Polymorphism, Inheritance and Lambda Function

Polymorphism

Polymorphism is the principle that one kind of thing can take a variety of forms. In the context of programming, this means that a single entity in a programming language can behave in multiple ways depending on the context.

Operator Polymorphism

- Operator polymorphism, or operator overloading, means that one symbol can be used to perform multiple operations.
- One of the simplest examples of this is the addition operator +. Python's addition operator works differently in relation to different data types.

```
int1 = 10
int2 = 15
print(int1 + int2)
# returns 25
```

```
str1 = "10"
str2 = "15"
print(str1 + str2)
# returns 1015
```

Function Polymorphism

- Certain functions in Python are polymorphic as well, meaning that they can act on multiple data types and structures to yield different kinds of information.
- Python’s built-in len() function, for instance, can be used to return the length of an object

```
str1 = "animal"
print(len(str1))
# returns 6

list1 = ["giraffe","lion","bear","dog"]
print(len(list1))
# returns 4
```

Method Overloading

- Two or more methods have the same name but different numbers of parameters or different types of parameters, or both. These methods are called overloaded methods and this is called method overloading.

Example

```
def product(a, b):
        p = a * b
        print(p)

def product(a, b, c):
        p = a * b*c
        print(p)

product(4, 5)
product(4, 5, 5)
```

Lambda Function

- Lambda functions are similar to user-defined functions but without a name. They're commonly referred to as anonymous functions.
- Lambda functions are efficient whenever you want to create a function that will only contain simple expressions – that is, expressions that are usually a single line of a statement.
- Syntax:

```
lambda argument(s) : expression
```

Example

```
(lambda x : x * 2)(3)
>> 6

list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
filter(lambda x: x % 2 == 0, list1)

>> <filter at 0x1e3f212ad60> # The result is always filter object so I will
   need to convert it to list using list()

list(filter(lambda x: x % 2 == 0, list1))
>> [2, 4, 6, 8, 10]
```

Filter() and Map()

- In Python, iterables include strings, lists, dictionaries, ranges, tuples, and so on. When working with iterables, you can use lambda functions in conjunction with two common functions: filter() and map().
- Filter(): When you want to focus on specific values in an iterable, you can use the filter function. The following is the syntax of a filter function:

  filter(function, iterable)

- Map(): You use the map() function whenever you want to modify every value in an iterable.

```
list1 = [2, 3, 4, 5]
list(map(lambda x: pow(x, 2), list1))
>> [4, 9, 16, 25]
```

Inheritance

- Inheritance allows us to create a new class from an existing class. The new class that is created is known as subclass (child or derived class) and the existing class from which the child class is derived is known as superclass (parent or base class).
- Syntax:

```
# define a superclass
class super_class:
    # attributes and method definition
# inheritance
class sub_class(super_class):
    # attributes and method of super_class
    # attributes and method of sub_class
```

Super Keyword

```
class Emp():
def __init__(self, id, name, Add):
self.id = id
self.name = name
self.Add = Add

# Class freelancer inherits EMP
class Freelance(Emp):
def __init__(self, id, name, Add, Emails):
super().__init__(id, name, Add)
self.Emails = Emails

Emp_1 = Freelance(103, "Suraj kr gupta", "Noida" , "KKK@gmails")
print('The ID is:', Emp_1.id)
print('The Name is:', Emp_1.name)
print('The Address is:', Emp_1.Add)
print('The Emails is:', Emp_1.Emails)
```