

Computational Thinking and Programming - 2

Functions

Functions:

types of function

(built-in functions, functions defined in module, user defined functions),

creating user defined function,

arguments and parameters, default parameters, positional parameters,

function returning value(s), flow of execution,

scope of a variable (global scope, local scope)

Definition

Large programs are often difficult to manage, thus they are broken down into smaller units which are called **functions**.

A **function** is simply a group of statements under any name (known as the function name) which can be invoked from other parts of the program. It contains line of code(s) that are executed sequentially from top to bottom by Python Interpreter

Advantages of using functions

1. **Code reusability** : Suppose we are writing an application in Python where we need to perform a specific task in several places of our code, assume that we need to write 10 lines of code to do that specific task. It would be better to write those 10 lines of code in a function and just call the function wherever needed, because writing those 10 lines every time you perform that task is tedious, it would make your code lengthy, less-readable and increase the chances of human errors.
2. **Improves Readability:** By using functions for frequent tasks you make your code structured and readable. It would be easier for anyone to look at the code and be able to understand the flow and purpose of the code.

Advantages of using functions

3. **Avoid redundancy** : When you no longer repeat the same lines of code throughout the code and use functions in places of those, you are actually avoiding the redundancy that you may have created by not using functions.
4. **Improves maintenance** : Since the use of functions make the program compact, it becomes easier to maintain it. Testing and correcting errors is easy because errors are localized and can be corrected easily

Types of Functions

There are three types of functions in Python :

- 1. Built - in functions** : They are the predefined functions that are already available in Python and we need not declare these functions before using them. We can freely invoke them as and when needed. For example, `print()` is a built in function.
- 2. Functions defined in modules** - Some functions are defined in the modules which needs to be imported in the program before they can be used. For example, `randint()` is a predefined function in random module.
- 3. User defined functions:** The functions which the programmer creates in the code are user-defined functions. We will now learn user defined functions.

User Defined Functions

Using a user defined function involves two steps :

- 1. Writing the function definition :** The function definition means writing the statements that are responsible for performing the task and assigning them a function name.
- 2. Invoking / calling a user defined function :** It means executing the statements given within the function by writing a function call statement at the location where these statements need to be executed.

Defining a Function

To define a function, Python provides the keyword **def**.



Function block begins with the keyword **def** followed by the function name and parentheses **()**. This is called the **function header**. The executable statements written after the function header are called the **function body**.

User Defined Functions

Syntax:

```
def <FunctionName> ([parameter 1, parameter 2, .....]):  
    set of instructions to be executed  
    [return <value>]
```

Function header

Function Body (set of statements should be indented within the function header.)

- The items enclosed with parentheses are called parameters, they are optional.
- Function header ends with a colon (:)
- Function name should be unique. Rules for naming a function are same as rules for naming an identifier.
- The statements outside the function header are not considered as part of the function body.

Components of a Function

- Keyword **'def'** marks the start of Function header
- The function name is an identifier which uniquely identifies the function.
- The parameters refer to the data items which are passed to the function to work on it.
- The colon **:** marks the end of Function header which also means it requires a block of code.
- One or more Python statements are given which defines the action that is to be performed by the function
- The optional return statement returns the calculated result
- A return statement with no arguments is the same as return None.

Parameters and Arguments

Parameters are the value(s) provided in the parentheses when we write a function header. These are the values required by a function to work. If there are more than one parameter, they must be separated by a comma(,).

Parameters are also called **Formal arguments / parameters**.

An **Argument** is a value that is passed to the function when it is called. In other words arguments are values provided during the function call statement. They are also called **Actual arguments / parameters**.

The argument passed in the function call statement is received in the corresponding parameter given in the function header. The arguments given in function call statement must match the number and order of parameters in function definition. This is called **Positional Argument Matching**.

Execution of a Program using Functions

- Flow of control refers to the order in which the statements are executed during program execution.
- When Python encounters a function definition, Python just executes the function header and checks it's correctness.
- Execution always begins from the first statement of the `_main_` segment
- When a function call statement is encountered, the control jumps to the called function and executes the statement given in that function.
- With the return statement or the last statement of the function the control comes back to the point where it is called.
- The functions are always defined at the top followed by the statements that are not the part of any of the functions.
- The statements inside a function body are not executed till the function is invoked.

Example 1

A function with no arguments, no parameters and no return value

CODE 1:

```
def welcome():  
    print("Welcome to Class XII")  
    print("This is the Computer Science class")  
welcome()
```

Function header (points to `def welcome():`)

no parameters (points to the empty parentheses in the header)

no arguments (points to the empty parentheses in the call)

Function call (points to `welcome()`)

OUTPUT:

```
Welcome to Class XII  
This is the Computer Science class
```

Example 2

CODE 1:

A function which takes in arguments and returns a value.

```
def add(a,b):  
    print("Value 1 : ", a)  
    print("Value 2 : ", b)  
    return (a+b)  
Number1 = int(input("Enter the number: "))  
Number2 = int(input("Enter the number: "))  
print(Number1, "+", Number2, "=", add(Number1, Number2))
```

a and b are parameters

Function Definition

arguments

OUTPUT:

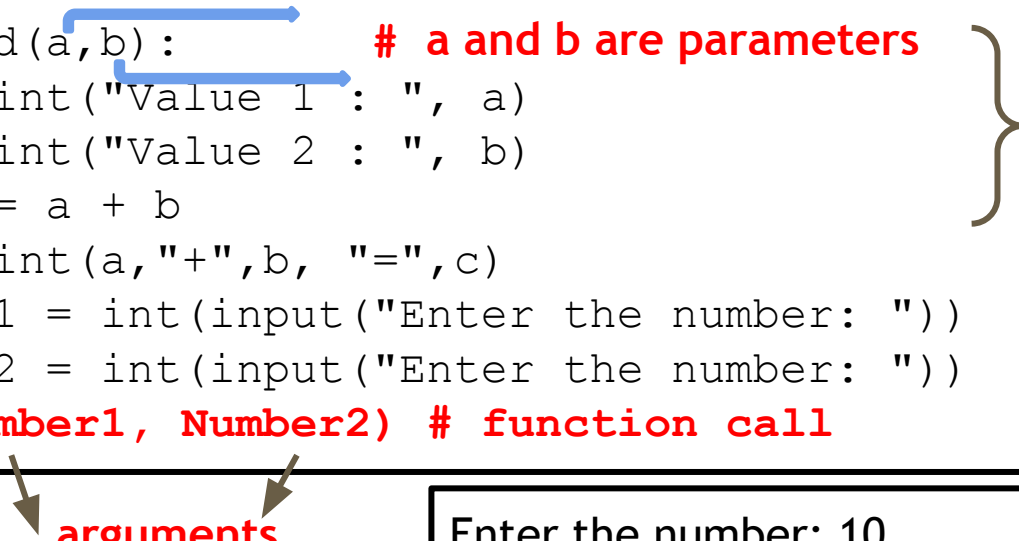
```
Enter the number: 10  
Enter the number: 20  
Value 1 : 10  
Value 2 : 20  
10 + 20 = 30
```

Example 3

CODE 1:

A function which takes in arguments & returns no value.

```
def add(a,b):           # a and b are parameters
    print("Value 1 : ", a)
    print("Value 2 : ", b)
    c = a + b
    print(a,"+",b, "=",c)
Number1 = int(input("Enter the number: "))
Number2 = int(input("Enter the number: "))
add(Number1, Number2) # function call
```



Function
definition

arguments

OUTPUT:

```
Enter the number: 10
Enter the number: 20
Value 1 : 10
Value 2 : 20
10 + 20 = 30
```

Formal arguments and actual arguments

```
import math
def area(r):
    a = math.pi * r * r
    return a

n = int(input("Enter Radius "))
ar = area(n)
print("Area of Circle = ", ar)
```

FORMAL ARGUMENT

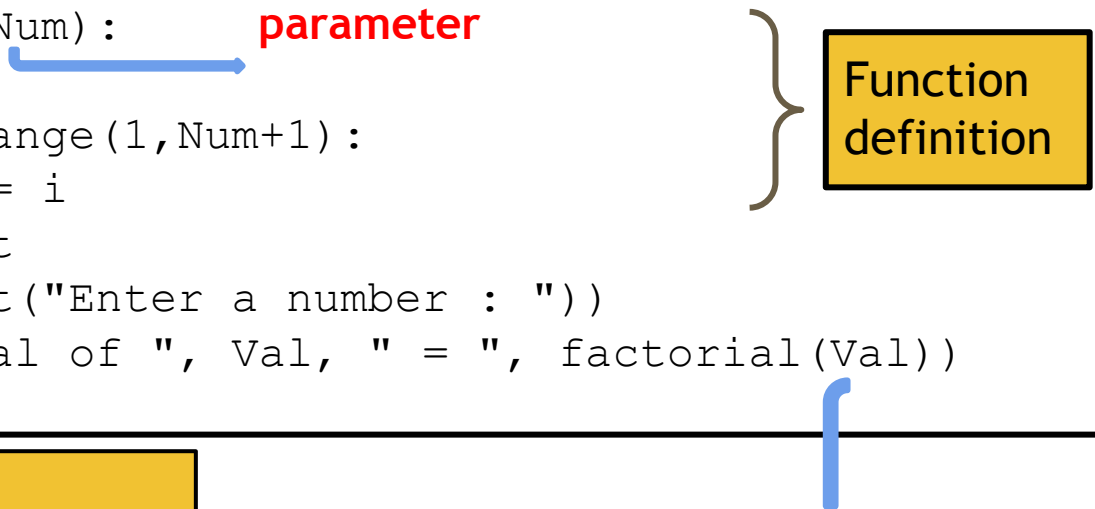
ACTUAL ARGUMENT

Example 4

CODE 1:

A function which takes in arguments & returns a value.

```
def factorial(Num):  
    fact = 1  
    for i in range(1, Num+1):  
        fact *= i  
    return fact  
Val = int(input("Enter a number : "))  
print("Factorial of ", Val, " = ", factorial(Val))
```



parameter

Function definition

arguments

OUTPUT:

```
Enter a number : 6  
Factorial of 6 = 720
```

Example 5:

Write a function `table(n)` which displays the table of an integer number `n`. Also write the main segment to invoke the function.

CODE:

```
def table(n):  
    for i in range(1,11):  
        print(n, " * ", i, " = ", n*i)  
Num = int(input("Enter a number : "))  
table(Num)
```

OUTPUT:

Enter a number : 6

```
6 * 1 = 6  
6 * 2 = 12  
6 * 3 = 18  
6 * 4 = 24  
6 * 5 = 30  
6 * 6 = 36  
6 * 7 = 42  
6 * 8 = 48  
6 * 9 = 54  
6 * 10 = 60
```

Example 6:

Write a function which takes in an integer x and n, returns the sum of the series:

$$1 - x^2 + x^3 - +.....(-)^2 x^{n+1}$$

```
def SumSeries(x,n):  
    Sum = 1  
    for i in range(2,n+1):  
        Sum += pow(-1,i+1)*pow(x,i)  
    return Sum  
  
X = int(input("Enter X: "))  
N = int(input("Enter N: "))  
  
print(SumSeries(X,N))
```

CODE:

OUTPUT:

Enter X: 2
Enter N: 3
5

Example 7:

Write a program that uses a function `sum_digits()` to find the sum of the digits of the number passed as an argument to the function.

```
def sum_digits(n) :  
    Sum = 0  
    while (n > 0):  
        digit = n % 10  
        Sum += digit  
        n = n // 10  
    return Sum  
N = int(input("Enter a number : "))  
print("Sum of digits of ", N, " = ", sum_digits(N))
```

CODE:



OUTPUT:

```
Enter a number : 654  
Sum of digits of 654 = 15
```

Example 8:

Write a program that uses a function `display_grade(percent)` to display the grade of a student where `percent` is passed to the function.

```
def display_grade(percent):  
    if percent >= 90:  
        print("Percent : ", percent, "Grade A")  
    elif percent >= 80:  
        print("Percent : ", percent, "Grade B")  
    elif percent >= 70:  
        print("Percent : ", percent, "Grade C")  
    elif percent >= 50:  
        print("Percent : ", percent, "Grade D")  
    else:  
        print("Percent : ", percent, "Grade E")  
display_grade(83)  
display_grade(65)  
display_grade(45)
```

CODE:



OUTPUT:

```
Percent : 83 Grade B  
Percent : 65 Grade D  
Percent : 45 Grade E
```

Example 9 :

Write a function Sequence(Start, Stop, Step) and displays the sequence of numbers.

CODE:

```
def Sequence(Start, Stop, Step) :  
    for i in range(Start, Stop+1, Step) :  
        print(i, end="*")  
    print()
```

Sequence(2, 30, 4)

Sequence(5, 50, 5)

OUTPUT:

```
2*6*10*14*18*22*26*30*  
5*10*15*20*25*30*35*40*45*50*
```

Immutable Arguments to a function (Integer)

```
def Change(A,B):  
    print("A= ", A, " B = ", B)  
    A+=10  
    B-=5  
    print("A= ", A, " B = ", B)  
X,Y = 10,20  
print(X,Y)  
Change(X,Y)  
print(X,Y)
```

10 20

A= 10 B = 20

A= 20 B = 15

10 20

Mutable Arguments to a function (List)

```
def Change(A):  
    print("A= ", A)  
    A[0] += 10  
    A[1] -= 5  
    print("A= ", A)  
X = [10, 20]  
print(X)  
Change(X)  
print(X)
```

[10, 20]

A= [10, 20]

A= [20, 15]

[20, 15]

Immutable Arguments to a function (Tuple)

```
try:
    def Change(A):
        print("A= ", A)
        A[0]+=10
        A[1]-=5
        print("A= ", A)
    X = (10,20)
    print(X)
    Change(X)
    print(X)
except TypeError:
    print("TypeError occurred")
```

(10, 20)

A= (10, 20)

TypeError occurred

Immutable Arguments to a function (Tuple)

```
try:
    def Change(A):
        print("A= ", A)
        A = list(A)
        A[0] += 10
        A[1] -= 5
        print("A= ", A)
    X = (10, 20)
    print(X)
    Change(X)
    print(X)
except TypeError:
    print("TypeError occurred")
```

```
(10, 20)
A= (10, 20)
A= [20, 15]
(10, 20)
```

Mutable Arguments to a function (Dictionary)

```
def Change(D):
```

CODE:

```
    D['Salary']=D['Salary'] +10000
```

```
    print(D)
```

```
D1 = {'Name': 'Arsh Gupta', 'Dept': 'Marketing', 'Salary': 60000}
```

```
D2 = {'Name': 'Arnav Gupta', 'Dept': 'Finance', 'Salary': 50000}
```

```
Change(D1)
```

```
Change(D2)
```

```
print(D1)
```

```
print(D2)
```

OUTPUT:

```
{'Name': 'Arsh Gupta', 'Dept': 'Marketing', 'Salary': 70000}
```

```
{'Name': 'Arnav Gupta', 'Dept': 'Finance', 'Salary': 60000}
```

```
{'Name': 'Arsh Gupta', 'Dept': 'Marketing', 'Salary': 70000}
```

```
{'Name': 'Arnav Gupta', 'Dept': 'Finance', 'Salary': 60000}
```

Positional Parameters

Positional (or required) arguments - Values that are passed into a function based on the order in which the parameters were listed during the function definition. Here, the order is especially important as values passed into these functions are assigned to corresponding parameters based on their position.

For example, if a **function header** is like :

```
def check(a, b, c) :
```

Then possible **function calls** for this can be

`check (x, y, z)` # Call 1 (a gets value of x, b gets value of y, c gets value of z)

`check (2, x, y)` # Call 2 (a gets value 2, b gets value of x, c gets value of y)

`check (2, 5, 7)` # Call 3 (a gets value 2, b gets value 5, c gets value 7)

Positional Parameters

In this method of parameter passing, the arguments must be provided for all the parameters (required).

Also the number of arguments in the function call statement must match with the number of parameters given in the header statement of function definition.

No value can be skipped from the function call and the order cannot be changed.

Default Parameters

Python allows us to assign default values to a function's parameters which are useful in case a matching argument is not passed in the function call statement. The default values are specified in the **function header** statement.

A parameter having a default value in the function header becomes optional in the function call statement.

Any parameter cannot have a default value unless all the parameters appearing on its right have default values.

The default values of the parameters are considered only when no value is provided for that parameter in the function call statement.

Default Parameters

Program to input the radius of the circle and then display its circumference

```
def circumcircle(r, pi=3.14):  
    circum=2*pi*r  
    print("Circumference:", circum)  
rad=float(input("Enter radius"))  
circumcircle(rad)  
p=3.14159  
circumcircle(rad, p)
```

Any parameter can have a default value only if all parameters on its right have default values

Since the actual argument for the parameter *pi* is missing, the default value of *pi* assigned is used.

The value of *p=3.14159* is passed as an argument to the function and the passed value is used.

Default Parameters

Ways of calling a function with default arguments

```
def circumcircle(r=5, pi=3.14) :  
    circum=2*pi*r  
    print("Circumference:", circum)
```

```
rad=float(input("Enter radius"))  
p=float(input("Enter pi value"))  
circumcircle(rad,p)           # Statement 1  
circumcircle(rad)             # Statement 2  
circumcircle()                # Statement 3
```


Default Parameters

Some examples of function headers with default values :

<code>def interest(prin, time, rate=0.10):</code>	Valid
<code>def interest(prin, time=2, rate):</code>	Invalid
<code>def interest(prin=2000, time=2, rate):</code>	Invalid
<code>def interest(prin, time=2, rate=0.10):</code>	Valid
<code>def interest(prin=2000, time=2, rate=0.10):</code>	Valid
<code>def interest(prin=2000, time, rate=0.10):</code>	Invalid

Using different arguments - Positional, Keyword and Default arguments

```
def area_perimeter(width=2, height=2):
```

```
    area = width * height
```

```
    perimeter = (2 * width) + (2 * height)
```

```
    print("Area = " + str(area) + " and Perimeter = " + str(perimeter))
```

Default Parameters

```
def Calc(U):  
    if U % 2 == 0:  
        return U + 1  
    else:  
        return U*2  
def Pattern(M, B=2):  
    for i in range(0, B +1):  
        print(Calc(i), M,end='')  
    print()  
Pattern('*')  
Pattern('#',4)  
Pattern('@',3)
```

Default Parameters

```
def Calc(U):  
    if U % 2 == 0:  
        return U + 1  
    else:  
        return U*2  
  
def Pattern(M, B=2):  
    for i in range(0, B +1):  
        print(Calc(i),M,end=' ')  
    print()  
  
Pattern('*')  
Pattern('#',4)  
Pattern('@',3)
```

OUTPUT:

1 *2 *3 *

1 #2 #3 #6 #5 #

1 @2 @3 @6 @

Give the output of the following code:

```
def Alter(P=15,Q=10):  
    P=P*Q  
    Q=P/Q  
    print(P,"#",Q)  
    return Q  
  
A=100  
B=200  
A=Alter(A,B)  
print(A,"$",B)  
B=Alter(B)  
print(A,"$",B)  
A=Alter(A)  
print(A,"$",B)
```

OUTPUT:

20000 # 100.0

100.0 \$ 200

2000 # 200.0

100.0 \$ 200.0

1000.0 # 100.0

100.0 \$ 200.0

Give the output of the following code:

```
def Alter (P=15,Q=10) :  
    P=P*Q  
    Q=P/Q  
    print (P,"#",Q)  
    return Q  
A=100  
B=200  
A=Alter (A,B)  
print (A,"$",B)  
B=Alter (B)  
print (A,"$",B)  
A=Alter (A)  
print (A,"$",B)
```

Give the output of the following code:

```
def ChangeVal(M,N):  
    for i in range(N):  
        if M[i]%5 == 0:  
            M[i] //= 5  
        if M[i]%3 == 0:  
            M[i] //= 3  
  
L=[25,8,75,12]  
ChangeVal(L,4)  
for i in L :  
    print(i, end='#')
```

Give the output of the following code:

```
def ChangeVal(M,N):  
    for i in range(N):  
        if M[i]%5 == 0:  
            M[i] //= 5  
        if M[i]%3 == 0:  
            M[i] //= 3  
  
L=[25,8,75,12]  
ChangeVal(L,4)  
for i in L :  
    print(i, end='#')
```

OUTPUT

5#8#5#4#

Give the output of the following code:

```
def Call(P=40,Q=20):  
    P=P+Q  
    Q=P-Q  
    print(P,'@',Q)  
    return P  
R=200  
S=100  
R=Call(R,S)  
print (R,'@',S)  
S=Call(S)  
print(R,'@',S)
```

Give the output of the following code:

```
def Call(P=40,Q=20):  
    P=P+Q  
    Q=P-Q  
    print(P,'@',Q)  
    return P  
  
R=200  
S=100  
R=Call(R,S)  
print(R,'@',S)  
S=Call(S)  
print(R,'@',S)
```

OUTPUT:

300 @ 200

300 @ 100

120 @ 100

300 @ 120

Returning values from functions

Python functions can be categorised into two categories :

★ Functions returning some values (Non-void functions)

★ Functions not returning any value (Void functions)

Non Void Functions

Non void functions return some value to the calling program using the statement **return <value>**

The value being returned can be

- ❖ A literal
- ❖ A variable
- ❖ An expression

- ❖ Some valid return statements are :
- ❖ `return 5` `return 6+5`
- ❖ `return a` `return a**3`
- ❖ `return (a+3**2)/b` `return a+b+c`

The return statement ends the execution of the function even if it is in the middle of the function.

A function may have multiple return statements but only one is executed.

Non Void Functions

When a **non void function** is called, the returned value is made available to the caller function / program by internally substituting the function call statement. For example :

```
def sum(x,y) :  
    s=x+y  
    return s  
  
result=sum(5, 10)  
print (sum(10,20))  
A=sum(10,20)+sum(30,40)
```

The function call to the functions that return a value can be written on the right hand side of an assignment statement, a part of print statement or in an expression.

Void Functions

The functions that perform some action or do some work but do not return any computed value or final value to the caller are called void functions. A void function may or may not have a return statement. If the void function has a return statement then it is used only to transfer the control to the calling function.

The void functions are not used inside a statement or expression in the caller program.

Every void function returns value **None** to its caller.

Void Functions

```
def welcome ( ) :  
    print("Hello")  
welcome( )  
print(welcome())
```

```
Hello  
Hello  
None
```

The first statement of the main program i.e. call to the the function will give the output **Hello**.

The second statement invokes the function first giving the output **Hello** and the **returns None** to the calling program. The **print** Statement then prints **None** on the screen.

Functions returning more than one value

Python allows you to return more than one value from the function to the calling program. To return multiple values :

- The return statement inside the function body should have the form :
return <value1/variable1/expression1>, <value2/variable2/expression2>,,
- The function call statement should receive or use the returned values as:
 - either receive the returned values in form of tuple variable
 - directly unpack the received values of the tuple by specifying the same number of variables on the left hand side of the assignment in function call.

Returning multiple values

Receiving the returned values in form of tuple variable

```
def squared (x , y , z ):  
    return x * x , y * y , z * z
```

```
t=squared (2,3,4)  
print (t)  
t1,t2,t3 = squared(10,20,30)  
print(t1,t2,t3)
```

Returning a tuple (any comma separated values is a tuple)

Value returned from the function assigned to a tuple

Value returned from the function is assigned to three objects (int variables)

Function to return income tax and allowance

Write a program to input the name and basic salary. Include a function, `CalcSalary(B)` which takes in the Basic Salary and calculates the DA and HRA as per the following table. The function returns the DA and HRA.

BASIC	DA	HRA
Upto 10000	80%	20%
10001-20000	85%	25%
Above 20001	90%	30%

The standard deduction of PF (Provident Fund) is 12%

Calculate the Net Salary = Basic Salary + DA + HRA - PF

Function returning multiple values

```
def CalcSalary(B):  
    if B >= 20001:  
        DA = 0.9 * B ; HRA = 0.3 * B  
    elif B >= 10001:  
        DA = 0.85 * B ; HRA = 0.25 * B  
    else:  
        DA = 0.8 * B ; HRA = 0.2 * B  
    return DA,HRA  
Name=input("Enter your name : ")  
Basic = int(input("Enter your salary : "))  
DA,HRA = CalcSalary(Basic)  
PF = 0.12 * Basic  
NetSalary = Basic + DA + HRA - PF  
print("Hello ", Name)  
print("Your salary is ", NetSalary)
```

```
Enter your name : Akshay Rawat  
Enter your salary : 200050  
Hello Akshay Rawat  
Your salary is 416104.0
```

Returning multiple values

Write a program to input total seconds and then convert these seconds to its equivalent hours, minutes and seconds.

```
def convertsec(sec):  
    hr=sec//3600  
    sec=sec%3600  
    min=sec//60  
    sec=sec%60  
    return hr,min,sec  
seconds=int(input("Enter seconds value "))  
h,m,s= convertsec(seconds)  
print("In ", seconds, " seconds", end = ' ')  
print("there are ")  
print(h, " hours")  
print(m, " minutes")  
print(s, "seconds")
```

```
Enter seconds value 12000  
In 12000 seconds there are  
3 hours  
20 minutes  
0 seconds
```

Input a string and perform toupper(), tolower() and isalnum()

```
def chupper(str):
    newstr=""
    for ch in str:
        if ch >="a" and ch<="z":
            newstr+=chr(ord(ch)-32)
        else:
            newstr+=ch
    print(newstr)
def chlower(str):
    newstr=""
    for ch in str:
        if ch >="A" and ch<="Z":
            newstr+=chr(ord(ch)+32)
        else:
            newstr+=ch
    print(newstr)
```

```
def chkisalnum(str):
    valid=True
    for ch in str:
        if not( ch >= 'A' and ch<='Z' or ch
        >= 'a' and ch<='z' ):
            if not (ch>='0' and ch<='9'):
                valid=False
    print(valid)
ch=int(input("1:upper\n2:lower\n3:check
isalnum\nEnter choice"))
s=input("Enter string")
if ch==1:
    chupper(s)
elif ch==2:
    chlower(s)
elif ch==3:
    chkisalnum(s)
else:
    print("Wrong choice")
```

Scope of a Variable/Object

The scope of an object refers to the part or a segment of the program in which it is accessible. There are broadly two kinds of scopes in python :

1. **Global scope** : An object/variable declared in top level segment (`_main_`) of a program is said to have a global scope and it is usable inside the whole program and all blocks contained within the program.
2. **Local scope** : An object/variable declared inside the function body is said to have a local scope i.e. it can be used only within that function in which it is defined.

Lifetime of an Object

The time period for which the object lives in memory is called the **lifetime of the object**.

Global objects have the lifetime of **the entire program** run. They remain in the memory as long as the program is being executed.

Local objects live in the memory as **long as the function in which they are defined is being executed**. Once the function terminates, local objects lose their lifetime.

Global and Local Objects

```
B = 200 # Global object
def fun():
    A = 10 # Local object
    print("Function A : ", A)
    print("In Fun B : ", B)

print("Main B : ", B)
fun()
print("Main A : ", A)
```

OUTPUT:

```
Main B : 200
Function A : 10
In Fun B : 200
NameError
```


Scope of a Variable

```
id = 25
def first():
    id = 35
    print("Function First", id)
    def second():
        id = 45
        print("Function Second ", id)
    second()
first()
print("Main ", id)
```

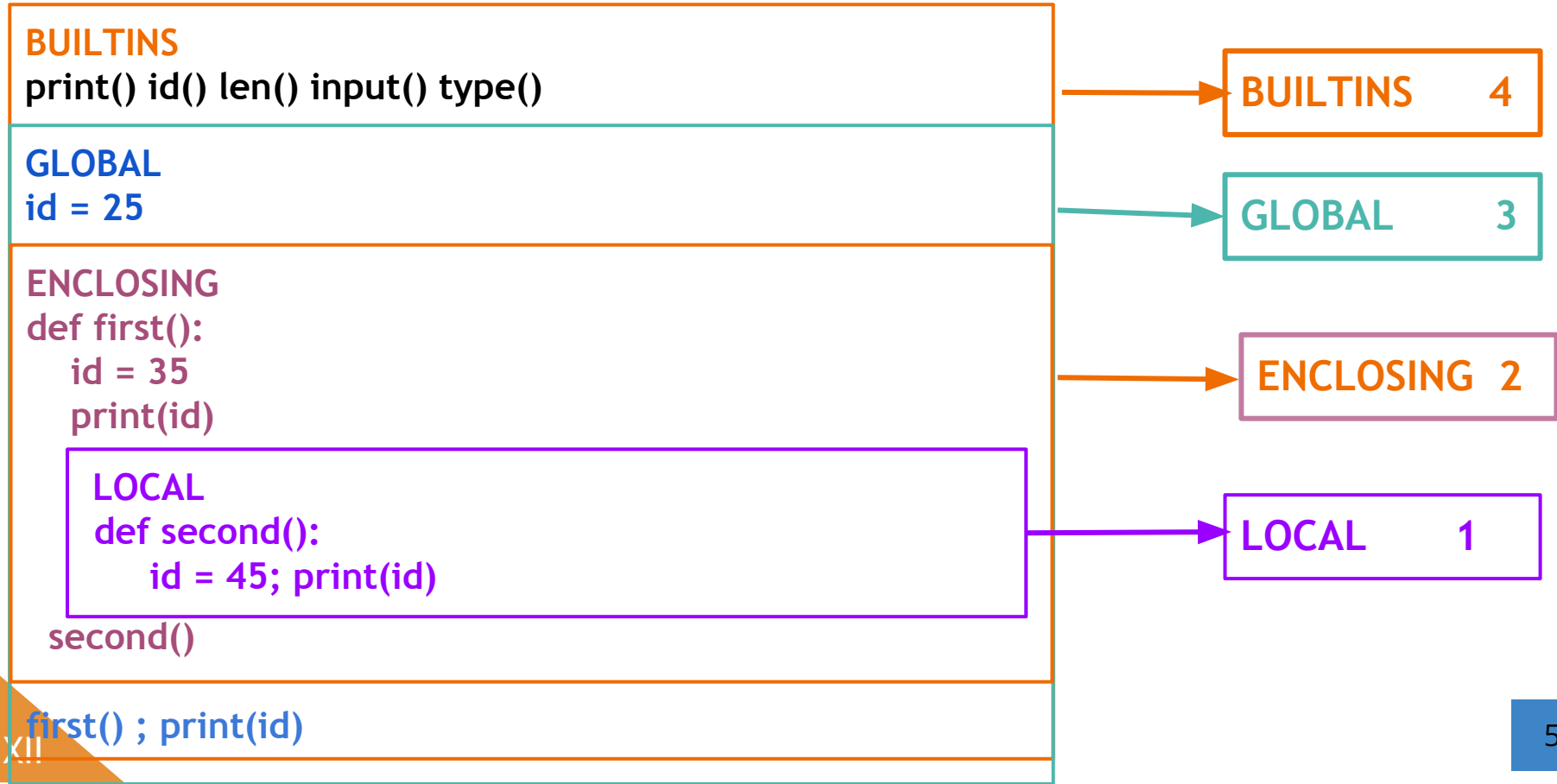
```
Function First 35
Function Second 45
Main 25
```

Resolving scope of a name (LEGB rule)

Whenever any variable is accessed in a program, Python follows a name resolution rule called **LEGB rule** i.e. whenever a name is referenced, Python :

- i) checks within its **Local environment**, if it has a variable with the same name, its value is used, else
- ii) Python then checks in its **Enclosing environment**. If present, Python uses its value otherwise repeats this step to higher level enclosing environments. Else
- iii) Python now checks in the **Global environment**. If present, Python uses its value otherwise
- iv) Python checks its **Built-in environment** that contains all built in variables and functions. If found, Python uses its value otherwise reports the error

Scope resolution via LEGB rule :



Global Keyword

Global keyword is a keyword that allows a user to modify a variable outside of the current scope. **Global keyword** is used inside a function only when we want to do assignments or when we want to change a variable. Global is not needed for printing and accessing.

To access a global variable inside a function there is no need to use global keyword.

```
a = 15    # global variable
b = 10
def add():
    c = a + b
    print(c)
add()
```

Modify global immutable variable w/o GLOBAL

```
# Python program showing to modify  
# a global value without using  
# global keyword  
a = 15  
# function to change a global value  
def change():  
    # increment value of a by 5  
    a = a + 5  
    print(a)  
  
change()
```

**UnboundLocalError: local
variable 'a' referenced
before assignment**

This output is an error because we are trying to assign a value to a variable in an outer scope. This can be done with the use of **global** variable.

With global keyword

```
# Python program to modify a global
# value inside a function
a = 15
def change():
    # using a global keyword
    global a
    # increment value of a by 5
    a = a + 5
    print("Value of a inside a function :", a)
change()
print("Value of a outside a function :", a)
```

Value of a inside a
function : 20
Value of a outside
a function : 20

GLOBAL KEYWORD REQUIRED FOR IMMUTABLE DATA TYPE

Global not required for Mutable Data type

```
L = [10,20]
def Change(N):
    L[0]+=N
    L[1]-=N
    print("After Change : ",L)
print("Main : ", L)
n=2
Change(n)
print("Main : ",L)
```

```
Main : [10, 20]
After Change : [12, 18]
Main : [12, 18]
```

GLOBAL KEYWORD NOT REQUIRED FOR MUTABLE DATA TYPE

Give the output of the following:

```
num = 100
def calc(x):
    global num
    if x > 200:
        num*=2
    else:
        num//=2
    return num/2
num = 1000
result = calc(num)
print (num,":",result)
```


Give the output of the following:

```
num = 100
def calc(x):
    global num
    if x > 200:
        num*=2
    else:
        num//=2
    return num/2
num = 1000
result = calc(num)
print (num,":",result)
```

OUTPUT:

2000 : 1000.0

Give the output of the following:

```
x = 5
y = 100
def func():
    global x
    y = x * 2
    x = y + 12
    print("x = ",x,"y = ",y)
    return x + y
print("x = ",x,"y = ",y)
x += func()
print("x = ",x,"y = ",y)
```

Annotations:

- x = 112*
- y = 10*
- 122 + 5 127*

Give the output of the following:

```
x = 5
y = 100
def func():
    global x
    y = x * 2
    x = y + 12
    print("x = ",x,"y = ",y)
    return x + y
print("x = ",x,"y = ",y)
x += func()
print("x = ",x,"y = ",y)
```

OUTPUT:

```
x = 5 y = 100
x = 22 y = 10
x = 37 y = 100
```

Give the output of the following:

```
def game(x,y=10):  
    global a  
    a+=x+2  
    x+=20//y+5  
    y*=a%3  
    print(a,x,y,sep=":")  
    return x  
  
a=5  
b,c=a+2,a*2  
b=game(b)  
c=game(b,c)  
print(a,b,c,sep=' : ')
```

Give the output of the following:

```
def game(x,y=10):  
    global a  
    a+=x+2  
    x+=20//y+5  
    y*=a%3  
    print(a,x,y,sep=":")  
    return x  
  
a=5  
b,c=a+2,a*2  
b=game(b)  
c=game(b,c)  
print(a,b,c,sep=':')
```

OUTPUT:

```
14:14:20  
30:21:0  
30:14:21
```

Give the output of the following:

```
def calculate(a,b = 4):  
    r =1  
    for i in range(1,b + 1):  
        if b % i == 0:  
            r += a ** i  
        else:  
            r -= a * i  
        print(r,end = '#')  
calculate(3)  
print()  
calculate(3,2)
```

OUTPUT:

Give the output of the following:

```
def calculate(a,b = 4):  
    r =1  
    for i in range(1,b + 1):  
        if b % i == 0:  
            r += a ** i  
        else:  
            r -= a * i  
        print(r,end = '#')  
calculate(3)  
print()  
calculate(3,2)
```

OUTPUT:

4#13#4#85#
4#13#

Give the output of the following:

```
def add(k):  
    s=0  
    for i in range(1,k+1):  
        if k % i == 0:  
            s += i  
    return s  
  
def calc(num):  
    for i in range(num,0,-3):  
        if i%2!=0:  
            a=add(i)  
    return a+1,2*a+1,3*a+1  
  
n=8  
p,q,r=calc(n)  
print(p,q,r)
```

OUTPUT:

Give the output of the following:

```
def add(k):  
    s=0  
    for i in range(1,k+1):  
        if k % i == 0:  
            s += i  
    return s  
  
def calc(num):  
    for i in range(num,0,-3):  
        if i%2!=0:  
            a=add(i)  
    return a+1,2*a+1,3*a+1  
  
n=8  
p,q,r=calc(n)  
print(p,q,r)
```

OUTPUT:

7 13 19

Keyword Arguments

A keyword argument is an argument passed to a function or method which is preceded by a keyword and an equals sign.

The general form is:

```
functionname(keyword=value)
```

For example, if a **function header** is like :

```
def check(a, b, c) :
```

Then possible **function calls** for this can be

```
check (b=10,a=20,c=50) #Call1(a gets value 20, b gets value 10, c gets value 50)
```

```
check (a=10,c=20,b=50) #Call1(a gets value 10, b gets value 50, c gets value 20)
```

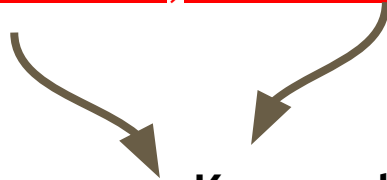
Example of Keyword Arguments

```
def display_info(first_name, last_name):
```

```
    print('First Name:', first_name)
```

```
    print('Last Name:', last_name)
```

```
display_info(last_name = 'Tanvi', first_name = 'Kaur')
```



Keyword arguments

Using different arguments - Default, Positional and Keyword arguments

```
def area_perimeter(width=2, height=2):  
    area = width * height  
    perimeter = (2 * width) + (2 * height)  
    print("Area = " + str(area) + " and Perimeter = " + str(perimeter))
```

```
area_perimeter()           # this function call uses the default arguments  
area_perimeter(10, 5)      # positional arguments width=10 height=5  
area_perimeter(width=10)   # this will set height to use the default value, 2  
area_perimeter(height=4, width=45) # keyword arguments
```

OUTPUT:

```
Area = 4 and Perimeter = 8  
Area = 50 and Perimeter = 30  
Area = 20 and Perimeter = 24  
Area = 180 and Perimeter = 98
```

THANK YOU!

HAPPY LEARNING!!!