Python Learning

Loop and conditional statement in Python

1. If condition

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

2. Elif condition

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

3. Short if condition

```
if a > b: print("a is greater than b")
```

4. Else condition

The else keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

5. Short if else condition

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line

```
a = 2
b = 330
print("A") if a > b else print("B")
```

WHILE

While loop

With the `while` loop we can execute a set of statements as long as a condition is true.

```
i = 1
while i < 6:
  print(i)
  i += 1
```

While with else

With the `else` statement we can run a block of code once when the condition no longer is true:

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

For loop

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

For loop for string

Even strings are iterable objects, they contain a sequence of characters:

```
for x in "banana":
  print(x)
```

Else in for loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

LAMBDA function in python

Lamda is an anonymous function that can take any number of arguments but can have only one expression.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

Example

Add 10 to argument `a`, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

In the example above, the expression is composed of:

- **The keyword:** `lambda`
- **A bound variable:** `x`
- **A body:** `x`

Function in Python

- In Python a function is defined using the def keyword

1. **Function without return type**

```
def my_function():
  print("Hello from a function")
```

2. **Function with arguments**

```
def my_function(fname):
  print(fname + " Refsnes")
```

3. **Arbitrary number of argument in Python**

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

4. **Keyword (Key value pair) argument or key argument**

You can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

5. **Arbitrary keyword argument in Python**

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

6. Passing list as an argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

7. Method with return type

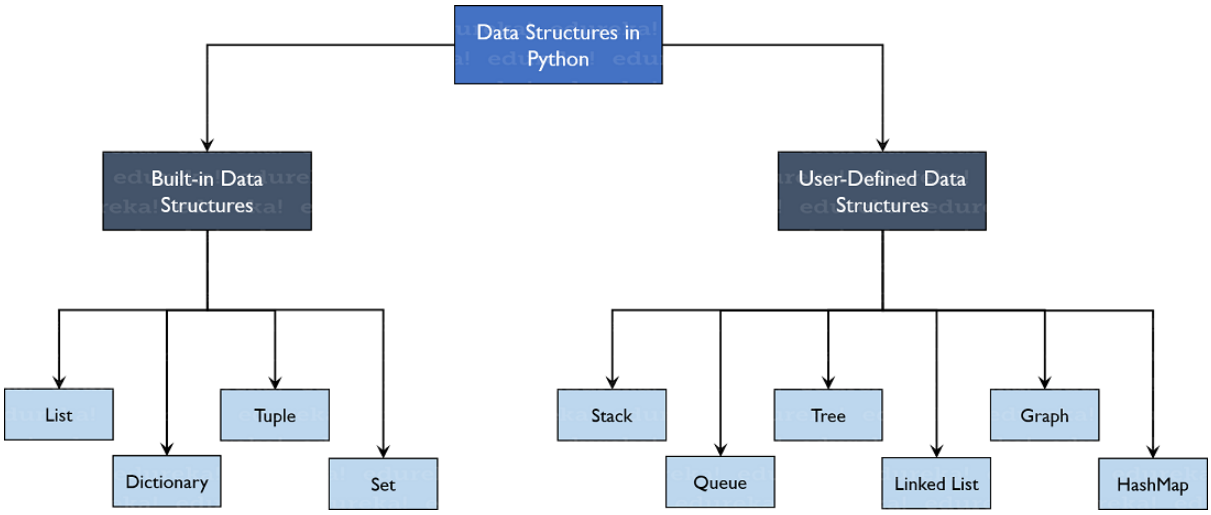
```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

8. The pass statement

`function` definitions cannot be empty, but if you for some reason have a `function` definition with no content, put in the `pass` statement to avoid getting an error.

```
def myfunction():
  pass
```

Data structure



1.1 Lists

Lists are used to store data of different data types in a sequential manner. There are addresses assigned to every element of the list, which is called as Index. The index value starts from 0 and goes on until the last element called the **positive index**. There is also **negative indexing** which starts from -1 enabling you to access elements from the last to first. Let us now understand lists better with the help of an example program.

1.1.1 Creating a list

To create a list, you use the square brackets and add elements into it accordingly. If you do not pass any elements inside the square brackets, you get an empty list as the output.

```
my_list = [] #create empty list
print(my_list)
my_list = [1, 2, 3, 'example', 3.132] #creating list with data
print(my_list)
```

Output:

[]

[1, 2, 3, ‘example’, 3.132]

1.1.2 Adding Elements

Adding the elements in the list can be achieved using the append(), extend() and insert() functions.

- The append() function adds all the elements passed to it as a single element.
- The extend() function adds the elements one-by-one into the list.
- The insert() function adds the element passed to the index value and increase the size of the list too.

```
my_list = [1, 2, 3]
print(my_list)
my_list.append([555, 12]) #add as a single element
print(my_list)
my_list.extend([234, 'more_example']) #add as different elements
print(my_list)
my_list.insert(1, 'insert_example') #add element i
print(my_list)
```

Output:
[1, 2, 3]
[1, 2, 3, [555, 12]]
[1, 2, 3, [555, 12], 234, ‘more_example’]
[1, ‘insert_example’, 2, 3, [555, 12], 234, ‘more_example’]

1.1.3 Deleting Elements

- To delete elements, use the *del* keyword which is built-in into Python but this does not return anything back to us.
- If you want the element back, you use the pop() function which takes the index value.
- To remove an element by its value, you use the remove() function.

Example:

```
        my_list = [1, 2, 3, 'example', 3.132, 10, 30]
          del my_list[5] #delete element at index 5
                        print(my_list)
    my_list.remove('example') #remove element with value
                        print(my_list)
          a = my_list.pop(1) #pop element from list
 print('Popped Element: ', a, ' List remaining: ', my_list)
              my_list.clear() #empty the list
                        print(my_list)
```

Output:
[1, 2, 3, ‘example’, 3.132, 30]
[1, 2, 3, 3.132, 30]
Popped Element: 2 List remaining: [1, 3, 3.132, 30]
[]

1.1.4 Accessing Elements

Accessing elements is the same as accessing Strings in Python. You pass the index values and hence can obtain the values as needed.

```
my_list = [1, 2, 3, 'example', 3.132, 10, 30]
for element in my_list: #access elements one by one
    print(element)
print(my_list) #access all elements
print(my_list[3]) #access index 3 element
print(my_list[0:2]) #access elements from 0 to 1 and exclude 2
print(my_list[::-1]) #access elements in reverse
```

Output:

1
2
3
example
3.132
10
30
[1, 2, 3, ‘example’, 3.132, 10, 30]
example
[1, 2]
[30, 10, 3.132, ‘example’, 3, 2, 1]

Other Functions

You have several other functions that can be used when working with lists.

- The len() function returns to us the length of the list.
- The index() function finds the index value of value passed where it has been encountered the first time.
- The count() function finds the count of the value passed to it.
- The sorted() and sort() functions do the same thing, that is to sort the values of the list. The sorted() has a return type whereas the sort() modifies the original list.

```
my_list = [1, 2, 3, 10, 30, 10]
print(len(my_list)) #find length of list
print(my_list.index(10)) #find index of element that occurs first
print(my_list.count(10)) #find count of the element
print(sorted(my_list)) #print sorted list but not change original
my_list.sort(reverse=True) #sort original list
print(my_list)
```

Output:

6
3
2
[1, 2, 3, 10, 10, 30]

```
[30, 10, 10, 3, 2, 1]
```

1.2 Dictionary

Dictionaries are used to store **key-value** pairs.

1.2.1 Creating a Dictionary

Dictionaries can be created using the flower braces or using the dict() function. You need to add the key-value pairs whenever you work with dictionaries.

```
my_dict = {} #empty dictionary
print(my_dict)
my_dict = {1: 'Python', 2: 'Java'} #dictionary with elements
print(my_dict)
```

Output:
{}
{1: ‘Python’, 2: ‘Java’}

1.2.2 Changing and Adding key, value pairs

To change the values of the dictionary, you need to do that using the keys. So, you firstly access the key and then change the value accordingly. To add values, you simply just add another key-value pair as shown below.

```
my_dict = {'First': 'Python', 'Second': 'Java'}
print(my_dict)
my_dict['Second'] = 'C++' #changing element
print(my_dict)
my_dict['Third'] = 'Ruby' #adding key-value pair
print(my_dict)
```

Output:
{‘First’: ‘Python’, ‘Second’: ‘Java’}
{‘First’: ‘Python’, ‘Second’: ‘C++’}
{‘First’: ‘Python’, ‘Second’: ‘C++’, ‘Third’: ‘Ruby’}

Deleting key, value pairs

- To delete the values, you use the pop() function which returns the value that has been deleted.
- To retrieve the key-value pair, you use the popitem() function which returns a tuple of the key and value.
- To clear the entire dictionary, you use the clear() function.

```
my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}
a = my_dict.pop('Third') #pop element
print('Value:', a)
print('Dictionary:', my_dict)
b = my_dict.popitem() #pop the key-value pair
print('Key, value pair:', b)
print('Dictionary', my_dict)
my_dict.clear() #empty dictionary
print('n', my_dict)
```

Output:

Value: Ruby
Dictionary: {‘First’: ‘Python’, ‘Second’: ‘Java’}

Key, value pair: (‘Second’, ‘Java’)
Dictionary {‘First’: ‘Python’}

{}

1.2.3 Accessing Elements

You can access elements using the keys only. You can use either the get() function or just pass the key values and you will be retrieving the values.

```
my_dict = {'First': 'Python', 'Second': 'Java'}
print(my_dict['First']) #access elements using keys
print(my_dict.get('Second'))
```

Output:
Python
Java

1.2.4 Other Functions

You have different functions which return to us the keys or the values of the key-value pair accordingly to the keys(), values(), items() functions accordingly.

```
my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}
print(my_dict.keys()) #get keys
print(my_dict.values()) #get values
print(my_dict.items()) #get key-value pairs
print(my_dict.get('First'))
```

Output:
dict_keys([‘First’, ‘Second’, ‘Third’])
dict_values([‘Python’, ‘Java’, ‘Ruby’])
dict_items([(‘First’, ‘Python’), (‘Second’, ‘Java’), (‘Third’, ‘Ruby’)])
Python

1.3 Tuple

Tuples are the same as lists are with the exception that the data once entered into the tuple cannot be changed no matter what. The only exception is when the data inside the tuple is mutable, only then the tuple data can be changed. The example program will help you understand better.

1.3.1 Creating a Tuple

You create a tuple using parenthesis or using the tuple() function.

```
my_tuple = (1, 2, 3) #create tuple
print(my_tuple)
```

Output:
(1, 2, 3)

1.3.2 Accessing Elements

Accessing elements is the same as it is for accessing values in lists.

```
my_tuple2 = (1, 2, 3, 'edureka') #access elements
for x in my_tuple2:
print(x)
print(my_tuple2)
print(my_tuple2[0])
print(my_tuple2[:])
print(my_tuple2[3][4])
```

Output:
1
2
3
edureka
(1, 2, 3, ‘edureka’)
1
(1, 2, 3, ‘edureka’)
e

1.3.3 Appending Elements

To append the values, you use the ‘+’ operator which will take another tuple to be appended to it.

```
my_tuple = (1, 2, 3)
my_tuple = my_tuple + (4, 5, 6) #add elements
print(my_tuple)
```

Output:
(1, 2, 3, 4, 5, 6)

1.3.4 Other Functions

These functions are the same as they are for lists.

```
my_tuple = (1, 2, 3, ['hindi', 'python'])
my_tuple[3][0] = 'english'
print(my_tuple)
print(my_tuple.count(2))
print(my_tuple.index(['english', 'python']))
```

Output:
(1, 2, 3, [‘english’, ‘python’])
1
3

1.4 Sets

Sets are a collection of unordered elements that are unique. Meaning that even if the data is repeated more than one time, it would be entered into the set only once. It resembles the sets that you have learnt in arithmetic. The operations also are the same as is with the arithmetic sets. An example program would help you understand better.

1.4.1 Creating a set

Sets are created using the flower braces but instead of adding key-value pairs, you just pass values to it.

```
my_set = {1, 2, 3, 4, 5, 5, 5} #create set
print(my_set)
```

Output:
{1, 2, 3, 4, 5}

1.4.2 Adding elements

To add elements, you use the add() function and pass the value to it.

```
my_set = {1, 2, 3}
my_set.add(4) #add element to set
print(my_set)
```

Output:
{1, 2, 3, 4}

1.4.3 Operations in sets

The different operations on set such as union, intersection and so on are shown below.

```
my_set = {1, 2, 3, 4}
my_set_2 = {3, 4, 5, 6}
print(my_set.union(my_set_2), '----------', my_set | my_set_2)
print(my_set.intersection(my_set_2), '----------', my_set &
my_set_2)
print(my_set.difference(my_set_2), '----------', my_set -
my_set_2)
print(my_set.symmetric_difference(my_set_2), '----------', my_set
^ my_set_2)
my_set.clear()
print(my_set)
```

- The union() function combines the data present in both sets.
- The intersection() function finds the data present in both sets only.
- The difference() function deletes the data present in both and outputs data present only in the set passed.
- The symmetric_difference() does the same as the difference() function but outputs the data which is remaining in both sets.

Output:
{1, 2, 3, 4, 5, 6} ———— {1, 2, 3, 4, 5, 6}
{3, 4} ———— {3, 4}
{1, 2} ———— {1, 2}
{1, 2, 5, 6} ———— {1, 2, 5, 6}
set()

Now that you have understood the built-in Data Structures, let's get started with the user-defined Data Structures. User-defined Data Structures, the name itself suggests that users define how the Data Structure would work and define functions in it. This gives the user whole control over how the data needs to be saved, manipulated and so forth.

Let us move ahead and study the most prominent Data Structures in most of the programming languages.

2 User-Defined Data Structures

2.1 Arrays vs. Lists

Arrays and lists are the same structure with one difference. Lists allow heterogeneous data element storage whereas Arrays allow only homogenous elements to be stored within them.

2.1.1 What is an Array in Python?

An array is basically a *data structure* which can hold more than one value at a time. It is a collection or ordered series of elements of the same type.

Example:

```
1                    a=arr.array('d',[1.2,1.3,2.3])
```

We can loop through the array items easily and fetch the required values by just specifying the index number. Arrays are mutable(changeable) as well, therefore, you can perform various manipulations as required.

Now, there is always a question that comes up to our mind –

2.1.2. Is Python list same as an Array?

Python Arrays and lists are store values in a similar way. But there is a key difference between the two i.e the values that they store. A list can store any type of values such as intergers, strings, etc. An arrays, on the other hand, stores single data type values. Therefore, you can have an array of integers, an array of strings, etc.

Python also provide Numpy Arrays which are a grid of values used in Data Science. You can look into Numpy Arrays vs Lists to know more.

2.1.3 Creating an Array in Python:

Arrays in Python can be created after importing the array module as follows –

→ **import array as arr**

The array**(data type, value list)** function takes two parameters, the first being the data type of the value to be stored and the second is the value list. The data type can be anything such as int, float, double, etc. Please make a note that arr is the alias name and is for ease of use. You can import without alias as well.

There is another way to import the array module which is –

→ from array import *

This means you want to import all functions from the array module.

The following syntax is used to create an array.

Syntax:

```
1          a=arr.array(data type,value list)
                #when you import using arr alias
```

OR

```
1          a=array(data type,value list)
                #when you import using *
```

Example: a=arr.array(‘d’ , [1.1 , 2.1 ,3.1])

Here, the first parameter is ‘d’ which is a data type i.e. float and the values are specified as the next parameter.

Note:

All values specified are of the type float. We cannot specify the values of different data types to a single array.

The following table shows you the various data types and their codes.

Type code	Python Data Type	Byte size
i	int	2
I	int	2
u	unicode character	2
h	int	2
H	int	2
l	int	4
L	int	4
f	float	4
d	float	8

2.1.4 Accessing array elements in Python :

To access array elements, you need to specify the index values. Indexing starts at 0 and not from 1. Hence, the index number is always 1 less than the length of the array.

Syntax:

Array_name[index value]

Example:

```
1          a=arr.array( 'd', [1.1 , 2.1 ,3.1] )
2                  a[1]
```

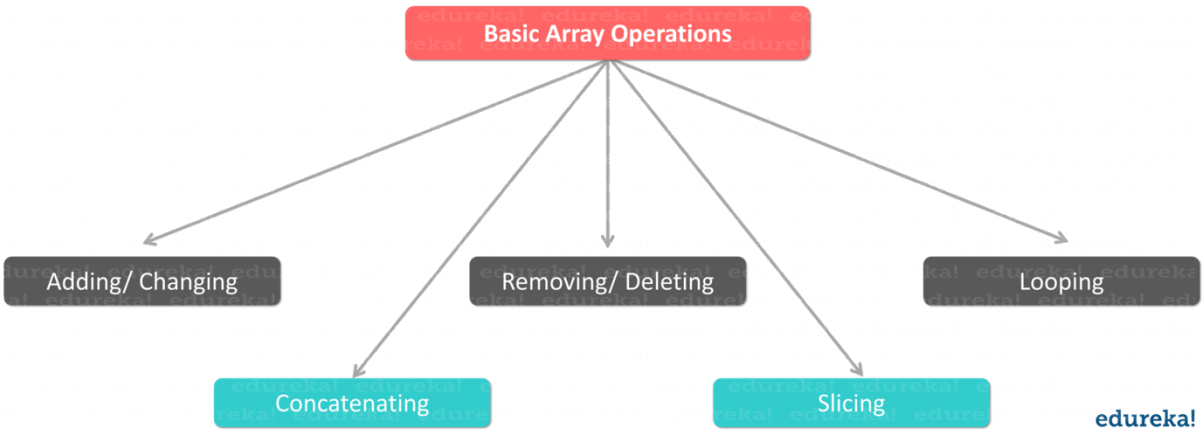
Output –

The output returned is the value, present at the second place in our array which is 2.1.

Let us have a look at some of the basic array operations now.

2.1.5 Basic array operations :

There are many operations that can be performed on arrays which are as follows –



2.1.6 Finding the Length of an Array

Length of an array is the number of elements that are actually present in an array. You can make use of **len()** function to achieve this. The len() function returns an integer value that is equal to the number of elements present in that array.

Syntax:

→ len(array_name)

Example:

```
a=arr.array('d', [1.1 , 2.1 ,3.1] )
len(a)
```

Output – 3

This returns a value of 3 which is equal to the number of array elements.

2.1.7 Adding/ Changing elements of an Array:

We can add value to an array by using the **append()**, **extend()** and the **insert (i,x)** functions.

The append() function is used when we need to add a single element at the end of the array.

Example:

```
a=arr.array('d', [1.1 , 2.1 ,3.1] )
a.append(3.4)
print(a)
```

Output –

array(‘d’, [1.1, 2.1, 3.1, 3.4])

The resultant array is the actual array with the new value added at the end of it. To add more than one element, you can use the extend() function. This function takes a list of elements as its parameter. The contents of this list are the elements to be added to the array.

Example:

```
a=arr.array('d', [1.1 , 2.1 ,3.1] )
a.extend([4.5,6.3,6.8])
print(a)
```

Output –

array(‘d’, [1.1, 2.1, 3.1, 4.5, 6.3, 6.8])

The resulting array will contain all the 3 new elements added to the end of the array.

However, when you need to add a specific element at a particular position in the array, the insert(i,x) function can be used. This function inserts the element at the respective index in the array. It takes 2 parameters where the first parameter is the index where the element needs to be inserted and the second is the value.

Example:

```
a=arr.array('d', [1.1 , 2.1 ,3.1] )
a.insert(2,3.8)
print(a)
```

Output –

array(‘d’, [1.1, 2.1, 3.8, 3.1])

The resulting array contains the value 3.8 at the 3rd position in the array.

Arrays can be merged as well by performing array concatenation.

2.1.8 Array Concatenation :

Any two arrays can be concatenated using the + symbol.

Example:

```
a=arr.array('d',[1.1 , 2.1 ,3.1,2.6,7.8])
b=arr.array('d',[3.7,8.6])
c=arr.array('d')
c=a+b
print("Array c = ",c)
```

Output –

Array c= array(‘d’, [1.1, 2.1, 3.1, 2.6, 7.8, 3.7, 8.6])

The resulting array c contains concatenated elements of arrays a and b.

Now, let us see how you can remove or delete items from an array.

2.1.9 Removing/ Deleting elements of an array:

Array elements can be removed using **pop()** or **remove()** method. The difference between these two functions is that the former returns the deleted value whereas the latter does not.

The pop() function takes either no parameter or the index value as its parameter. When no parameter is given, this function pops() the last element and returns it. When you explicitly supply the index value, the pop() function pops the required elements and returns it.

Example:

```
a=arr.array('d', [1.1, 2.2, 3.8, 3.1, 3.7, 1.2, 4.6])
print(a.pop())
print(a.pop(3))
```

Output –

```
4.6
3.1
```

The first pop() function removes the last value 4.6 and returns the same while the second one pops the value at the 4th position which is 3.1 and returns the same.

The remove() function, on the other hand, is used to remove the value where we do not need the removed value to be returned. This function takes the element value itself as the parameter. If you give the index value in the parameter slot, it will throw an error.

Example:

```
a=arr.array('d',[1.1 , 2.1 ,3.1])
a.remove(1.1)
print(a)
```

Output –

array(‘d’, [2.1,3.1])

The output is an array containing all elements except 1.1.

When you want a specific range of values from an array, you can slice the array to return the same, as follows.

2.1.10 Slicing an array :

An array can be sliced using the : symbol. This returns a range of elements that we have specified by the index numbers.

Example:

```
a=arr.array('d',[1.1 , 2.1 ,3.1,2.6,7.8])
print(a[0:3])
```

Output –

array(‘d’, [1.1, 2.1, 3.1])

The result will be elements present at 1st, 2nd and 3rd position in the array.

2.1.11 Looping through an array:

Using the for loop, we can loop through an array.

Example:

```
a=arr.array('d', [1.1, 2.2, 3.8, 3.1, 3.7, 1.2, 4.6])
print("All values")
    for x in a:
     print(x)
print("specific values")
    for x in a[1:3]:
     print(x)
```

Output –

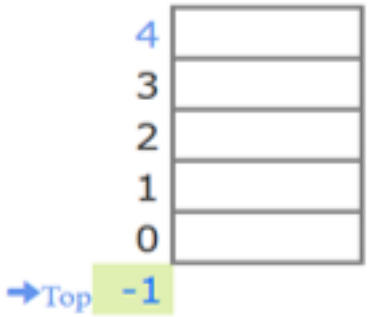
All values

1.1
2.2
3.8
3.1
3.7
1.2
4.6
specific values
2.2
3.8

The above output shows the result using for loop. When we use for loop without any specific parameters, the result contains all the elements of the array given one at a time. In the second for loop, the result contains only the elements that are specified using the index values. Please note that the result does not contain the value at index number 3.

2.2 Stack

Stacks are linear Data Structures which are based on the principle of Last-In-First-Out (LIFO) where data which is entered last will be the first to get accessed. It is built using the array structure and has operations namely, pushing (adding) elements, popping (deleting) elements and accessing elements only from one point in the stack called as the TOP. This TOP is the pointer to the current position of the stack. Stacks are prominently used in applications such as Recursive Programming, reversing words, undo mechanisms in word editors and so forth.



2.2.1 What is Stack in Data Structures?

Data structures are the key to organize storage in computers so that we can efficiently access and edit data. *Stacks* is one of the earliest data structures defined in computer science. In simple words, Stack is a linear collection of items. It is a collection of objects that supports fast last-in, first-out (LIFO) semantics for insertion and deletion. It is an array or list structure of function calls and parameters used in modern computer programming and CPU architecture. Similar to a stack of plates at a restaurant, elements in a stack are added or removed from the top of the stack, in a “last in, first out” order. Unlike lists or arrays, random access is not allowed for the objects contained in the stack.

There are two types of operations in Stack-

- **Push**– To add data into the stack.
- **Pop**– To remove data from the stack.

Stacks are simple to learn and easy to implement, they are extensively incorporated in many software for carrying out various tasks. They can be implemented with an Array or Linked List. We’ll be relying on the List data structure here.

2.1.2 Why and When do we use Stack?

Stacks are simple data structures that allow us to store and retrieve data sequentially.

Talking about performance, a proper stack implementation is expected to take *O(1)* time for insert and delete operations.

To understand Stack at the ground level, think about a pile of books. You add a book at the top of the stack, so the first one to be picked up will be the last one that was added to the stack.

There are many real-world use cases for stacks, understanding them allows us to solve many data storage problems in an easy and effective way.

Imagine you're a developer and you are working on a brand new word processor. You need to create an undo feature – allowing users to backtrack their actions until the beginning of the session. A stack is an ideal fit for this scenario. We can record every action of the user by pushing it to the stack. When the user wants to undo an action they can pop accordingly from the stack.

How can we implement a stack in Python?

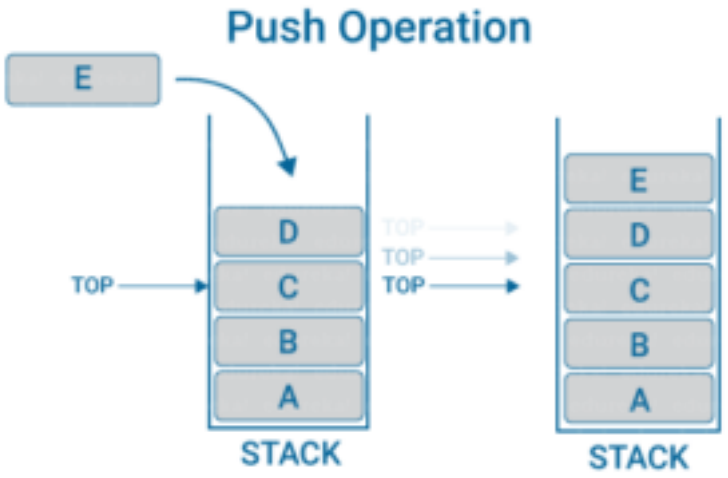
In Python, we can implement python stacks by:

1. Using the built-in List data structure. Python's built-in List data structure comes with methods to simulate both *stack* and *queue* operations.
2. Using the deque library which efficiently provides stack and queue operations in one object.
3. Using the queue.LifoQueue Class.

As mentioned earlier, we can add items to a stack using the "PUSH" operation and remove items using the "POP" operation.

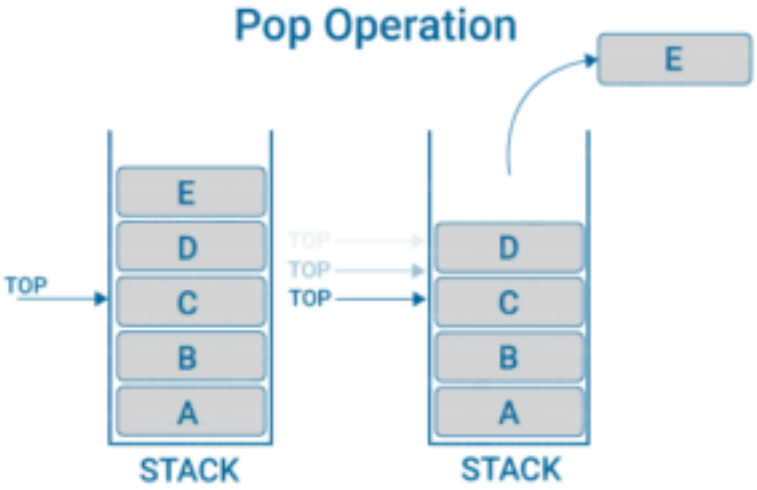
2.2.3 PUSH Operation

Push – adds an element at the top of the stack. Refer the below image for more understanding:



2.2.4 POP Operation

Pop – removes an element from the top of the stack.




```
class Stack
def __init__(self):
self.items=[]
def is_empty(self):
return self.items==[]
def push(self, data):
self.items.append(data)
def pop(self):
return self.items.pop()
s= Stack()
while True:
print('push<value>')
print('pop')
print('quit')
do= input('What would you like to do?').split()
operation= do[0].strip().lower()
if operation== 'push':
```

```
                s.push(int(do[1]))
            elif operation== 'pop':
                if s.is_empty():
                print('Stack is empty')
                else:
            print('Popped value:', s.pop())
            elif operation=='quit':
                break
```

Here is a simple program to illustrating Stack in Python-

More about Usage of Stacks in Python and related programs

Stacks provide a wide range of uses in algorithms, for eg, in language parsing and run-time memory management ("call stack"). A short and useful algorithm using a stack is a depth-first search (DFS) on a tree or graph data structure. Python plays with several stack implementations and each has slightly different characteristics. Let's take a brief look at them:

The list Built-in

Python's built-in list type makes a decent stack data structure as it supports push and pop operations in amortized *O(1)* time.

Python's lists are implemented as dynamic arrays internally which means they occasionally need to resize the storage space for elements stored in them whenever they are added or removed. The storage space is allocated more than required, so that not every push or pop requires resizing and you get an amortized *O(1)* time complexity for these operations.

Although this makes their performance less consistent than the stable *O(1)* inserts and deletes provided by a linked list-based implementation. On the other hand, lists provide fast *O(1)* time random access to elements on the stack which can be an added benefit.

Here's an important performance caveat while using lists as stacks:

To get the amortized *O(1)* performance for insertion and deletion, new is added to the end of the list with the append() method and are removed from the end using pop(). Stacks based on Pythonlists extend to the right and shrink to the left.

Adding and removing from the front takes much more time (*O(n)* time), as the existing elements have to be shifted around to make room for the new element to be added.

Using a Python list as a stack (LIFO):

```
s = []

s.append('eat')
s.append('sleep')
s.append('code')

>>> s
['eat', 'sleep', 'code']

>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'

>>> s.pop()
IndexError: "pop from empty list"
```

2.2.5 The collections.deque Class

The deque class implements a double-ended queue which supports addition and removal of elements from either end in *O(1)* time (non-amortized).

Because deques support adding and removing elements from both ends equally well, they can serve both as queues and as stacks.

Python's deque objects are implemented as doubly-linked lists which gives them proper and consistent performance insertion and deletion of elements, but poor *O(n)* performance as they randomly access elements in the middle of the stack.

collections.deque is a favourable choice if you're looking for a stack data structure in Python's standard library with the performance characteristics of a linked-list implementation.

Using collections.deque as a stack (LIFO):

```
from collections import deque
q = deque()

q.append('eat')
q.append('sleep')
q.append('code')

>>> q
deque(['eat', 'sleep', 'code'])

>>> q.pop()
'code'
>>> q.pop()
'sleep'
>>> q.pop()
```

```
'eat'

>>> q.pop()
IndexError: "pop from an empty deque"
```

2.2.6 The queue.LifoQueue Class

This stack implementation in the Python standard library is synchronized and provides locking semantics to support multiple concurrent producers and consumers.

The queue module contains several other classes implementing multi-producer, multi-consumer queues that are useful for parallel computing.

Depending on your use case the locking semantics might be helpful, or just incur unneeded overhead. In this case you'd be better off with using a list or a deque as a general purpose stack.

Using queue.LifoQueue as a stack:

```
from queue import LifoQueue
s = LifoQueue()

s.put('eat')
s.put('sleep')
s.put('code')

>>> s
<queue.LifoQueue object at 0x108298dd8>

>>> s.get()
'code'
>>> s.get()
'sleep'
>>> s.get()
'eat'

>>> s.get_nowait()
queue.Empty

>>> s.get()
# Blocks / waits forever...
```

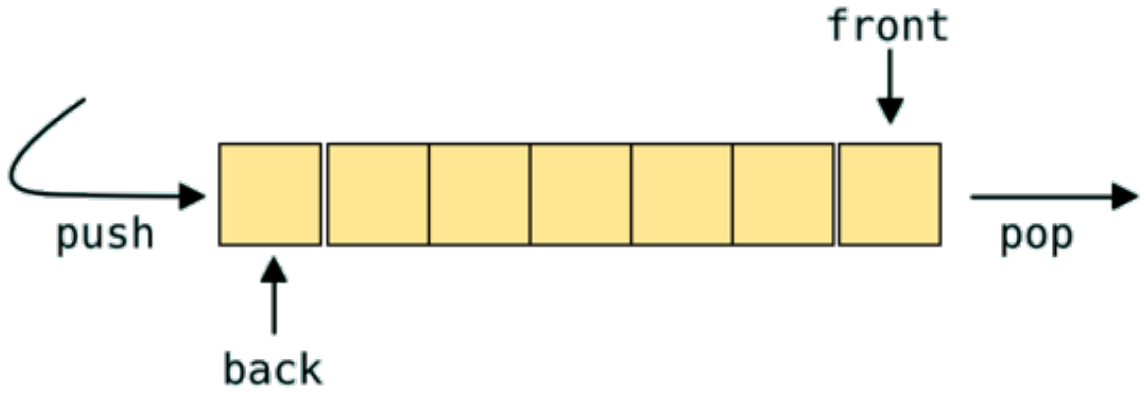
If you have continued this far, you must now be in a position to use stacks in Python, I hope this blog helped you go through the different implementation methods of a stack in Python.

So this concludes our "stack in python" article. I hope you enjoyed reading this blog and found it informative. By now, you must have acquired a sound understanding of

what a python stack is and how it is used. Now go ahead and practice all the examples.

2.3 Queue

A queue is also a linear data structure which is based on the principle of First-In-First-Out (FIFO) where the data entered first will be accessed first. It is built using the array structure and has operations which can be performed from both ends of the Queue, that is, head-tail or front-back. Operations such as adding and deleting elements are called En-Queue and De-Queue and accessing the elements can be performed. Queues are used as Network Buffers for traffic congestion management, used in Operating Systems for Job Scheduling and many more.



2.3.1 Creating a Queue Data Structure

Apart from the complementary operations, I may say that the main Operations possible on the Queue are:

1. **En-queue** or add an element to the end of the queue.

2. **De-queue** or remove an element from the front of the queue

Now, let's start via creating class Queue in Python:

```
class Queue:
    def __init__(self,max_size):
        self.__max_size=max_size
        self.__elements=[None]*self.__max_size
        self.__rear=-1
        self.__front=0
```

- **max_size** is the maximum number of elements expected in the queue.
- Elements of the queue are stored in the python list
- rear indicates the index position of the last element in the queue.
- The rear is initially taken to be -1 because the queue is empty

- Front indicates the position of the first element in the queue.
- The front is taken to be 0 initially because it will always point to the first element of

```
                                #returns max_size of queue
                                def get_max_size(self):
                                    return self.__max_size

    #returns bool value whether queue is full or not, True if full and False otherwise
                                def is_full(self):
                                return self.__rear==self.__max_size-1

                            #inserts/enqueue data to the queue if it is not full
                                def enqueue(self,data):
                                    if(self.is_full()):
                                print("Queue is full. No enqueue possible")
                                    else:
                                        self.__rear+=1
                                self.__elements[self.__rear]=data

                            #display all the content of the queue
                                def display(self):
                                for i in range(0,self.__rear+1):
                                    print(self.__elements[i])

#You can use the below __str__() to print the elements of the DS object while debugging
                                def __str__(self):
                                        msg=[]
                                    index=self.__front
                                while(index<=self.__rear):
                                msg.append((str)(self.__elements[index]))
                                        index+=1
                                    msg=" ".join(msg)
                                msg="Queue data(Front to Rear): "+msg
                                        return msg
```

the queue

2.3.2 Enqueue

Now, when you are trying to enqueue elements to the Queue, you have to remember the following points:

- Whether there is space left in the queue or not i.e. if rear equals max_size -1
- The rear will point to the last element inserted in the queue.

So, what will be the algorithm??

Now, When you execute the following:

queue1=Queue(5)

#Enqueue all the required element(s).

queue1.enqueue(“A”)

queue1.enqueue(“B”)

queue1.enqueue(“C”)

queue1.enqueue(“D”)

queue1.enqueue(“E”)

queue1.display()

queue1.enqueue(“F”)

print(queue1)

Output:

A

B

C

D

E

Queue is full. No enqueue possible

Queue data(Front to Rear): A B C D E

2.3.3 De-Queue

Now, as you have inserted/enqueued the elements into the queue, you want to dequeue/delete them from front, so you need to take care of following:

- There are elements in the queue i.e. rear should not be equal to -1.
- Secondly, you need to remember that as elements are deleted from the front so, after deleting front should be incremented to point next front.
- The front should not point end of queue i.e. equal to max_size.

So, what will be the algorithm??

```
1
2
3                    #function to check if queue is empty or not
4                                 def is_empty(self):
5                   if(self.__rear==-1 or self.__front==self.__max_size):
6                                              return True
7                                          else: return False
8
9                  #function to deque an element and return it
10                                  def dequeue(self):
11                                 if(self.is_empty()):
12                              print("queue is already empty")
13                                             else:
14                           data=self.__elements[self.__front]
15                                     self.__front+=1
16                                        return data
17
18   #function to display elements from front to rear if queue is not empty
19                                  def display(self):
20                                  if(not self.is_empty()):
21                               for i in range(self.__front,self.__rear+1):
22                               print(self.__elements[i])
                                           else:
                                        print("empty queue")
```

Now when you execute the following :

queue1=Queue(5)

#Enqueue all the required element(s)

queue1.enqueue(“A”)

queue1.enqueue(“B”)

queue1.enqueue(“C”)

queue1.enqueue(“D”)

queue1.enqueue(“E”)

print(queue1)

#Dequeue all the required element(s)

print(“Dequeued : “, queue1.dequeue())

print(“Dequeued : “, queue1.dequeue())

```
print(“Dequeued : “, queue1.dequeue())

print(“Dequeued : “, queue1.dequeue())

print(“Dequeued : “, queue1.dequeue())

print(“Dequeued : “, queue1.dequeue())

#Display all the elements in the queue.

queue1.display()
```

Output:

```
Queue data(Front to Rear): A B C D E

Dequeued :  A

Dequeued :  B

Dequeued :  C

Dequeued :  D

Dequeued :  E

queue is already empty

Dequeued :  None

empty queue
```

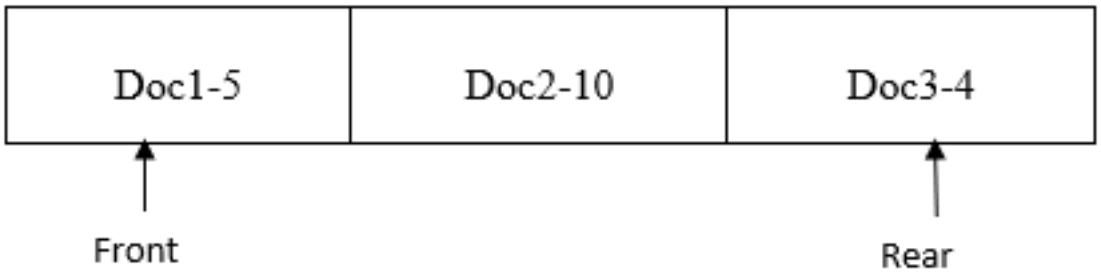
Applications of Queue

As of now, you have already understood the basics of queue. Now to dive deeper we will look into some of its applications.

- **Example 1:**

Print Queue in Windows uses a queue to store all the active and pending print jobs. When we want to print documents, we issue print commands one after the other. Based on the print commands, the documents will get lined up in the print queue. When the printer is ready, the document will be sent in first in first out order to get it printed.

Suppose you have issued print commands for 3 documents in the order doc1, followed by doc2 and doc3.
The print queue will be populated as shown below:



doc-n, where the doc is the document sent for printing and n, is the number of pages in the document. For example, doc2-10 means doc2 is to be printed and it has 10 pages. Here is a code that simulates print queue operation. Go through the code and observe how the queue is used in this implementation.

```
class Queue:
    def __init__(self,max_size):
        self.__max_size=max_size
        self.__elements=[None]*self.__max_size
        self.__rear=-1
        self.__front=0

    def is_full(self):
        if(self.__rear==self.__max_size-1):
            return True
        return False

    def is_empty(self):
        if(self.__front>self.__rear):
            return True
        return False

    def enqueue(self,data):
        if(self.is_full()):
            print("Queue is full!!!")
        else:
            self.__rear+=1
            self.__elements[self.__rear]=data

    def dequeue(self):
        if(self.is_empty()):
            print("Queue is empty!!!")
        else:
            data=self.__elements[self.__front]
            self.__front+=1
            return data

    def display(self):
        for index in range(self.__front, self.__rear+1):
            print(self.__elements[index])
```

```
    def get_max_size(self):
        return self.__max_size

    #You can use the below __str__() to print the elements of the DS object whi
    def __str__(self):
        msg=[]
        index=self.__front
        while(index<=self.__rear):
            msg.append((str)(self.__elements[index]))
            index+=1
        msg=" ".join(msg)
        msg="Queue data(Front to Rear): "+msg
        return msg

#function that enqueue are the documents to be printed in Queue named p
def send_for_print(doc):
    global print_queue
    if(print_queue.is_full()):
        print("Queue is full")
    else:
        print_queue.enqueue(doc)
    print(doc,"sent for printing")

#function that prints the document if number of pages of document is less than #total r
def start_printing():
    global print_queue
    while(not print_queue.is_empty()):
        #here we dequeue the Queue and take the coument that was input first f
        doc=print_queue.dequeue()
        global pages_in_printer
        #the aim of this for loop is to find number of pages of the of document which is do
        for i in range(0,len(doc)):
            if(doc[i]=="-"):
                no_of_pages=int(doc[i+1:])
                break
        if(no_of_pages<=pages_in_printer):
            print(doc,"printed")
            pages_in_printer-=no_of_pages
            print("Remaining no. of pages in printer:", pages_in_pr
        else:
            print("Couldn't print",doc[:i],". Not enough pages in the p

pages_in_printer=12
print_queue=Queue(10)
send_for_print("doc1-5")
send_for_print("doc2-3")
send_for_print("doc3-6")
start_printing()
```

81
82
83
84

Output:

doc1-5 sent for printing

doc2-3 sent for printing

doc3-6 sent for printing

doc1-5 printed

Remaining no. of pages in printer: 7

doc2-3 printed

Remaining no. of pages in printer: 4

Couldn’t print doc3 . Not enough pages in the printer

- **Example 2:**

Let’s try to understand another example which uses Queue data structure. Can you try to understand the code and tell what the following function does?

```
1. def fun(n):
2. aqueue = Queue(100)
3. for num in range(1, n+1):
4. enqueue(num)
5. result=1
6. while (not(aqueue.is_empty())):
7. num = aqueue.dequeue()
8. result*=num
9. print(result)
```

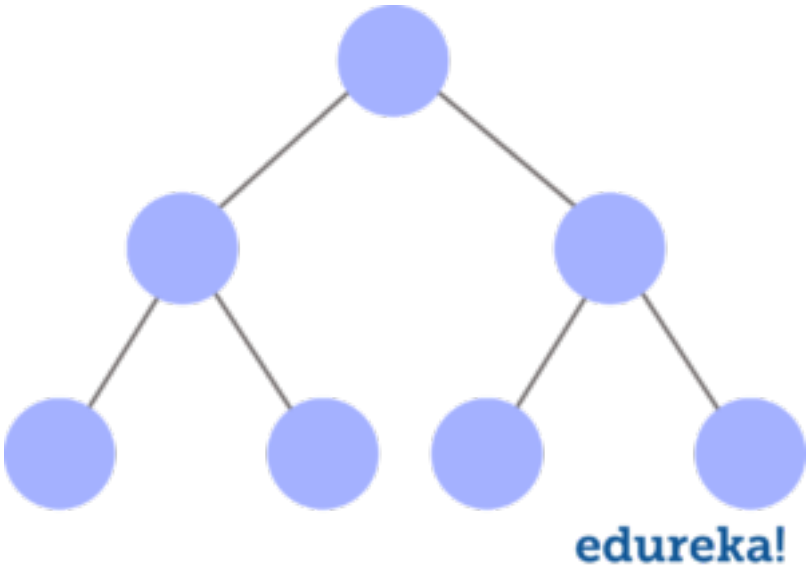
When function fun() is invoked by passing n,

- lines 2 to 4 en-queues the elements from 1 to n
- lines 5 to 8 finds the product of those elements by de-queuing it one by one

With this, we come to an end of this Queue Data Structure article. If you successfully understood and ran the codes by yourselves you are no longer a newbie to Queue Data Structure.

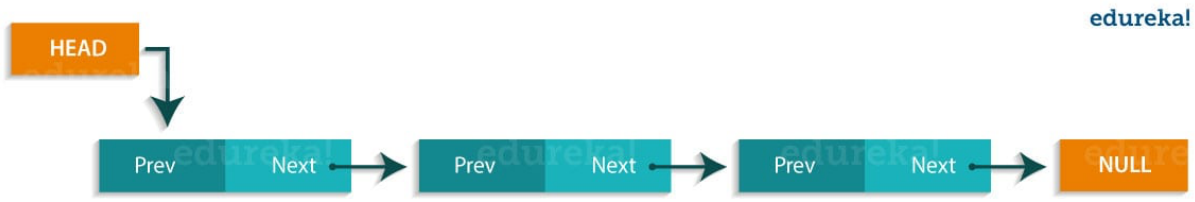
Tree

Trees are non-linear Data Structures which have root and nodes. The root is the node from where the data originates and the nodes are the other data points that are available to us. The node that precedes is the parent and the node after is called the child. There are levels a tree has to show the depth of information. The last nodes are called the leaves. Trees create a hierarchy which can be used in a lot of real-world applications such as the HTML pages use trees to distinguish which tag comes under which block. It is also efficient in searching purposes and much more.



Linked List

Linked lists are linear Data Structures which are not stored consequently but are linked with each other using pointers. The node of a linked list is composed of data and a pointer called next. These structures are most widely used in image viewing applications, music player applications and so forth.



Graph

Graphs are used to store data collection of points called vertices (nodes) and edges (edges). Graphs can be called as the most accurate representation of a real-world map. They are used to find the various cost-to-distance between the various data

points called as the nodes and hence find the least path. Many applications such as Google Maps, Uber, and many more use Graphs to find the least distance and increase profits in the best ways.

4. What is the use of self in Python?

4.1 What is the use of Self in Python?

The self is used to represent the instance of the class. With this keyword, you can access the attributes and methods of the class in python. It binds the attributes with the given arguments. The reason why we use self is that Python does not use the '@' syntax to refer to instance attributes. In Python, we have methods that make the instance to be passed automatically, but not received automatically.

Example:

```
class food():

    # init method or constructor
    def __init__(self, fruit, color):
        self.fruit = fruit
        self.color = color

    def show(self):
        print("fruit is", self.fruit)
        print("color is", self.color )

apple = food("apple", "red")
grapes = food("grapes", "green")

apple.show()
grapes.show()
```

4.2 Python Class self Constructor

self is also used to refer to a variable field within the class. Let's take an example and see how it works:

```
class Person:

# name made in constructor
def __init__(self, John):
self.name = John

def get_person_name(self):
return self.name
```

In the above example, self refers to the name variable of the entire Person class. Here, if we have a variable within a method, self will not work. That variable is simply existent only while that method is running and hence, is local to that method. For defining global fields or the variables of the complete class, we need to define them outside the class methods.

Is self a Keyword?

self is used in different places and often thought to be a keyword. But unlike in C++, self is not a keyword in Python.

self is a parameter in function and the user can use a different parameter name in place of it. Although it is advisable to use self because it increases the readability of code.

Example:

```
class this_is_class:

def show(in_place_of_self):
print("It is not a keyword "
"and you can use a different keyword")

object = this_is_class()
object.show()
```

Output:

```
It is not a keyword and you can use a different keyword
```

With this, we have come to the end of our article. I hope you understood the use of self and how it works in Python.

4.3 What is the Main Function in Python and how to use it?

In most programming languages, there is a special function which is executed automatically every time the program is run. This is nothing but the main function, or

main() as it is usually denoted. It essentially serves as a starting point for the execution of a program.

In Python, it is not necessary to define the main function every time you write a program. This is because the Python interpreter executes from the top of the file unless a specific function is defined. Hence, having a defined starting point for the execution of your Python program is useful to better understand how your program works.

A Basic Python main()

In most Python programs/scripts, you might see a function definition, followed by a conditional statement that looks like the example shown below:

```
def main():
    print("Hello, World!")
    if __name__== "__main__" :
main()
```

Does Python need a Main Function?

It is not a compulsion to have a a Main Function in Python, however, In the above example, you can see, there is a function called 'main()'. This is followed by a conditional 'if' statement that checks the value of **__name__**, and compares it to the string "**__main__**". On evaluation to True, it executes main().

And on execution, it prints "Hello, World!".

This kind of code pattern is very common when you are dealing with files that are to be executed as Python scripts, and/or to be imported in other modules.

Let's understand how this code executes. Before that, it's very necessary to understand that the Python interpreter sets **__name__** depending on the way how the code is executed. So, let's learn about the execution modes in Python

Python Execution Modes

There are two main ways by which you can tell the Python interpreter to execute the code:

- The most common way is to execute the file as a Python Script.
- By importing the necessary code from one Python file to another.

Whatever the mode of execution you choose, Python defines a special variable called **__name__**, that contains a string. And as I said before, the value of this string depends on how the code is being executed.

Sometimes, when you are importing from a module, you would like to know whether a particular module's function is being used as an import, or if you are just using the original .py (Python script) file of that module.

To help with this, Python has a special built-in variable, called **__name__**. This variable gets assigned the string "**__main__**" depending on how you are running or executing the script.

What is __main__ in Python?

Python Main Function is the beginning of any Python program. When we run a program, the interpreter runs the code sequentially and will not run the main function if imported as a module, but the Main Function gets executed only when it is run as a Python program.

So, if you are running the script directly, Python is going to assign "**__main__**" to **__name__**, i.e., **__name__**= "__main__". (This happens in the background).

As a result, you end up writing the conditional if statement as follows:

```
if __name__ == "__main__" :
        Logic Statements
```

It is important to understand that, if you are running something directly on the Python shell or terminal, this conditional statement, by default, happens to be True.

As a result, programmers write all the necessary functional definitions on the top, and finally write this statement at the end, to organize the code.

In short, **__name__** variable helps you to check if the file is being run directly or if it has been imported.

4.4 Try/Except in Python

The Try method is used in Error and Exception Handling. There are two kinds of errors :

- **Syntax Error**: It is also known as Parsing Error. This occurs when the Python parser is unable to understand a line of code.
- **Exception Error**: These errors are detected during execution.

Now, in these situations, we need to handle these errors within our code in Python. That is where try-except in python comes handy.

Syntax

```
try:
// Code
except:
// Code
```

Example

```
try:
print(x)
except:
print("An exception occurred")
```

4.4.1 Custom Exception Handling in Python

In Python, users can define custom exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from the built-in `Exception` class. Most of the built-in exceptions are also derived from this class.

```
>>> class CustomError(Exception):
...     pass
...
>>> raise CustomError
Traceback (most recent call last):
...
__main__.CustomError

>>> raise CustomError("An error occurred")
Traceback (most recent call last):
...
__main__.CustomError: An error occurred
```

Here, we have created a user-defined exception called `CustomError` which inherits from the `Exception` class. This new exception, like other exceptions, can be raised using the `raise`statement with an optional error message.

4.5 Comments

4.5.1 Single line comment

```
#This is a comment
print("Hello, World!")
```

4.5.2 Multi-line comment

```
"""
If I really hate pressing `enter` and
typing all those hash marks, I could
just do this instead
"""
```

4.6 Python Enhancement Proposal

1. Indentation

When programming in Python, indentation is something that you will definitely use. However, you should be careful with it, as it can lead to syntax errors. The recommendation is therefore to use 4 spaces for indentation. For example, this statement uses 4 spaces of indentation:

```
if True:
    print("If works")
```

And also this `for` loop with `print` statement is indented with 4 spaces:

```
for element in range(0, 5):
    print(element)
```

When you write a big expression, it is best to keep the expression vertically aligned. When you do this, you'll create a "hanging indent".

Here are some examples of the hanging indent in big expressions, which show some variations of how you can use it:

```
1. value = square_of_numbers(num1, num2,
2.                           num3, num4)
```

```
1. def square_of_number(
2.       num1, num2, num3,
3.       num4):
4.  return num1**2, num2**2, num3**2, num4**2
```

```
5. value = square_of_numbers(
6.                 num1, num2,
7.                 num3, num4)
```

```
8. list_of_people = [
9.  "Rama",
10.  "John",
11.  "Shiva"
12. ]


13. dict_of_people_ages = {
14.  "ram": 25,
15.  "john": 29,
16.  "shiva": 26
17. }
```

- 4 Spaces, although this is up to you on continuation lines, as long as there is some form of indentation.
- You can line up wrapped elements in parentheses vertically or with a hanging indent.

```
# 4 space indent

def hello(var):
    print(var)

# vertical alignment

example2 = function(first_var, second_var,
                    third_Var, fourth_var)

# hanging indent
example3 = function(
    first_var, second_var,
    third_var, fourth_var)
```

1. You can use similar workflows for conditionals that either go too long or you want to be a continuation line. You can use parenthesis after a two-character keyword will allow the indentation to be automatic when going to a new line. This can be useful for long conditional statements, such as an `if` statement. For nested statements you can choose to have no extra indentation, comment your indentation details, or add extra indentation.

2. You can line the closing punctuation (brace, bracket, or parenthesis) up with the first character of the last line or the first character that starts the multiline expression that is not whitespace.

```
# last line, first character
new_list = [
    'one', 'two', 'three',
    'four', 'five', 'six'
    ]# first line, first character
new_list = [
    'one', 'two', 'three',
    'four', 'five', 'six'
]
```

2.Maximum Line Length

Generally, it's good to aim for a line length of 79 characters in your Python code.

Following this target number has many advantages. A couple of them are the following:

- It is possible to open files side by side to compare;
- You can view the whole expression without scrolling horizontally which adds to better readability and understanding of the code.

Comments should have 72 characters of line length. You'll learn more about the most common conventions for comments later on in this tutorial!

3. Blank Lines

In Python scripts, top-level function and classes are separated by two blank lines. Method definitions inside classes should be separated by one blank line. You can see this clearly in the following example:

```
class SwapTestSuite(unittest.TestCase):
    """
        Swap Operation Test Case
    """
    def setUp(self):
        self.a = 1
        self.b = 2
#one blank
#two blank
def test_swap_operations(self):
        instance = Swap(self.a,self.b)
        value1, value2 =instance.get_swap_values()
        self.assertEqual(self.a, value2)
        self.assertEqual(self.b, value1)


class OddOrEvenTestSuite(unittest.TestCase):
    """
        This is the Odd or Even Test case Suite
    """
    def setUp(self):
        self.value1 = 1
        self.value2 = 2
#one blank
#two blank
    def test_odd_even_operations(self):
        instance1 = OddOrEven(self.value1)
        instance2 = OddOrEven(self.value2)
        message1 = instance1.get_odd_or_even()
        message2 = instance2.get_odd_or_even()
        self.assertEqual(message1, 'Odd')
        self.assertEqual(message2, 'Even')
```

4. Whitespaces in Expressions and Statements

You should try to avoid whitespaces when you see your code is written just like in the following examples:

Good	Not Good
`func(data, {pivot: 4})`	`func( data, { pivot: 4 } )`
`indexes = (0,)`	`indexes = (0, )`
`if x == 4: print x, y; x, y = y, x`	`if x == 4 : print x , y ; x , y = y , x`

`x = 1`
`y = 2`
`long_variable = 3`
	`x = 1`
	`y = 2`
	`long_variable = 3`

5.Imports

Importing libraries and/or modules is something that you'll often do when you're working with Python for data science. As you might already know, you should always import libraries at the start of your script.

Note that if you do many imports, you should make sure to state each import on a single line.

Good	Not Good
`import os` `import sys`	`import os, sys`

- **Importing order**

    1. Standard library imports.
    2. Related third-party imports.
    3. Local application/library specific imports.

- **Absolute and Relative Imports**

Next, it's good to know the difference between absolute and relative imports. In general, absolute imports are preferred in Python, as it adds up more readability. However, as your application becomes more complex, you can go on using the relative imports also. Implicit relative imports should never be used and have been removed in Python 3.

But what are these absolute and relative imports?

An absolute import is an import that uses the absolute path of the function or class, separated by `.`. For example,

```
import sklearn.linear_model.LogisticRegression
```

A relative import is an import that is relative to the current position where your Python file exists. You could use this type of import if your project structure is growing, as it will make your project more readable. That means that, if you have a Python project structure like the following:

```
.
├── __init__.py
├── __init__.pyc
├── __pycache__
│   ├── __init__.cpython-35.pyc
│   ├── bubble_sort.cpython-35.pyc
│   ├── selection_sort.cpython-35.pyc
├── bubble_sort.py
├── heap_sort.py
├── insertion_sort.py
├── insertion_sort.pyc
├── merge_sort.py
├── merge_sort.pyc
├── quick_sort.py
├── radix_sort.py
├── selection_sort.py
├── selection_sort.pyc
├── shell_sort.py
├── tests
│   ├── test1.py
```

You could use a relative import to import the bubble sort algorithm `BubbleSort`, stored in `bubble_sort.py` in `test1`. That would look like this:

```
from ..bubble_sort import BubbleSort
```

- **Wildcard**

Lastly, you should try to avoid wildcard imports, because they do not add to the readability; You have no view on which classes, methods or variables you are using from your module, for example:

- `from scikit import *`

OBJECT ORIENTED PYTHON

How to Define a Class?

All class definitions start with the `class` keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class's body.

Here's an example of a `Dog` class:

```
class Dog:
    pass
```

The body of the `Dog` class consists of a single statement:
the `pass` keyword. `pass` is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

Init method in Python

init

__init__ is one of the reserved methods in Python. In object oriented programming, it is known as a constructor. The __init__ method can be called when an object is created from the class, and access is required to initialize the attributes of the class.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Creating an object in Python

Class definition:

```
class Person:
    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')


# Output: 10
print(Person.age)

# Output: <function Person.greet>
print(Person.greet)
```

Creating object:

```
P1= Person ()
```

Class with constructor:

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')
```

Creating object:

```
# Create a new ComplexNumber object
num1 = ComplexNumber(2, 3)
```

Deleting attribute from object

```
class ComplexNumber:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag = i

    def get_data(self):
        print(f'{self.real}+{self.imag}j')
```

```
num1 = ComplexNumber(2,3)
>>> del num1.imag
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'imag'

>>> del ComplexNumber.get_data
>>> num1.get_data()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'get_data'
```

We can even delete the object itself, using the del statement.

```
>>> c1 = ComplexNumber(1,3)
>>> del c1
>>> c1
Traceback (most recent call last):
...
NameError: name 'c1' is not define
```

Access modifier in Python

All members in a Python class are **public** by default. Any member can be accessed from outside the class environment.

Example: Public Attributes

Copy

```
class Student:
    schoolName = 'XYZ School' # class attribute

    def __init__(self, name, age):
        self.name=name # instance attribute
        self.age=age # instance attribute
```

You can access the `Student` class's attributes and also modify their values, as shown below.

Example: Access Public Members

Copy

```
>>> std = Student("Steve", 25)
>>> std.schoolName
'XYZ School'
>>> std.name
'Steve'
>>> std.age = 20
>>> std.age
20
```

Protected Members

Protected members of a class are accessible from within the class and are also available to its sub-classes. No other environment is permitted access to it. This enables specific resources of the parent class to be inherited by the child class.

Python's convention to make an instance variable **protected** is to add a prefix _ (single underscore) to it. This effectively prevents it from being accessed unless it is from within a sub-class.

Example: Protected Attributes

Copy

```
class Student:
    _schoolName = 'XYZ School' # protected class attribute

    def __init__(self, name, age):
        self._name=name  # protected instance attribute
        self._age=age # protected instance attribute
```

In fact, this doesn't prevent instance variables from accessing or modifying the instance. You can still perform the following operations:

Example: Access Protected Members

Copy

```
>>> std = Student("Swati", 25)
>>> std._name
'Swati'
>>> std._name = 'Dipa'
>>> std._name
'Dipa'
```

However, you can define a property using property decorator and make it protected, as shown below.

Example: Protected Attributes

Copy

```
class Student:
	def __init__(self,name):
		self._name = name
	@property
	def name(self):
		return self._name
	@name.setter
	def name(self,newname):
		self._name = newname
```

Above, @property decorator is used to make the `name()` method as property and `@name.setter` decorator to another overloads of the `name()` method as property setter method. Now, `_name` is protected.

Example: Access Protected Members

Copy

```
>>> std = Student("Swati")
>>> std.name
'Swati'
>>> std.name = 'Dipa'
>>> std.name
'Dipa'
>>> std._name # still accessible
```

Above, we used `std.name` property to modify `_name` attribute. However, it is still accessible in Python. Hence, the responsible programmer would refrain from accessing and modifying instance variables prefixed with `_` from outside its class.

Private Members

Python doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the variable/method with a single or double underscore to emulate the behavior of protected and private access specifiers.

The double underscore `__` prefixed to a variable makes it **private**. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:

Example: Private Attributes

Copy

```
class Student:
    __schoolName = 'XYZ School' # private class attribute

    def __init__(self, name, age):
        self.__name=name  # private instance attribute
        self.__salary=age # private instance attribute
    def __display(self):  # private method
           print('This is private method.')
```

Example:

Copy

```
>>> std = Student("Bill", 25)
>>> std.__schoolName
AttributeError: 'Student' object has no attribute '__schoolName'
>>> std.__name
AttributeError: 'Student' object has no attribute '__name'
>>> std.__display()
AttributeError: 'Student' object has no attribute '__display'
```

Python performs name mangling of private variables. Every member with a double underscore will be changed to `_object._class__variable`. So, it can still be accessed from outside the class, but the practice should be refrained.

Example:

Copy

```
>>> std = Student("Bill", 25)
>>> std._Student__name
'Bill'
>>> std._Student__name = 'Steve'
>>> std._Student__name
'Steve'
>>> std._Student__display()
'This is private method.'
```

Thus, Python provides conceptual implementation of public, protected, and private access modifiers, but not like other languages like C#, Java, C++.

Decorator in Python

Python Property Decorator - @property

The `@property` decorator is a built-in decorator in Python for the property() function. Use `@property` decorator on any method in the class to use the method as a property.

You can use the following three decorators to define a property:

- @property: Declares the method as a property.
- @<property-name>.setter: Specifies the setter method for a property that sets the value to a property.
- @<property-name>.deleter: Specifies the delete method as a property that deletes a property.

Declare a Property

The following declares the method as a property. This method must return the value of the property.

Example: @property decorator
Copy

```
class Student:

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

    @property
    def name(self):
        return self.__name
```

Above, `@property` decorator applied to the `name()` method. The `name()` method returns the private instance attribute value `__name`. So, we can now use the `name()` method as a property to get the value of the `__name` attribute, as shown below.

Example: Access Property decorator
Copy

```
>>> s = Student('Steve')
>>> s.name
'Steve'
```

Property Setter

Above, we defined the `name()` method as a property. We can only access the value of the `name` property but cannot modify it. To modify the property value, we must define the setter method for the `name` property using `@property-name.setter` decorator, as shown below.

Example: Property Setter
Copy

```
class Student:

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

    @property
    def name(self):
        return self.__name

    @name.setter   #property-name.setter decorator
    def name(self, value):
        self.__name = value
```

Above, we have two overloads of the `name()` method. One is for the getter and another is the setter method. The setter method must have the value argument that can be used to assign to the underlying private attribute. Now, we can retrieve and modify the property value, as shown below.

Example: Access Property
Copy

```
>>> s = Student('Steve')
>>> s.name
'Steve'
>>> s.name = 'Bill'
'Bill'
```

Property Deleter

Use the `@property-name.deleter` decorator to define the method that deletes a property, as shown below.

Example: Property Deleter
Copy

```
class Student:
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
```

```
        return self.__name

    @name.setter
    def name(self, value):
        self.__name=value

    @name.deleter   #property-name.deleter decorator
    def name(self, value):
        print('Deleting..')
        del self.__name
```

The deleter would be invoked when you delete the property using keyword `del`, as shown below. Once you delete a property, you cannot access it again using the same instance.

Example: Delete a Property

Copy

```
>>> s = Student('Steve')
>>> del s.name
Deleting..
>>> s.name
Traceback (most recent call last):
File "<pyshell#16>", line 1, in <module>
    p.name
File "C:\Python37\test.py", line 6, in name
    return self.__name
AttributeError: 'Student' object has no attribute '_Student__name'
```

Python Class Method Decorator @classmethod

In Python, the `@classmethod` decorator is used to declare a method in the class as a class method that can be called using `ClassName.MethodName()`. The class method can also be called using an object of the class.

The `@classmethod` is an alternative of the classmethod() function. It is recommended to use the `@classmethod` decorator instead of the function because it is just a syntactic sugar.

@classmethod Characteristics

- Declares a class method.
- The first parameter must be `cls`, which can be used to access class attributes.
- The class method can only access the class attributes but not the instance attributes.
- The class method can be called using `ClassName.MethodName()` and also using object.

- It can return an object of the class.

The following example declares a class method.

Example: @classmethod
Copy
```
class Student:
    name = 'unknown' # class attribute
    def __init__(self):
        self.age = 20  # instance attribute

    @classmethod
    def tostring(cls):
        print('Student Class Attributes: name=',cls.name)
```

Above, the `Student` class contains a class attribute `name` and an instance attribute `age`. The `tostring()` method is decorated with the `@classmethod` decorator that makes it a class method, which can be called using the `Student.tostring()`. Note that the first parameter of any class method must be `cls` that can be used to access the class's attributes. You can give any name to the first parameter instead of `cls`.

Now, you can use the class method, as shown below.

Example: Access Class Method
Copy
```
>>> Student.tostring()
Student Class Attributes: name=unknown
```

However, the same method can be called using an object also.

Example: Calling Class Method using Object
Copy
```
>>> std = Student()
>>> std.tostring()
Student Class Attributes: name=unknown
>>> Student().tostring()
Student Class Attributes: name=unknown
```

The class method can only access class attributes, but not the instance attributes. It will raise an error if trying to access the instance attribute in the class method.

Example: @classmethod
Copy
```
class Student:
    name = 'unknown' # class attribute
    def __init__(self):
        self.age = 20  # instance attribute
```

```
    @classmethod
    def tostring(cls):
        print('Student Class Attributes: name=',cls.name,', age=',
cls.age)
```

Example: Access Class Method

Copy

```
>>> Student.tostring()
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    Student.tostring()
  File "<pyshell#21>", line 7, in display
    print('Student Class Attributes: name=',cls.name,', age=',
cls.age)
AttributeError: type object 'Student' has no attribute 'age'
```

The class method can also be used as a factory method to get an object of the class, as shown below.

Example: @classmethod

Copy

```
class Student:

    def __init__(self, name, age):
        self.name = name  # instance attribute
        self.age = age # instance attribute

    @classmethod
    def getobject(cls):
        return cls('Steve', 25)
```

The following calls the class method to get an object.

Example: Class Method as Factory Method

Copy

```
>>> std = Student.getobject()
>>> std.name
'Steve'
>>> std.age
25
```

@classmethod vs @staticmethod

The following table lists the difference between the class method and the static method:

@classmethod	@staticmethod
Declares a class method.	Declares a static method.
It can access class attributes, but not the instance attributes.	It cannot access either class attributes or instance attributes.
It can be called using the `ClassName.MethodName()` or `object.MethodName()`.	It can be called using the `ClassName.MethodName()` or `object.MethodName()`.
It can be used to declare a factory method that returns objects of the class.	It cannot return an object of the class.

Define Static Method using @staticmethod Decorator in Python

The `@staticmethod` is a built-in decorator that defines a static method in the class in Python. A static method doesn't receive any reference argument whether it is called by an instance of a class or by the class itself.

@staticmethod Characteristics

- Declares a static method in the class.
- It cannot have `cls` or `self` parameter.
- The static method cannot access the class attributes or the instance attributes.
- The static method can be called using `ClassName.MethodName()` and also using `object.MethodName()`.
- It can return an object of the class.

The following example demonstrates how to define a static method in the class:

Example: Define Static Method

Copy

```
class Student:
    name = 'unknown' # class attribute

    def __init__(self):
        self.age = 20  # instance attribute

    @staticmethod
    def tostring():
        print('Student Class')
```

Above, the `Student` class declares the `tostring()` method as a static method using the `@staticmethod` decorator. Note that it cannot have `self` or `cls` parameter.

The static method can be called using the `ClassName.MethodName()` or `object.MethodName()`, as shown below.

Example: Calling Class Method using Object

Copy

```
>>> Student.tostring()
'Student Class'
>>> Student().tostring()
'Student Class'
>>> std = Student()
>>> std.tostring()
'Student Class'
```

The static method cannot access the class attributes or instance attributes. It will raise an error if try to do so.

Example: Static Method

Copy

```
class Student:
    name = 'unknown' # class attribute

    def __init__(self):
        self.age = 20  # instance attribute

    @staticmethod
    def tostring():
        print('name=',name,'age=',self.age)
```

The following will be the output when you call the above static method.

```
>>> Student.tostring()
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    Student.tostring()
  File "<pyshell#21>", line 7, in display
    print('name=',name,'age=',self.age)
NameError: name 'name' is not defined
```

Inheritance in Python

The child class inherits data definitions and methods from the parent class. This facilitates the reuse of features already available. The child class can add a few more definitions or redefine a base class method.

This feature is extremely useful in building a hierarchy of classes for objects in a system. It is also possible to design a new class based upon more than one existing classes. This feature is called multiple inheritance.

The general mechanism of establishing inheritance is illustrated below:

Syntax:

```
class parent:
    statements

class child(parent):
    statements
```

While defining the child class, the name of the parent class is put in the parentheses in front of it, indicating the relation between the two. Instance attributes and methods defined in the parent class will be inherited by the object of the child class.

To demonstrate a more meaningful example, a quadrilateral class is first defined, and it is used as a base class for the rectangle class.

A quadrilateral class having four sides as instance variables and a perimeter() method is defined below:

Example:

```
class quadriLateral:
    def __init__(self, a, b, c, d):
        self.side1=a
        self.side2=b
        self.side3=c
        self.side4=d

    def perimeter(self):
        p=self.side1 + self.side2 + self.side3 + self.side4
        print("perimeter=",p)
```

The constructor (the `__init__()` method) receives four parameters and assigns them to four instance variables. To test the above class, declare its object and invoke the `perimeter()` method.

```
>>>q1=quadriLateral(7,5,6,4)

>>>q1.perimeter()

perimeter=22
```

We now design a rectangle class based upon the quadriLateral class (rectangle IS a quadrilateral!). The instance variables and the perimeter() method from the base class should be automatically available to it without redefining it.

Since opposite sides of the rectangle are the same, we need only two adjacent sides to construct its object. Hence, the other two parameters of the __init__() method are set to none. The __init__() method forwards the parameters to the constructor of its base (quadrilateral) class using the **super()** function. The object is initialized with side3 and side4 set to none. Opposite sides are made equal by the constructor of rectangle class. Remember that it has automatically inherited the perimeter()method, hence there is no need to redefine it.

Example: Inheritance

```
class rectangle(quadriLateral):
    def __init__(self, a, b):
        super().__init__(a, b, a, b)
```

We can now declare the object of the rectangle class and call the perimeter()method.

```
>>> r1=rectangle(10, 20)

>>> r1.perimeter()

perimeter=60
```

Overriding in Python

First, we shall define a new method named area() in the rectangle class and use it as a base for the square class. The area of rectangle is the product of its adjacent sides.

Example:

```
class rectangle(QuadriLateral):
    def __init__(self, a,b):
        super().__init__(a, b, a, b)


    def area(self):
        a = self.side1 * self.side2
        print("area of rectangle=", a)
```

Let us define the square class which inherits the rectangle class. The `area()` method is overridden to implement the formula for the area of the square as the square of its sides.

Example:

```
class square(rectangle):
    def __init__(self, a):
        super().__init__(a, a)


    def area(self):
        a=pow(self.side1, 2)
        print('Area of Square: ', a)
```

```
>>>s=Square(10)

>>>s.area()

Area of Square: 100
```

Python Inheritance Syntax

```
class BaseClass:

  Body of base class

class DerivedClass(BaseClass):

  Body of derived class
```

Derived class inherits features from the base class where new features can be added to it. This results in re-usability of code.

Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called `Polygon` defined as follows.

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i
in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides `n` and magnitude of each side as a list called `sides`.

The `inputSides()` method takes in the magnitude of each side and `dispSides()` displays these side lengths.

A triangle is a polygon with 3 sides. So, we can create a class called `Triangle` which inherits from `Polygon`. This makes all the attributes of `Polygon` class available to the `Triangle`class. We don't need to define them again (code reusability). `Triangle` can be defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, class `Triangle` has a new method `findArea()` to find and print the area of the triangle. Here is a sample run.

```
>>> t = Triangle()

>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4

>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0

>>> t.findArea()
The area of the triangle is 6.00
```

We can see that even though we did not define methods like `inputSides()` or `dispSides()` for class `Triangle` separately, we were able to use them.

If an attribute is not found in the class itself, the search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Method Overriding in Python

In the above example, notice that `__init__()` method was defined in both classes, `Triangle` as well `Polygon`. When this happens, the method in the derived class overrides that in the base class. This is to say, `__init__()` in `Triangle` gets preference over the `__init__` in `Polygon`.

Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling `Polygon.__init__()` from `__init__()` in `Triangle`).
A better option would be to use the built-in function `super()`. So, `super().__init__(3)` is equivalent to `Polygon.__init__(self,3)` and is preferred.

Two built-in functions `isinstance()` and `issubclass()` are used to check inheritances.
The function `isinstance()` returns `True` if the object is an instance of the class or other classes derived from it.

```
>>> isinstance(t,Triangle)
True

>>> isinstance(t,Polygon)
True

>>> isinstance(t,int)
False

>>> isinstance(t,object)
True
```

Similarly, issubclass() is used to check for class inheritance.

```
>>> issubclass(Polygon,Triangle)
False

>>> issubclass(Triangle,Polygon)
True

>>> issubclass(bool,int)
True
```

Python Multiple Inheritance

A class can be derived from more than one base class in Python, similar to C++. This is called multiple inheritance.

In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

Example

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

Python Multilevel Inheritance

We can also inherit from a derived class. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class are inherited into the new derived class.

An example with corresponding visualization is given below.

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```

Here, the `Derived1` class is derived from the `Base` class, and the `Derived2` class is derived from the `Derived1` class.

Method Resolution Order in Python

Every class in Python is derived from the `object` class. It is the most base type in Python.
So technically, all other classes, either built-in or user-defined, are derived classes and all objects are instances of the `object` class.

```
# Output: True
print(issubclass(list,object))

# Output: True
print(isinstance(5.5,object))

# Output: True
print(isinstance("Hello",object))
```

In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching the same class twice.

So, in the above example of `MultiDerived` class the search order is [`MultiDerived`, `Base1`, `Base2`, `object`]. This order is also called linearization of `MultiDerived` class and the set of rules used to find this order is called **Method Resolution Order (MRO)**.

MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents. In case of multiple parents, the order is the same as tuples of base classes.

MRO of a class can be viewed as the `__mro__` attribute or the `mro()` method. The former returns a tuple while the latter returns a list.

```
>>> MultiDerived.__mro__
(<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>)

>>> MultiDerived.mro()
[<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>]
```

Visualizing Multiple Inheritance in Python

```
# Demonstration of MRO

class X:
    pass


class Y:
    pass
```

```
class Z:
    pass


class A(X, Y):
    pass


class B(Y, Z):
    pass


class M(B, A, Z):
    pass

# Output:
# [<class '__main__.M'>, <class '__main__.B'>,
#  <class '__main__.A'>, <class '__main__.X'>,
#  <class '__main__.Y'>, <class '__main__.Z'>,
#  <class 'object'>]

print(M.mro())
```

Output

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>,
<class '__main__.X'>, <class '__main__.Y'>, <class '__main__.Z'>,
<class 'object'>]
```