Sieve of Eratosthenes

In mathematics, the **Sieve of Eratosthenes** is a simple, ancient algorithm for finding all prime numbers up to any given limit.

It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2. The multiples of a given prime are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime. This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime.

One of a number of prime number sieves, it is one of the most efficient ways to find all of the smaller primes. It may be used to find primes in arithmetic progressions.

Algorithm:-

Following is the algorithm to find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1. Create a list of consecutive integers from 2 to n: $(2, 3, 4, \ldots, n)$.
2. Initially, let p equal 2, the first prime number.
3. Starting from p^2, count up in increments of p and mark each of these numbers greater than or equal to p^2 itself in the list. These numbers will be $p(p+1)$, $p(p+2)$, $p(p+3)$, etc..
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this number (which is the next prime), and repeat from step 3.
5. When the algorithm terminates, all the numbers in the list that are not marked are prime.

References:

1. https://www.geeksforgeeks.org/sieve-of-eratosthenes/
2. https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes#/media/File:Sieve_of_Eratosthenes_animation.gif

Example:

To find all the prime numbers less than or equal to 30, proceed as follows.

First, generate a list of integers from 2 to 30:

```
 2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30
```

The first number in the list is 2; cross out every 2nd number in the list after 2 by counting up from 2 in increments of 2 (these will be all the multiples of 2 in the list):

 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25
~~26~~ 27 ~~28~~ 29 ~~30~~

The next number in the list after 2 is 3; cross out every 3rd number in the list after 3 by counting up from 3 in increments of 3 (these will be all the multiples of 3 in the list):

 2 3 ~~4~~ 5 ~~6~~ 7 ~~8 9 10~~ 11 ~~12~~ 13 ~~14 15 16~~ 17 ~~18~~ 19 ~~20 21 22~~ 23 ~~24~~ 25
~~26 27 28~~ 29 ~~30~~

The next number not yet crossed out in the list after 3 is 5; cross out every 5th number in the list after 5 by counting up from 5 in increments of 5 (i.e. all the multiples of 5):

 2 3 ~~4~~ 5 ~~6~~ 7 ~~8 9 10~~ 11 ~~12~~ 13 ~~14 15 16~~ 17 ~~18~~ 19 ~~20 21 22~~ 23 ~~24 25~~
~~26 27 28~~ 29 ~~30~~

The next number not yet crossed out in the list after 5 is 7; the next step would be to cross out every 7th number in the list after 7, but they are all already crossed out at this point, as these numbers (14, 21, 28) are also multiples of smaller primes because 7 × 7 is greater than 30. The numbers not crossed out at this point in the list are all the prime numbers below 30:

```
 2  3     5     7           11    13          17    19          23
29
```

Python Program for Sieve of Eratosthenes

```
# Python program to print all primes smaller than or equal to
# n using Sieve of Eratosthenes

def SieveOfEratosthenes(n):

    # Create a boolean array "prime[0..n]" and initialize
    # all entries it as true. A value in prime[i] will
    # finally be false if i is Not a prime, else true.
    prime = [True for i in range(n + 1)]
    p = 2
    while (p * p <= n):

        # If prime[p] is not changed, then it is a prime
        if (prime[p] == True):

            # Update all multiples of p
            for i in range(p * 2, n + 1, p):
                prime[i] = False
        p += 1
    prime[0]= False
    prime[1]= False
    # Print all prime numbers
    for p in range(n + 1):
        if prime[p]:
            print p,

# driver program
if __name__=='__main__':
    n = 30
    print ("Following are the prime numbers smaller than or equal
    to",n)
    SieveOfEratosthenes(n)
```

Import module in Python

Import in python is similar to #include header_file in C/C++. Python modules can get access to code from another module by importing the file/function using import. The import statement is the most common way of invoking the import machinery, but it is not the only way.

Import module_name

When import is used, it searches for the module initially in the local scope by calling __import__() function. The value returned by the function is then reflected in the output of the initial code.

Example:

```
import math
print(math.pi)
```

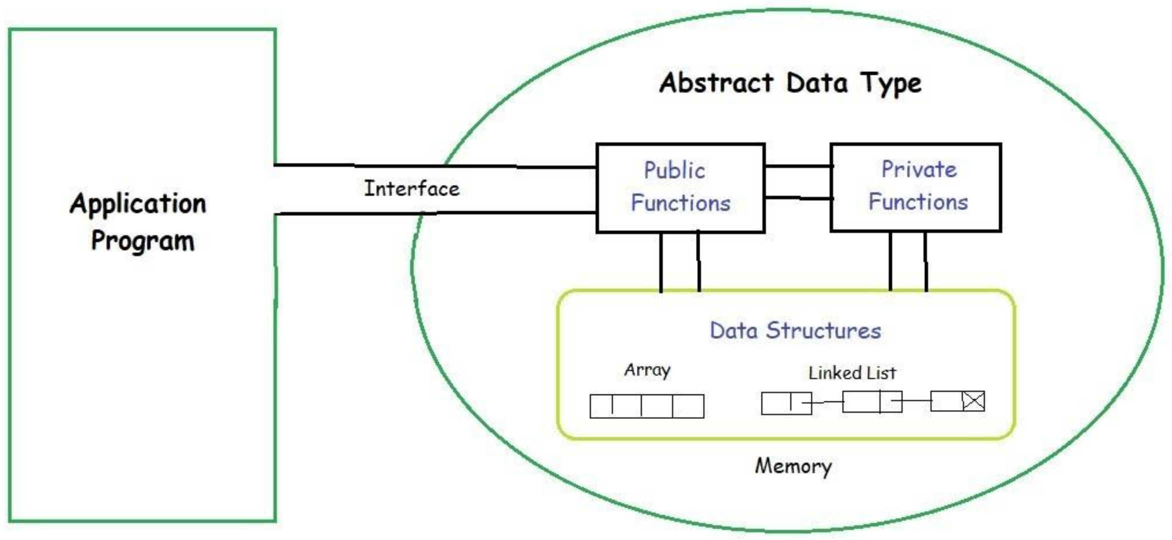
OR

```
import numpy as np
```

<u>Abstract Data Types</u>

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.

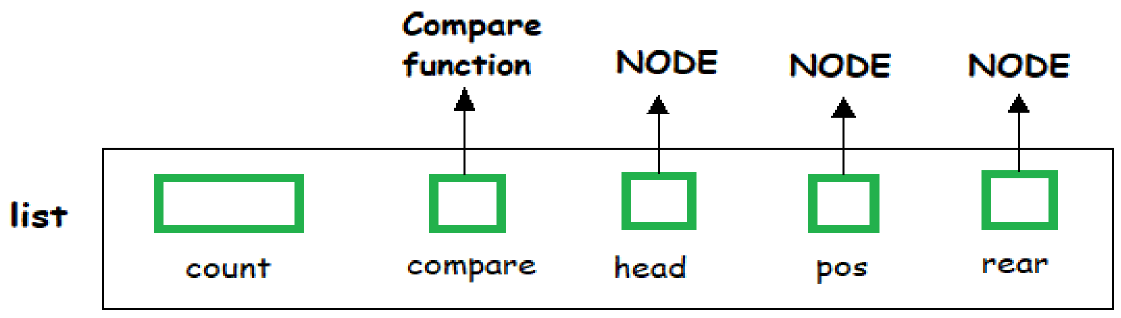
The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

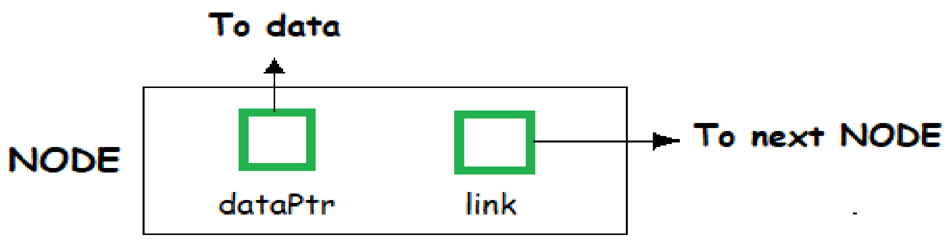


The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented. So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

List ADT

The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list.

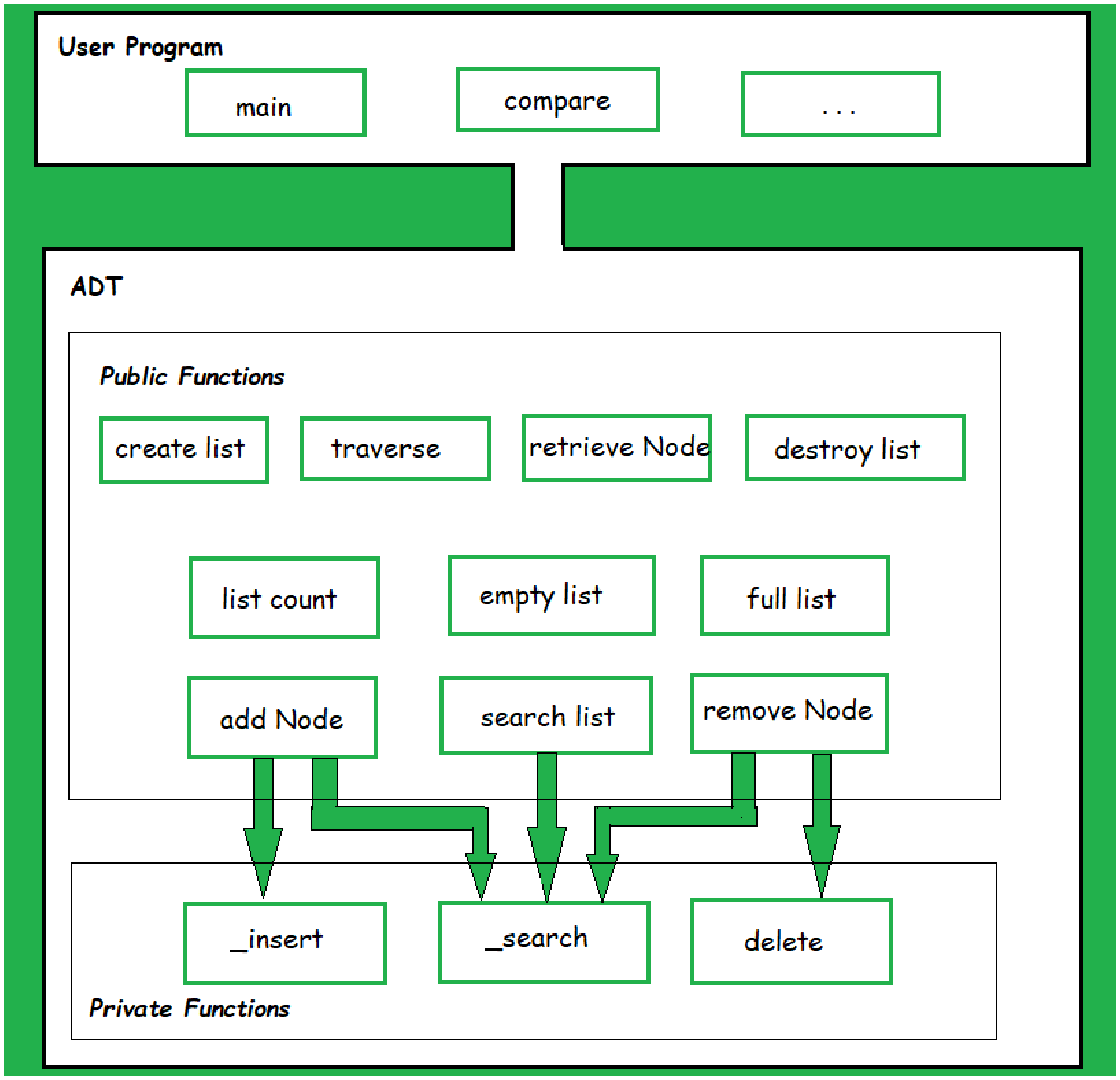




The data node contains the pointer to a data structure and a self-referential pointer which points to the next node in the list.

```
//List ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *link;
} Node;
typedef struct
{
 int count;
 Node *pos;
 Node *head;
 Node *rear;
 int (*compare) (void *argument1, void *argument2)
} LIST;
```

The List ADT Functions is given below:



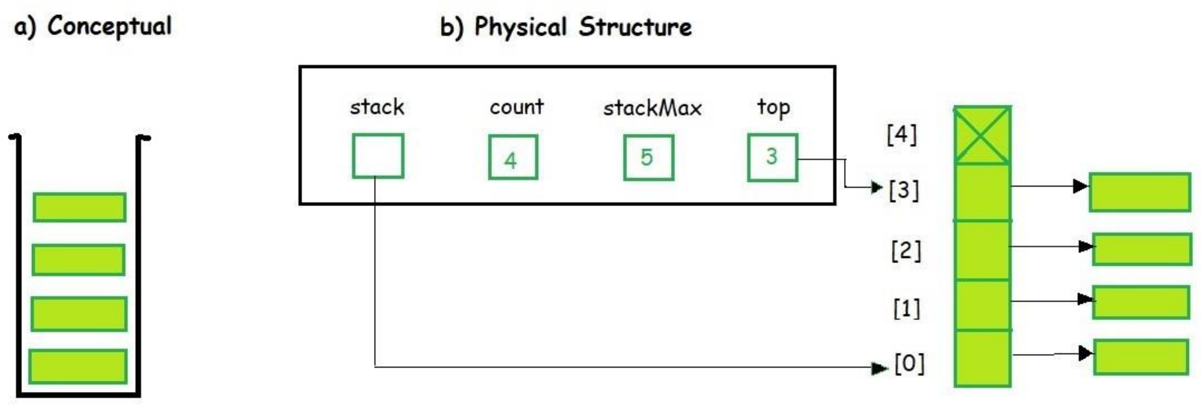
A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.

get() – Return an element from the list at any given position.
insert() – Insert an element at any position of the list.
remove() – Remove the first occurrence of any element from a non-empty list.
removeAt() – Remove the element at a specified location from a non-empty list.

replace() – Replace an element at any position by another element.
size() – Return the number of elements in the list.
isEmpty() – Return true if the list is empty, otherwise return false.
isFull() – Return true if the list is full, otherwise return false.

Stack ADT

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
The program allocates memory for the data and address is passed to the stack ADT.



The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
The stack head structure also contains a pointer to top and count of number of entries currently in stack.

```
//Stack ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *link;
} StackNode;
typedef struct
{
 int count;
 StackNode *top;
```

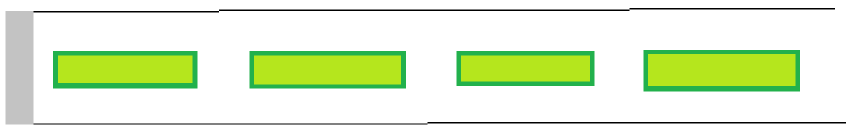
} STACK;

A Stack contains elements of the same type arranged in sequential order. All operations take place at a single end that is top of the stack and following operations can be performed:

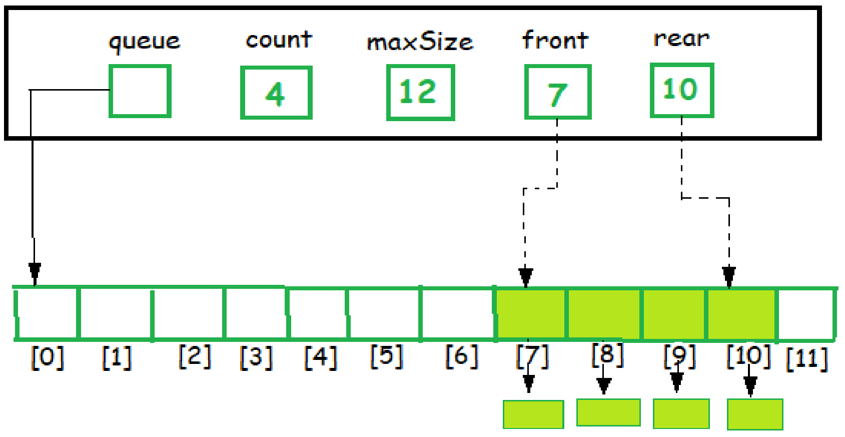
push() – Insert an element at one end of the stack called top.
pop() – Remove and return the element at the top of the stack, if it is not empty.
peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
size() – Return the number of elements in the stack.
isEmpty() – Return true if the stack is empty, otherwise return false.
isFull() – Return true if the stack is full, otherwise return false.

Queue ADT

The queue abstract data type (ADT) follows the basic design of the stack abstract data type.



a) Conceptual



b) Physical Structures

Each node contains a void pointer to the data and the link pointer to the next element in the queue. The program's responsibility is to allocate memory for storing the data.

```
//Queue ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *next;
} QueueNode;
typedef struct
{
 QueueNode *front;
 QueueNode *rear;
 int count;
} QUEUE;
```

A Queue contains elements of the same type arranged in sequential order. Operations take place at both ends, insertion is done at the end and deletion is done at the front. Following operations can be performed:

enqueue() – Insert an element at the end of the queue.
dequeue() – Remove and return the first element of the queue, if the queue is not empty.
peek() – Return the element of the queue without removing it, if the queue is not empty.
size() – Return the number of elements in the queue.
isEmpty() – Return true if the queue is empty, otherwise return false.
isFull() – Return true if the queue is full, otherwise return false.
From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.

After going through the basics of python, you would be interested to know more about further and bit more advance topics of the Python3 programming language.
Please remember that Python completely works on indentation and it is advised to practice it a bit by running some programs. Use the tab key to provide indentation to your code.

This article is divided in following five sections:

Classes
Just like every other Object Oriented Programming language Python supports classes. Let’s look at some points on Python classes.
Classes are created by keyword class.
Attributes are the variables that belong to class.
Attributes are always public and can be accessed using dot (.) operator.
Eg.: Myclass.Myattribute
A sample E.g for classes:

```
# creates a class named MyClass
class MyClass:
        # assign the values to the MyClass attributes
        number = 0
        name = "noname"

def Main():
        # Creating an object of the MyClass.
        # Here, 'me' is the object
        me = MyClass()

        # Accessing the attributes of MyClass
        # using the dot(.) operator
        me.number = 1337
        me.name = "Harssh"

        # str is an build-in function that
        # creates an string
```

```
    print(me.name + " " + str(me.number))

# telling python that there is main in the program.
if __name__=='__main__':
    Main()
```

Output :
```
Harssh 1337
```

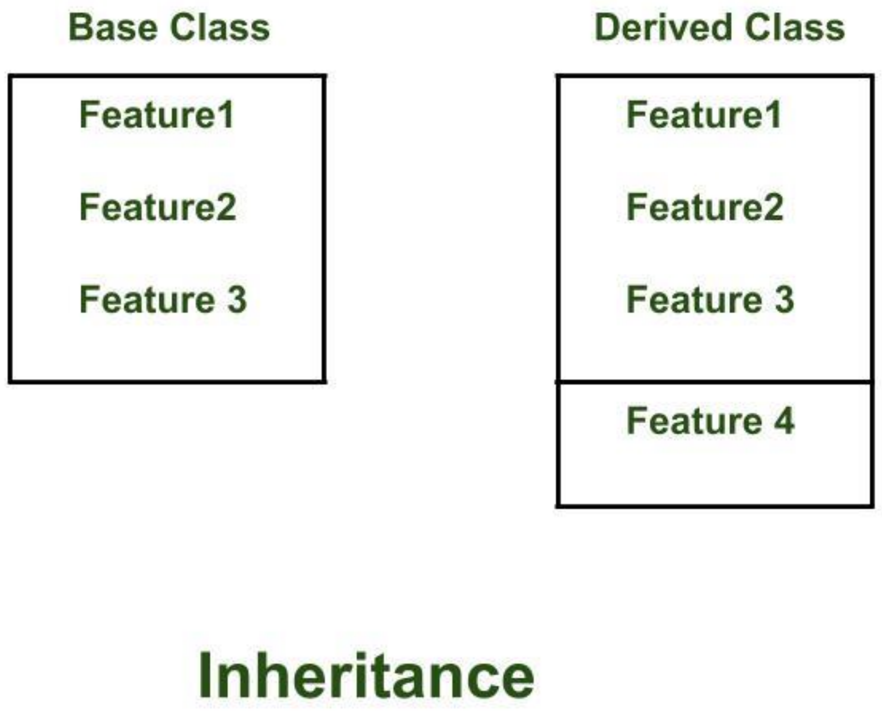
<u>Methods</u>

Method is a bunch of code that is intended to perform a particular task in your Python’s code.Function that belongs to a class is called an Method. All methods require ‘self’ parameter. If you have coded in other OOP language you can think of ‘self’ as the ‘this’ keyword which is used for the current object. It unhides the current instance variable.’self’ mostly work like ‘this’.‘def’ keyword is used to create a new method.

```
# A Python program to demonstrate working of class
# methods

class Vector2D:
    x = 0.0
    y = 0.0

    # Creating a method named Set
    def Set(self, x, y):
        self.x = x
        self.y = y

def Main():
    # vec is an object of class Vector2D
    vec = Vector2D()

    # Passing values to the function Set
    # by using dot(.) operator.
    vec.Set(5, 6)
    print("X: " + str(vec.x) + ", Y: " + str(vec.y))

if __name__=='__main__':
    Main()
```
Output :
```
X: 5, Y: 6
```

Inheritance

Inheritance is defined as a way in which a particular class inherits features from its base class. Base class is also knows as ‘Superclass’ and the class which inherits from the Superclass is known as ‘Subclass’.



As shown in the figure the Derived class can inherit features from its base class, also it can define its own features too.

# Syntax for inheritance

```
class derived-classname(superclass-name)
# A Python program to demonstrate working of inheritance
class Pet:
	#__init__ is an constructor in Python
	def __init__(self, name, age):
		self.name = name
		self.age = age

# Class Cat inheriting from the class Pet
class Cat(Pet):
	def __init__(self, name, age):
		# calling the super-class function __init__
		# using the super() function
		super().__init__(name, age)

def Main():
	thePet = Pet("Pet", 1)
	jess = Cat("Jess", 3)

	# isinstance() function to check whether a class is
	# inherited from another class
	print("Is jess a cat? " +str(isinstance(jess, Cat)))
	print("Is jess a pet? " +str(isinstance(jess, Pet)))
	print("Is the pet a cat? "+str(isinstance(thePet, Cat)))
	print("Is thePet a Pet? " +str(isinstance(thePet, Pet)))
	print(jess.name)

if __name__=='__main__':
	Main()
```

Output :

```
Is jess a cat? True
Is jess a pet? True
Is the pet a cat? False
Is thePet a Pet? True
Jess
```

Iterators

Iterators are objects that can be iterated upon.
Python uses the __iter__() method to return an iterator object of the class.
The iterator object then uses the __next__() method to get the next item.
for loops stops when Stop Iteration Exception is raised.

```
# This program will reverse the string that is passed
# to it from the main function
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index-= 1
        return self.data[self.index]

def Main():
    rev = Reverse('Drapsicle')
    for char in rev:
        print(char)

if __name__=='__main__':
    Main()
```

Output :

```
e
l
c
i
s
p
a
r
D
```

Generators

Another way of creating iterators.Uses a function rather than a separate class.Generates the background code for the next() and iter() methods.Uses a special statement called yield which saves the state of the generator and set a resume point for when next() is called again.

```
# A Python program to demonstrate working of Generators
def Reverse(data):
    # this is like counting from 100 to 1 by taking one(-1)
    # step backward.
    for index in range(len(data)-1, -1, -1):
        yield data[index]

def Main():
    rev = Reverse('Harssh')
    for char in rev:
        print(char)
    data ='Harssh'
    print(list(data[i] for i in range(len(data)-1, -1, -1)))

if __name__=="__main__":
    Main()
```

Output :

```
h
```

```
s
s
r
a
H
['h', 's', 's', 'r', 'a', 'H']
```

Inheritance

One of the major advantages of Object Oriented Programming is re-use. Inheritance is one of the mechanisms to achieve the same. In inheritance, a class (usually called superclass) is inherited by another class (usually called subclass). The subclass adds some attributes to superclass.

Below is a sample Python program to show how inheritance is implemented in Python.

```
# A Python program to demonstrate inheritance

# Base or Super class. Note object in bracket.
# (Generally, object is made ancestor of all classes)
# In Python 3.x "class Person" is
# equivalent to "class Person(object)"
class Person(object):

    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name

    # To check if this person is employee
    def isEmployee(self):
        return False
# Inherited or Sub class (Note Person in bracket)
class Employee(Person):
 # Here we return true
```

OOPS Concept

Below is a simple Python program that creates a class with single method.

```
# A simple example class
class Test:

    # A sample method
    def fun(self):
        print("Hello")

# Driver code
obj = Test()
obj.fun()
Output:
Hello
```

As we can see above, we create a new class using the class statement and the name of the class. This is followed by an indented block of statements which form the body of the class. In this case, we have defined a single method in the class.
Next, we create an object/instance of this class using the name of the class followed by a pair of parentheses.

The self

Class methods must have an extra first parameter in method definition. We do not give a value for this parameter when we call the method, Python provides it
If we have a method which takes no arguments, then we still have to have one argument – the self. See fun() in above simple example.
This is similar to this pointer in C++ and this reference in Java.
When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.