Date
June 3, 2021

classmate
Date
Page

WEEK 5

Functions

What is a function in Python?

In python, a function is a group of related statements that performs a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition & makes the code reusable.

SYNTAX OF A FUNCTION:

```
def function_name(parameters):
    statement(s)
```

Above shown is a function definition that consists of the following components.

1. Keyword def that marks the start of the function header.
2. A function name to uniquely identify the function.

Function naming follows the same rules of writing variables in Python.

3. Parameters (arguments) through which we pass values to a function. They are optional.

4. A colon (:) to mark the end of the function header.

5. One or more valid python statements that make up the function body. Statements must have the same indentation level.

6. An optional return statement to return a value from the function.

Example of a Function

```
def sums(a,b):
    c = a+b
    return c
```

---

* How to call a function in python?

Once we have defined a function, we can call it from another function, program or even the python prompt.

To call a function, we simply type the function name with appropriate parameters.

CODE: sums (5, 6)
print (sums)

Output: 11

---

* The return statement

The return statement is used to exit a function and go back to the place from where it was called.

SYNTAX : return [expression_list]

This statement can contain an expression that gets evaluated and the value is returned.
If there is no expression in the statement or the return statement itself is not present inside function, then the function will return the None object.

For example :

```
def greet (name):
    print ("Hello" + name + ". Morning")

print ( greet ("Paul") )
```

OUTPUT: Hello Paul. Morning
None

→ Here None is the returned value since greet() directly prints the name and no return statement is used.

Example of return :

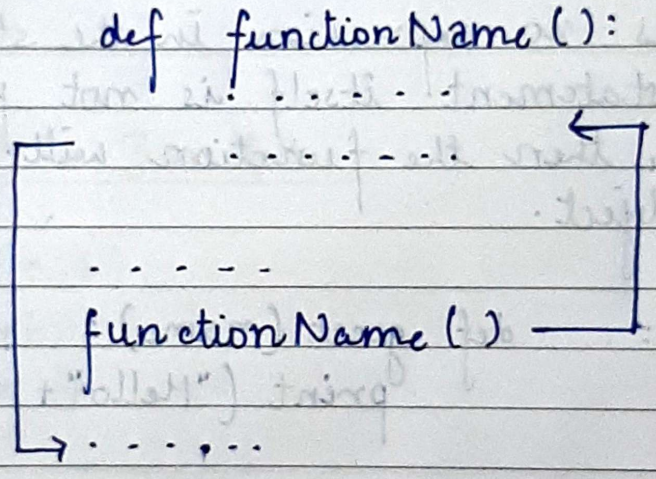
```
def absolute_num (num):
    if num >= 0:
        return num
    else:
        return -num

print (absolute_value (2))
print (absolute_value (-4))
```

OUTPUT

```
2
4
```

★ How Function Works in Python ?



★ Scope & Lifetime Of Variables

→ Scope of a variable is the portion of a program where the variable is recognized.
Parameters and variables defined inside a function are not visible from outside the function.
Hence, they have a local scope.

→ The lifetime of a variable is the period throughout which the variable exists in the memory. The lifetime of variables inside a function is as long as the function executes.

→ They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

For Example :

```
def my_func():
    x = 10
    print("Value inside the function :", x)

x = 20
my_func()
print("Value outside function:", x)
```

OUTPUT : Value inside the function : 10
Value outside function : 20

→ Here we can see that the value of [x] is 20 initially. Even though the function my_func() changed the value of x to 10, it did not affect the value outside the function.

→ This is because the variable x inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with diff. scopes.

→ On the other hand, variables outside of the function are visible from inside. They have global scope.

→ We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

Example:

```
def myFunction1():
    global x
    x = x * 2
    print("Value of x in function 1", x)

def myFunction2():
    global x
    x = x * 3
    print("Value of x in function 2", x)

x = 5
print("Value of x before function call", x)
myFunction1()
myFunction2()
print("Value of x after function call", x)
```

OUTPUT:
```
Value of x before function call 5
Value of x in function 1 10
Value of x in function 2 30
Value of x after function call 30
```

* Types Of Function Arguments

Let us start with an example :

CODE :
```
def add ( a , b ):
    ans = a+b
    return ans

add ( 4, 2)
print ( add)
```

OUTPUT : 6

→ Here the function add ( ) has two parameters.

→ Since, we have called this function with two arguments, it runs smoothly and we do not get any error.

→ If we call it with a different no. of arguments, the interpreter will show an error message.

#1 For Example :
```
add (4)
print (add)
```

OUTPUT : TypeError : add ( ) missing 1 required positional argument : 'b'

#2 Example :
```
print ( add ( ) )
```

OUTPUT : TypeError : add ( ) missing 2 required positional arguments : 'a' and 'b'

In python, we have different types of function arguments to deal with such situations.

(1) Python DEFAULT Arguments

→ Function arguments can have default values in Python.

→ We can provide a default value to an argument by using the assignment operator (=).

EXAMPLE :

```
def add(a, b=10):
    ans = a+b
    return ans

print(add(4))
print(add(2,3))
```

Output

```
14
5
```

→ In this function, the parameter 'a' does not have a default value and is required (mandatory) during a call.

→ On the other hand, the parameter 'b' has a default value of '10'. So, it is optional during a call. If a value is provided, it will overwrite the default value.

→ Any no. of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must

also have default values.

→ This means to say, non-default arguments cannot follow default arguments.

EXAMPLE:

```
def add ( a=10 , b ):
    ans = a+b
    return ans
```

→ We would get an error as:

Syntax Error: non-default argument follows default argument

(2) Python KEYWORD Arguments

→ When we call a function with some values, these values get assigned to the arguments according to their position.

→ For example, in above function add() , when we called it as add (4, 2) , the value '4' gets assigned to the argument a and similarly 2 to b .

→ Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can

be changed.

→ Following calls to the above function are all valid and produce the same result.

CODE:

```
#2 keyword arguments
add (a=2, b=4)

# 2 keyword arguments (out of order)
add (b=9, a=3)

# 1 positional, 1 keyword argument
add (4, b=2)
```

→ As we can see, we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional args.

→ Having a positional argument after keyword arguments will result in errors.

For example: add (a=4, 2)

Will result in an error:

SyntaxError: non-keyword arg after keyword arg

* TYPES OF FUNCTIONS

(1) In-Built Functions

print(), input(), len()

(2) Library Functions

log(), sqrt(), random(), randrange(), calendar(), month()

(3) String Methods (functions)

upper(), lower(), strip(), count(), index(), replace()

(4) User-Defined functions

```
def square(x):
    sqr = x**2
    return sqr

print(square(5))
```

Date
June 4, 2021

classmate
Date
Page

Functions : Examples

```
# function to find minimum in list

def list_min (l):
    mini = l[0]
    for i in range (len(l)):
        if (l[i] < mini):
            mini = l[i]
    return mini

# function to find maximum in list

def list_max (l):
    maxi = l[0]
    for i in range (len(l)):
        if (l[i] > maxi):
            maxi = l[i]
    return maxi

# function to append a list before given list

def list_appendbefore (l, z):
    newl = []
    for i in range (len(z)):
        newl.append(z[i])
    for i in range (len(l)):
        newl.append (l[i])
    return newl
```

```
# function to append a list after given list

def list_appendEnd (l, z):
    newl = []
    for i in range (len(l)):
        newl.append (l[i])
    for i in range (len(z)):
        newl.append (z[i])
    return newl

# function to calculate average of all list elements

def list_average (l):
    sum = 0
    for i in range (len(l)):
        sum = sum + l[i]
    ans = sum / len(l)
    return ans
```

```
# let us call our functions

l = [1, 8, 4, 3, 2, 9]
z = [10, 16, 20]

print ( list_min (l))
print ( list_max (l))
print (list_appendbefore (l, z))
print (list_appendEnd (l, z))
print (list_average (l))
```

OUTPUT

```
1
9
[10, 16, 20, 1, 8, 4, 3, 2, 9]
[1, 8, 4, 3, 2, 9, 10, 16, 20]
4.5
```

Date
June 5, 2021

classmate
Date
Page

Sorting Using Functions

```
# find out the minimum most element in the list l
# append that to a new list x
# remove the minimum from the original list l

# function to find minimum
def min_list(l):
    mini = l[0]
    for i in range(len(l)):
        if (l[i] < mini):
            mini = l[i]
    return mini

# sort function
def obvious_sort1(l):
    x = []
    while (len(l) > 0):
        mini = min_list(l)
        x.append(mini)
        l.remove(mini)
    return x


# We just learnt that breaking our problem into
# smaller modules and solving them makes it easy
# on our mind.
```

June 6, 2021

Matrix Multiplication
USING FUNCTIONS

```
# initialise C to zero
# need to consider two matrices A and B
# need to find dot product of two lists
# need to pick ith row & jth column in a matrix.

def initialize_mat (dim):
    C=[]
    for i in range (dim):
        C.append([])
    for ii in range (dim):
        for j in range (dim):
            C[i].append (0)
    return C

def dot_product (u,v):
    dim = len(u)
    ans = 0
    for i in range (dim):
        ans = ans +(u[i] * v[i])
    return ans

def row (M,i):
    dim = len (M)
    l = []
    for k in range (dim):
        l.append (M[i][k])
    return l
```

```
def column(M, j):
    dim = len(M)
    l = []
    for k in range(dim):
        l.append(M[k][j])
    return l

def mat_mul(A,B):
    dim = len(A)
    C = initialize_mat(dim)
    for i in range(dim):
        for j in range(dim):
            C[i][j] = dot_product(row(A,i), column(B,j))
    return C
```

Date June 7, 2021

Recursion

WHAT IS RECURSION?

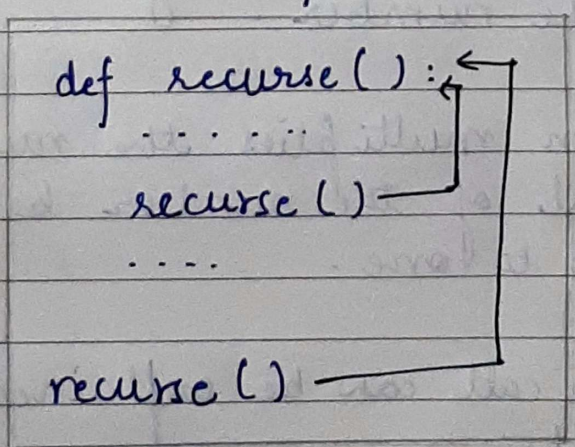
Recussion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other! Any object in between them would be reflected recursively.

PYTHON RECURSIVE FUNCTION

In python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

This is how recursive function recurse works.



Following is an example of a recursive functions to find factorial of an integer.

Factorial = n (n-1) (n-2) .... 3 x 2 x 1

CODE:

```
def factorial (x):
    if x == 1:
        return 1
    else
        return ( x* factorial(x-1))

num = 3
print ("The factorial of", num, "is", factorial (num))
```

OUTPUT: The factorial of 3 is 6

→ In the above example, factorial() is a recursive function as it calls itself.

→ When we call this function with a positive integer, it will recursively call itself by decreasing the number.

→ Each function multiplies the number with the factorial of the number below it until it is equal to one.

→ This recursive call can be explained in the following steps:

factorial(3)	# 1st call with 3
3 * factorial(2)	# 2nd call with 2
3 * 2 * factorial(1)	# 3rd call with 1
3 * 2 * 1	# returns from 3rd call as 1
3 * 2	# returns from 2nd call
6	# returns from 1st call

→ Our recursion ends when the number reduces to 1. This is called base condition.

→ Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

→ The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

→ By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in RecursionError.

Advantages Of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.

3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages Of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory & time.
3. Recursive functions are hard to debug.
