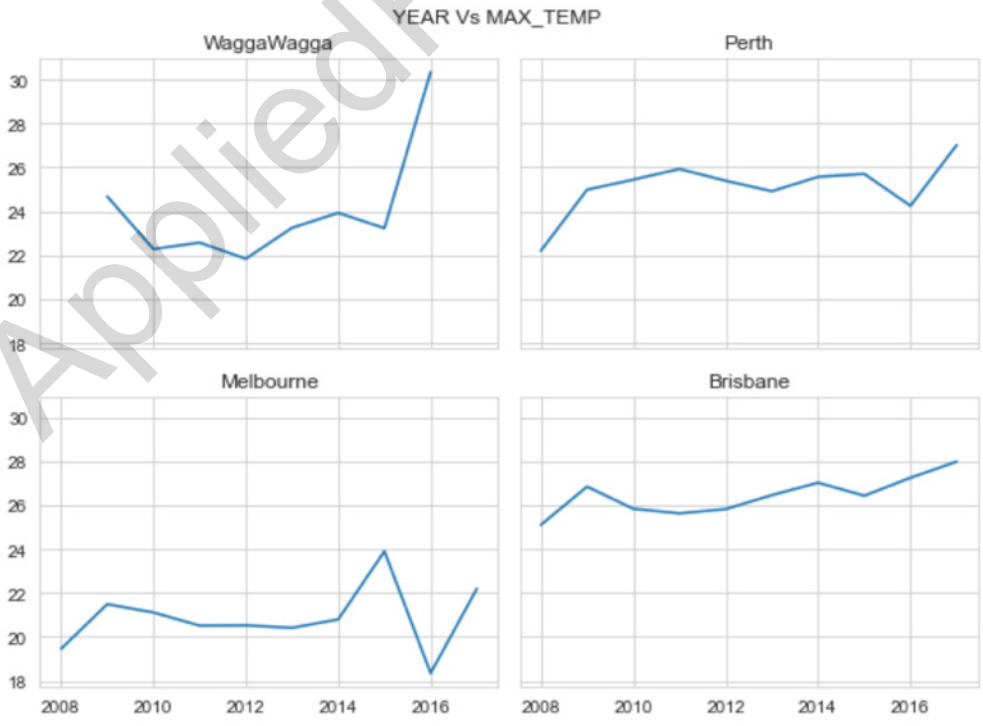
Note the subtle differences, when using the Object Oriented Paradigm:

1. All the plotting and plot attribute setting/specification is achieved by using axes or figure **methods** instead of using **functions**. Most of the functions used in the **imperative style interface** translate to object oriented **axes methods** by adding the "**set_**" prefix (ex: **xlabel** becomes **set_xlabel** and so on)
2. Only the **legend** method does not need the "**set_**" prefix to be added to it.
3. The "**tight_layout**" figure method is used at the end of all the other plotting methods and it basically makes sure all the subplots and its attributes are plotted neatly with the correct amount of spacing between each.

The use of methods that can applied individually to any axes object, makes it very convenient while specifying plots and their attributes using iteration. This is demonstrated in the code shown below.

```
listO_locs = ['WaggaWagga', 'Perth', 'Melbourne', 'Brisbane']
fig, axes = plt.subplots(2, 2, figsize = (10, 7), sharex = True, sharey = True)
axes = axes.flatten()
for idx, loc in enumerate(listO_locs):

    df_loc = df[df.location == loc]
    s_loc_maxtemp = df_loc.groupby('year').mean().maxtemp
    x, y = s_loc_maxtemp.index, s_loc_maxtemp.values
    axes[idx].plot(x, y)
    axes[idx].set_title(loc)

fig.suptitle('YEAR Vs MAX_TEMP')
fig.tight_layout()
```

Axes array is flatted to 1 dim for ease during iteration

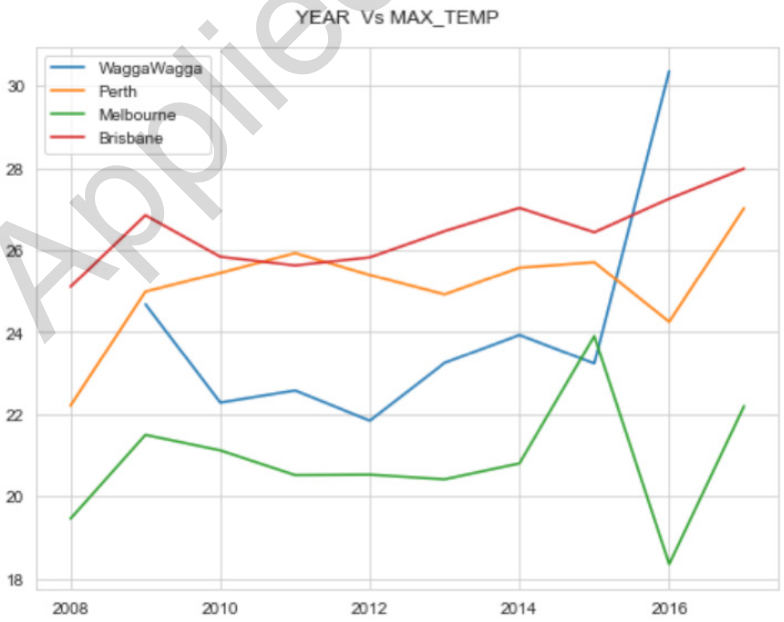


Note the following aspects of the code/plot shown above:

1. We have used a **for loop** to reduce the number of statements. If it were not for the **for loop**, we would have had to write plotting and attribute setting statements for each of the four subplots individually, instead of doing it just once as shown above.
2. The axes array is flattened and hence single 1D indexes can be used during iteration.
3. The **suptitle** method applied to the **figure** object, is used to set the title for the entire figure as a whole.
4. The **sharex** and **sharey** kwargs of the **subplot** function lets us specify whether we want to share the x or y axis among all the subplots.

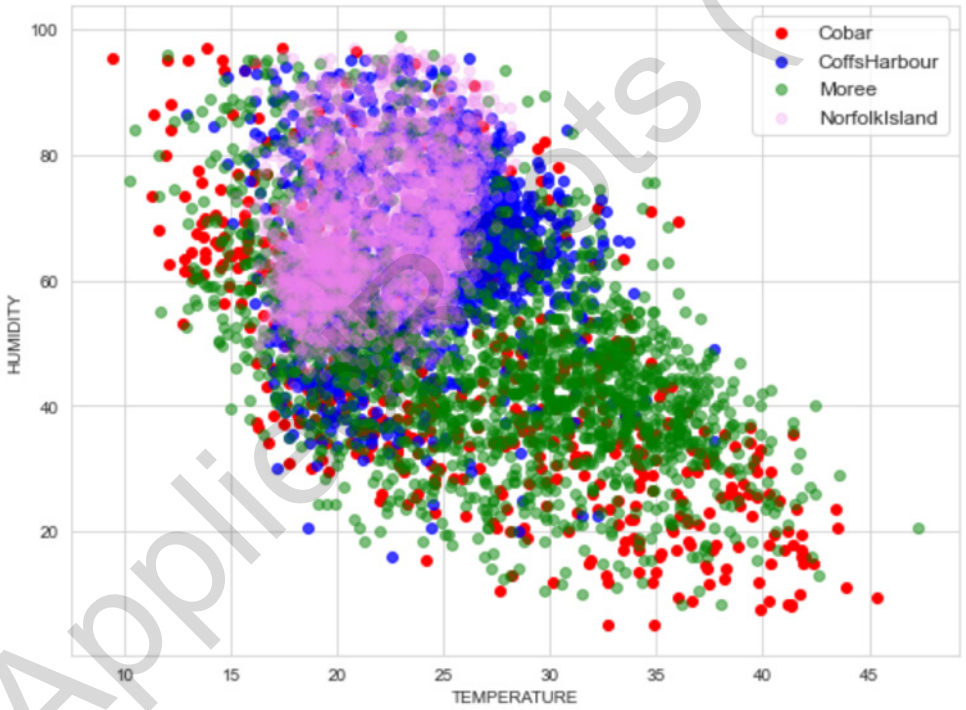
Consider another example shown below. Here we do not specify the number of rows/columns arguments of the **subplot** function. When these arguments are not specified the subplot function defaults to a single **axes** object contained within a **figure** object. We have used a **for loop** just as before, but this time to plot multiple line plots sharing the same x axis, on a single axes object.

```
listO_locations = ['WaggaWagga', 'Perth', 'Melbourne', 'Brisbane']
fig, axes = plt.subplots(figsize = (8, 6))

for idx, loc in enumerate(listO_locations):

    df_loc = df[df.location == loc]
    s_loc_maxtemp = df_loc.groupby('year').mean().maxtemp
    x, y = s_loc_maxtemp.index, s_loc_maxtemp.values
    axes.plot(x, y, label = loc)
    axes.legend()

fig.suptitle('YEAR  Vs MAX_TEMP')
figure.tight_layout()
```



The example shown below, has code that performs the same executions as the one above, except that this time we are plotting multiple **scatter** plots on a single **axes** object, instead of **line** plots.

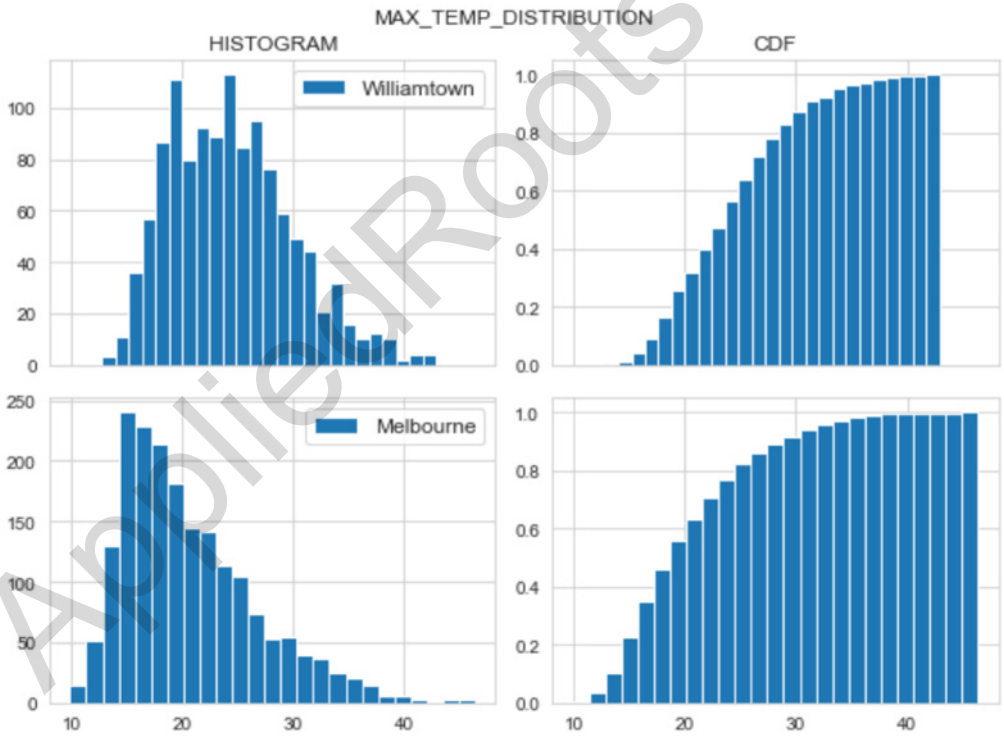
```
list0_locations = df.location.unique()[:4]
colors = ['red', 'blue', 'green', 'violet']

figure, axes = plt.subplots(figsize = (8, 6), sharex = True, sharey = False)
alpha = 1
for idx, location in enumerate(list0_locations):

    df1 = df[df.location == location]
    x1, y1 = df1.maxtemp, df1.humidity
    axes.scatter(x1, y1, alpha = alpha, label = location, color = colors[idx])
    alpha = alpha - 0.25

    axes.set_xlabel('TEMPERATURE')
    axes.set_ylabel('HUMIDITY')
    axes.legend(fontsize = 12)

figure.tight_layout()
```



Note how the **color** kwarg is used within the scatter method to specify the color of the plots. The desired colors are specified within the list named **colors** and the color **kwarg** is specified within the scatter method via iteration of this list. The **color** kwarg is available for all other plotting methods (plot, bar, hist etc). Also note how the value of the **alpha** kwarg is reduced by 0.25 during each iteration thus making every "next" plot be slightly more transparent than the previous one.

Shown below is a multi plot example, where each column contains the same plot type. The column having index 0 plots **histograms** and the column having index 1 plots **CDFs.** This is achieved by not flattening the **axes** array and using 2D indexes (instead of 1D indexes as in the previous example).

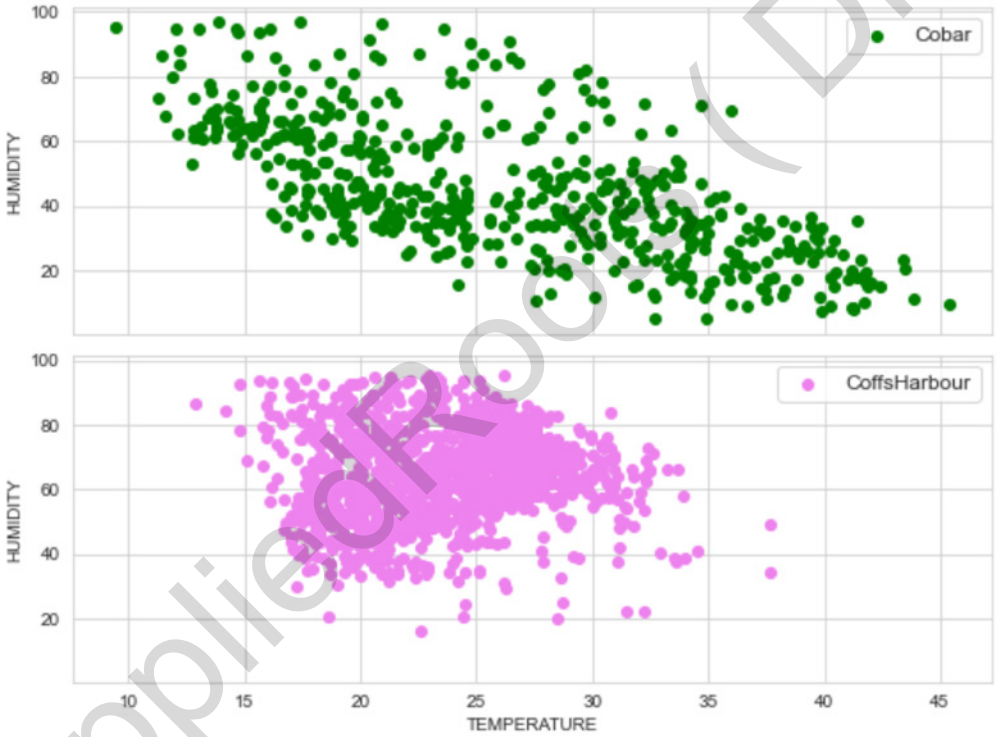
```
list0_locs = ['Williamtown', 'Melbourne']
fig, axes = plt.subplots(2, 2, figsize = (8, 6), sharex = True,
                                                  sharey = False)
for idx, location in enumerate(list0_locs):

    df1 = df[df.location == location]
    x = df1.maxtemp
    axes[idx, 0].hist(x, bins = 25, label = location)
    axes[idx, 1].hist(x, bins = 25, label = location,
                              cumulative = True,
                                 density = True)
    axes[idx, 0].legend(fontsize = 12)
    axes[idx, 0].legend(fontsize = 12)

axes[0, 0].set_title('HISTOGRAM')
axes[0, 1].set_title('CDF')
fig.suptitle('MAX_TEMP_DISTRIBUTION')
fig.tight_layout()
```



Note the **density** kwarg used within the second **hist** method. This kwarg normalizes the numerical values fed to the hist function (ie: scales the values to be between 0 & 1) by using a scaling computation similar to kernel density estimation (KDE).

The example shown below shows a multi plot containing just rows (ie: only one column).

```
listO_locations = df.location.unique()[:2]
colors = ['green', 'violet']

figure, axes = plt.subplots(2, 1, figsize = (8, 6), sharex = True, sharey = True)
axes, alpha = axes.flatten(), 1

for idx, location in enumerate(listO_locations):

    df1 = df[df.location == location]
    x1, y1 = df1.maxtemp, df1.humidity
    axes[idx].scatter(x1, y1, alpha = alpha, label = location, color = colors[idx])
    axes[idx].set_ylabel('HUMIDITY')
    axes[idx].legend(fontsize = 12)

axes[-1].set_xlabel('TEMPERATURE')
figure.tight_layout()
```



This concludes the eighth chapter eight chapter.