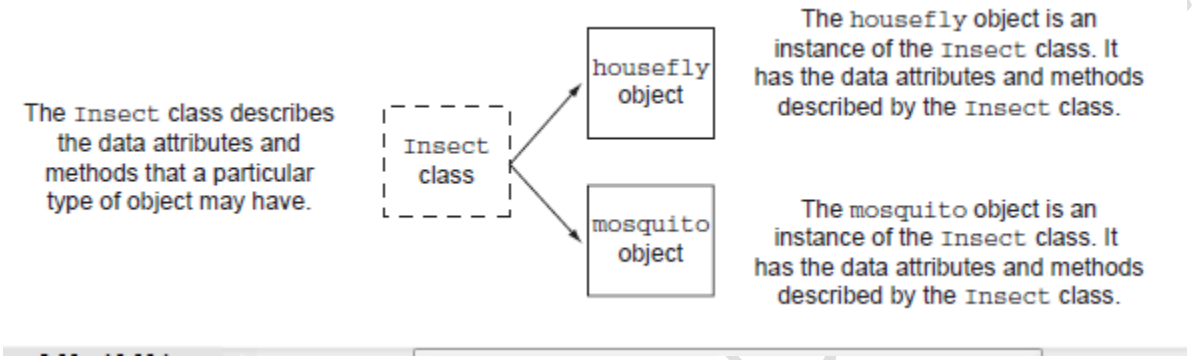
Procedural and Object-Oriented Programming

Procedural programming is a method of writing software. It is a programming practice centered on the procedures or actions that take place in a program. Object-oriented programming is centered on objects. Objects are created from abstract data types that encapsulate data and functions together.

Classes:

A class is code that specifies the data attributes and methods for a particular type of object.



Each object that is created from a class is called an instance of the class.

```
class myclass:
   '''this is my class'''
   a=10
   def func(self):
      print("hello")

ob=myclass()
print(myclass.a)
print(myclass.__doc__)
ob.func()
```

Output :

10

this is my class

hello

When we assign the value through object of the class

Eg. Ob.a=20 # Assigning the value to object instance data members

print(ob.a)

```
print(myclass.a)
```

output:

20

10

deleting the attribute value through object

```
del ob.a
```

Use of self keyword

```
class person():
   def __init__(self, person_name,person_age,person_weight,person_height):
      self.name=person_name
      self.age=person_age
      self.weight=person_weight
      self.height=person_height
ob1=person("honey",22,"70 kg","170 cm")
print(ob1.name)
print(ob1.age)
print(ob1.weight)
```

Inheritance in Python

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
```

```
x.printname()
```

Create the child class using the parent class

```
#create the child class
class student(Person):
   pass # pass keyword is used when you do not
      #want to add any other properties or methods to the class
x=student("honey", "pasricha")
x.printname()
```

Add the __init__() function

```
class Student(Person):
  def __init__(self, fname, lname):
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

```
#create the child class
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
    print("old init")

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe") # make the comment to show overridden method
x.printname() # make the comment
#create the child class
```

```
class student(Person):
   def __init__(self,fname,lname):
      print("hello")
      Person.__init__(self,fname,lname)


       #pass # pass keyword is used when you do not
      #want to add any other properties or methods to the class
y=student("honey", "pasricha")
#y.printname()
```

output

```
old init
John Doe
hello
old init
```

to override the previous method latest will call from child class

Python Iterators

An iterator is an object that contains a countable number of values.

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.

All these objects have a `iter()` method which is used to get an iterator:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
```

```
print(next(myit))
```

Looping Through an Iterator

We can also use a `for` loop to iterate through an iterable object:

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)

mystr = "banana"

for x in mystr:
  print(x)
```

Create an Iterator
To create an object/class as an iterator you have to implement the methods `__iter__()` and `__next__()` to your object.

The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

StopIteration

The example above would continue forever if you had enough next() statements, or if it was used in a `for` loop.

To prevent the iteration to go on forever, we can use the `StopIteration` statement.

In the `__next__()` method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```