22/17/2025

Work Done Today

- Functions

Description

Functions :

➔ Functions are block code that do specific work .
➔ Function avoids redundancy .
➔ Function avoids complexity in code
➔ Function use **def** keyword for defining it.
➔ We can call a function n number of times.

Defining a function
Syntax:

```
def function_name():
        Block of code
```

Example :

```
def display():
        print(“Hello World”)
```

Calling a function:
Syntax :

```
function_name();
```

Example:

```
display()
```

➔ In functions , a **return** statement is used for returning value or data to the code.
➔ When the computer reaches the return statement, the computer stops executing code and returns data or results to the code.
➔ If the return statement is not present then it will not return anything to the code by default .

Example :

```
def my_fun():
        return “hello !”

Msg = my_fun(); # creating a variable to accept a return value
```

```
print(Msg)

#Or
print(my_fun())  # directly printing return value
```

- ➔ Information that passed in the function as an argument.
- ➔ You can pass as much as you want to pass arguments.

Example:

```
def my_fun(name):
        print("my name is " + name)

my_fun("sayma")
my_fun("arish")
```

Parameters Vs Arguments

Example:

```
def fav_animal(animal):  # animal is parameter.
        print("my favourite animal is "+ animal)

fav_animal("Cat")  # Cat is an argument.
```

Default parameter value

Example:

```
def my_function(fname = " - "):
        print("My name is "+ fname)

my_function("Sayma")
my_function("Arish")
my_function()
```

Keyword argument

Example:

```
def my_function(std_no , name):
        print(std_no + " " + name)

my_function(std_no = 1,name = "Rohan")
```

Passing different data types

- **Passing list**

  Example:

```
def list_var(flowers):
        for flower in flowers:
            print(flower)

my_flower = [“lily” , “sunflower” , “rose”]
list_var(my_flower)
```

- **Passing dictionary**

  Example:

```
def my_fun(student):
        print(“roll no : ” + student[“no”])
        print(“Name : ” + student[“name‘])

my_std {“no” : 230 , “name” : “Sayma”}
mu_fun(my_std)
```

*args and *kwargs

➔ By default , a function should have same number number of parameter and arguments

➔ Sometimes you don't know how many arguments are passed then you can use *args and *kwargs to avoid conflict.

***args:(non keyword argument)**

➔ If you don't know how many arguments are passed in functions then you can use *args

➔ Add * before parameters this way you can access tuples of items accordingly .

Example:

```
def my_function(*name):
        print(“i’m ” + name[2])

my_function(“alice” , “bob” , “sam”) # i’m sam
```

***args with regular arguments**

Example:

```
def my_function(greeting , *name):
        for names in name:
            print(greeting , name)
```

```
my_function(“good morning” , “reshma” , “shreya”)
#good morning reshma
#good morning shreya
```

****kwargs (keyword argument):**

- ➔ If you dont know how many keywords are passed to the function then you can use **kwargs
- ➔ Add ** before parameter this way you can access dictionary items accordingly .

Example:

```
def my_function(**name):
        print(“sirname : ” + name[“lname”])

my_function(fname = “Sayma” , lname = “kazi”)
```

****kwargs with regular argument**

```
def my_function(username, **details):
  print("Username:", username)
  print("Additional details:")
  for key, value in details.items():
    print(" ", key + ":", value)

my_function("emil123", age = 25, city = "Oslo", hobby = "coding")
```

Lambda function :

- ➔ Lambda function is a small , anonymous function(no name) that is written in one line.
- ➔ It is used when you need a short function for a short time.
- ➔ No **def**
- ➔ No function name
- ➔ Only one expression
- ➔ Automatically returns the result

Syntax:
lambda argument : expression

Examples:

1) Add = lambda x , y : x + y
   print(Add(120 , 453))

2) Square = lambda y : y * y

```
print(Square(65))
```

3) Even_odd = lambda x : “Even” if x % 2 == 0 else “Odd”
   print(Even_odd(5))

4) Lambda function in another function

```
def my_fun(n):
        return lambda x : x * n

My_doubler = my_fun(2)
print(My_doubler(5))
```

Lambda function with built- function:

- **map()** - used to apply function to all elements of a list

```
My_list = [23,45,89,10,36,83]
Square =list(map(lambda x : x * x , My_list))
print(Square)
```

- **filter() -** used to filter elements based on condition

```
Numbers = [34,78,42,36,95,83,51]
Even = list(filter(lambda x : x % 2 == 0 , Numbers))
print(Even)
```

- **reduce() -** used to reduce elements in single value

```
from functools import reduce
Numbers = [45,7,22,40,16,72]
Multiply = list(reduce(lambda x , y : x * y , Numbers))
print(Multiply)
```

- **sort() -** sort elements in ascending order

```
Number = [34,56,32,56,67]
Number.sort(key=lambda x : x ) # not required
print(Number)

student(“sayma” : 99 , “bob” : 89 , “sam” : 90)
student.sort(key = lambda x : x[1])
print(student)
 # sorting elements according to marks (x[1]) = (“sayma” , 99)
```

```
student(“sayma” : 99 , “bob” : 89 , “sam” : 90)
student.sort(key = lambda x : x[0])
 # sorting elements according to names (x[0]) = (“sayma” , 99)
print(student)

student(“sayma” : 99 , “bob” : 89 , “sam” : 90)
student.sort(key = lambda x : x[1] , reverse = True)
 # sorting elements in  descending order according to marks (x[1]) = (“sayma” , 99)
print(student)
```

- **sorted() -** sorting element

```
student(“sayma” : 99 , “bob” : 89 , “sam” : 90)
new_list = sorted(student , key = lambda x : x[1])
 # sorting elements according to marks (x[1]) = (“sayma” , 99)
print(student)
```

Difference
sort() - change original list
sorted() - create new formatted list

Recursion:

➔ Recursion means function call itself

```
def eat_pizza(slice):
        if slice == 0 :
                print(“Pizza finished …”)
        else:
                print(“Eating pizza”)
                eat_pizza(slice - 1)

eat_pizza(5)
```