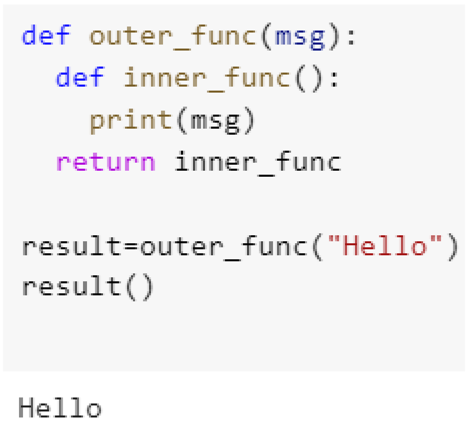
First Class Functions:
First class functions allow us to treat function as values like we can assign function to the variable , we can pass function as parameter to another function and also we can return the function.
Ex- Assigning function to the variable

```
        def square(x):
                Return x^2
        result=square
        result(5) —> 25
```

Ex2- Passing function as argument we can see below function or map(int,[1,2,3,4]) is best example.

Closures: [(https://www.youtube.com/watch?v=FsAPt_9Bf3U)](https://www.youtube.com/watch?v=FsAPt_9Bf3U)
Closure is an inner function present inside the outer function which can access the variable of the outer function even though the outer function has executed.

Ex-



```
def outer_func(msg):
  def inner_func():
    print(msg)
  return inner_func

result=outer_func("Hello")
result()
```

```
Hello
```

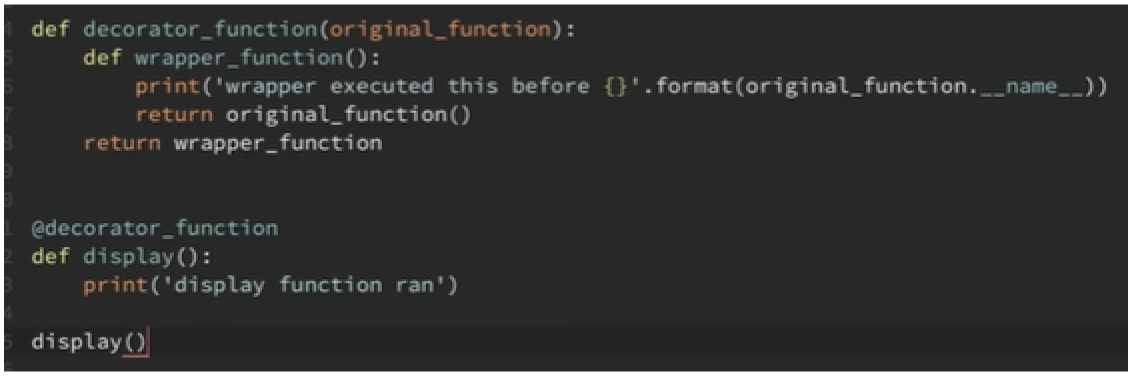
Explanation: Here inner_func() is a clouser which is present inside outer_func() and inner_func() has access to the msg variable of outer_func().

Decorators:
Decorators is just a function which takes another function as an argument and adds some kind of functionality and then returns the function. It does all of this without altering the source code of the function which we passed.

Note: Decorators can be a function as well as Class.

The Purpose of the decorator is that they allow us to add functionality to the existing function without changing that original function

Decorators are mostly used in Flask,Django and also for logging.

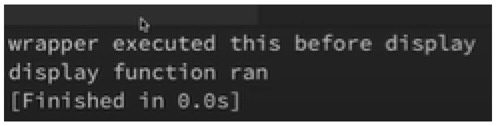


```
def decorator_function(original_function):
    def wrapper_function():
        print('wrapper executed this before {}'.format(original_function.__name__))
        return original_function()
    return wrapper_function


@decorator_function
def display():
    print('display function ran')

display()
```

Output:



```
wrapper executed this before display
display function ran
[Finished in 0.0s]
```

Iterators:

An iterator is an object used to iterate over iterable objects such as lists, tuples, dictionaries, and sets. An object is called iterable if we can get an iterator from it or loop over it.

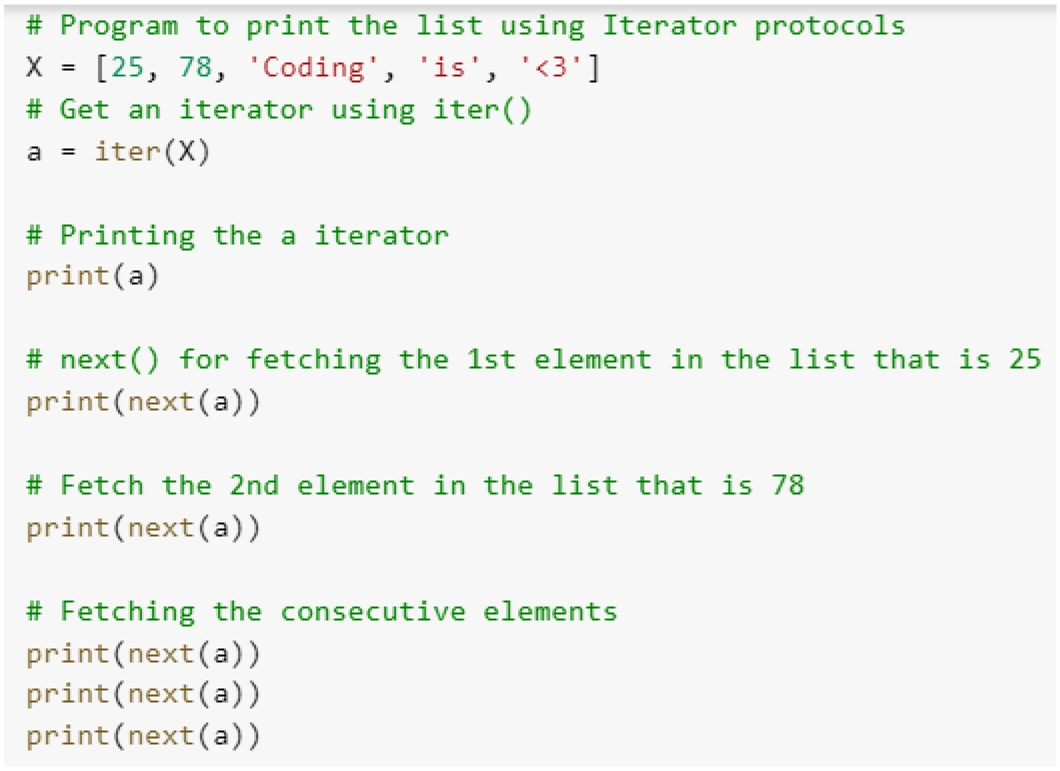
Simply iterators are used to iterate or loop over a list or tuple without using a for loop. Iterators use iter() and next() functions to iterate over the list.

iter() function takes two parameters:

- Object: An object whose iterator needs to be created (lists, sets, tuples, etc.).
- Sentinel (optional): Special value that represents the end of the sequence.

next() function returns the next item from the iterator. The next() function holds the value one at a time.
The next() method accepts two parameters:

- Iterator : next( ) function retrieves the next item from the iterator.
- default(optional): this value is returned if the iterator is exhausted (not tired, but no next item to retrieve).



```python
# Program to print the list using Iterator protocols
X = [25, 78, 'Coding', 'is', '<3']
# Get an iterator using iter()
a = iter(X)

# Printing the a iterator
print(a)

# next() for fetching the 1st element in the list that is 25
print(next(a))

# Fetch the 2nd element in the list that is 78
print(next(a))

# Fetching the consecutive elements
print(next(a))
print(next(a))
print(next(a))
```

Output:

```
<list_iterator object at 0x7f4452d6dd10>
25
78
Coding
is
<3
```

Note: if we do one more time next(a) we will get an error saying “stopiteration” because all elements in the list are over. Now to handle this error we can use try exceptions.

Ex-

```
# Program to print the tuple using Iterator protocols
tup = (87, 90, 100, 500)

# get an iterator using iter()
tup_iter = iter(tup)

# Infinite loop
while True:
   try:
       # To fetch the next element
       print(next(tup_iter),end=" ")
       # if exception is raised, break from the loop
   except StopIteration:
       break
```

```
87 90 100 500
```

Generators:
A generator is a function that produces or yields a sequence of values using a yield statement

```
# Here without Generator we are soing squaring
def square(lst):
  result=[]
  for i in lst:
    result.append(i*i)
  return result

square_list=square([1,2,3,4,5])
for i in square_list:
  print(i,end=",")
```

```
1,4,9,16,25,
# Here we are using generator as yield()
#Using generator memory will be saved & spped is fast
def square(lst):
  for i in lst:
    yield(i*i)

square_list=square([1,2,3,4,5])
for i in square_list:
  print(i,end=",")
```

```
1,4,9,16,25,
```

```
# Here we are using generator in list comprension
# ()--> is generator
square_list1=(i*i for i in [1,2,3,4,5])
for i in square_list1:
  print(i,end=",")
```

```
1,4,9,16,25,
```

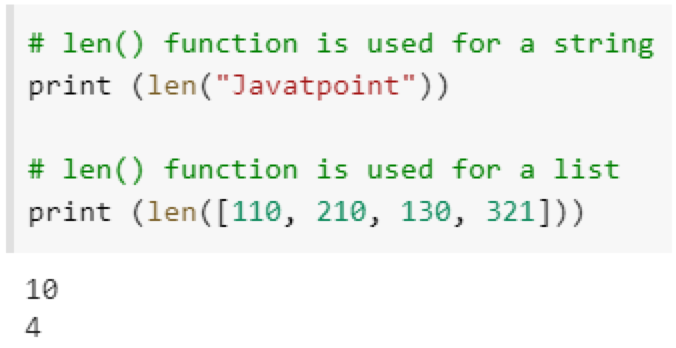
OOPS Concepts:

Polymorphism:

Polymorphism refers to having multiple forms. Polymorphism is a programming term that refers to the use of the same function name, but with different signatures, for multiple types.

Python in-built polymorphism functions:

Below if you len() is a poly-morphic function which has the same name but it is taking length of two different data types.



```python
# len() function is used for a string
print (len("Javatpoint"))

# len() function is used for a list
print (len([110, 210, 130, 321]))
```

```
10
4
```

Polymorphism with Inheritance:
Polymorphism allows us to define methods in Python that are the same as methods in the parent classes.

Inheritance:

Inheritance allows us to define a class that inherits/access all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Ex- Simply we will create a person class as a parent class with method printname() which will print firstname and lastname.now we can create another class student as child class which uses the parent class method printname() to print the student name.

we can create Person class as parent class with `firstname` and `lastname` properties, and a `printname` method:

Now we create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

```
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()
```

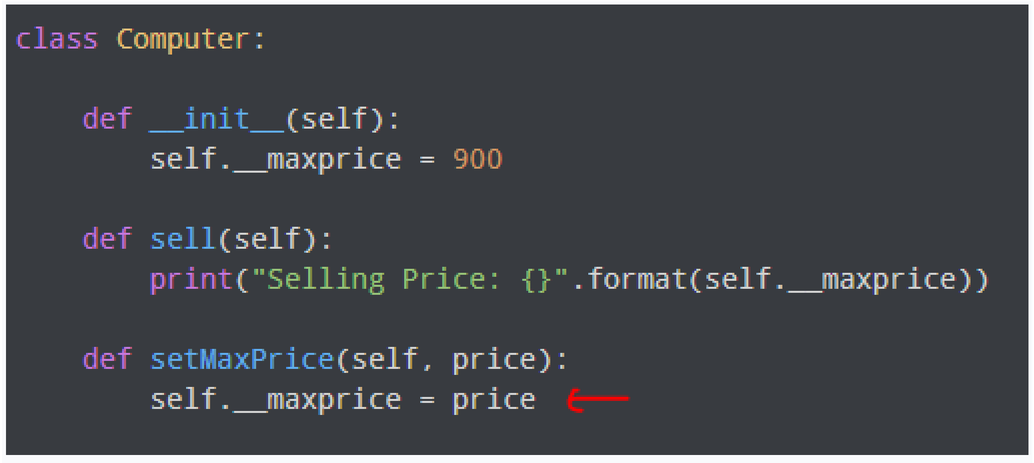
Create a class named `Student`, which will inherit the properties and methods from the `Person` class:

```
class Student(Person):
  pass
```

Use the `Student` class to create an object, and then execute the `printname` method:

```
x = Student("Mike", "Olsen")
x.printname()
```

Encapsulation:
Using encapsulation we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single _ or double __.



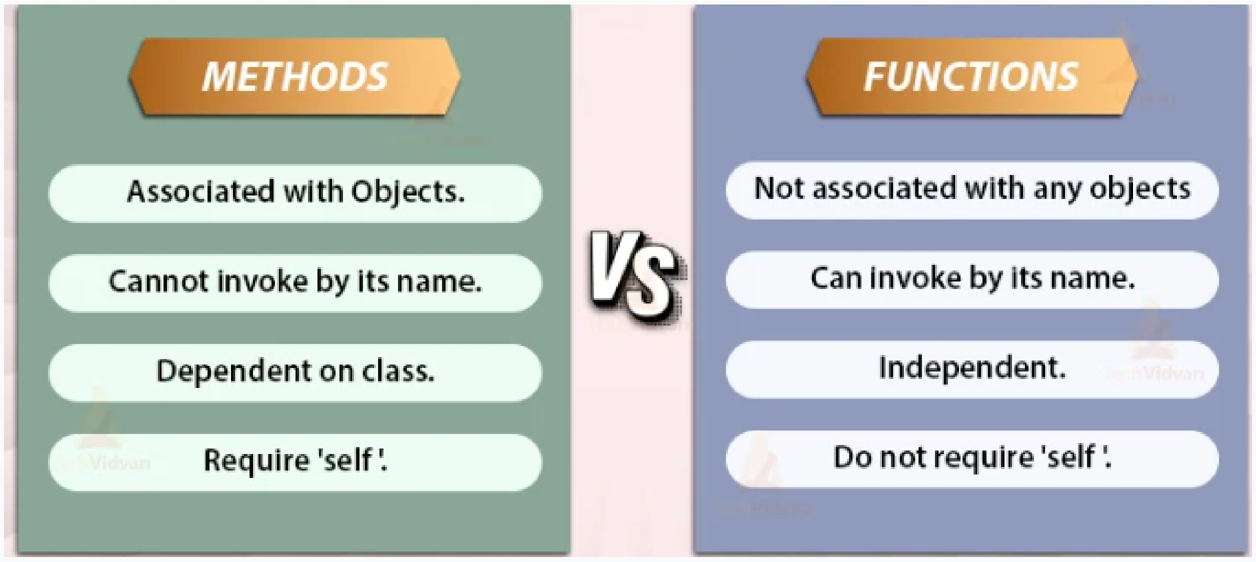
```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price
```

Functions Vs Methods:



METHODS	FUNCTIONS
Associated with Objects.	Not associated with any objects
Cannot invoke by its name.	Can invoke by its name.
Dependent on class.	Independent.
Require 'self '.	Do not require 'self '.