FUNCTIONS

Function is the block of code that we can call repeatedly instead of writing the same code again and again.
Main advantage of functions is code reusability.

Types of functions:

- **Built-in functions:** id(), type(), print(0,input(), eval(), etc.

- **User defined functions:**
  Functions that we define on our own to fulfill our requirements.
  Syntax:
  def funciton_name (parameters): *(parameter is optional)*
  '''doc string'''
  {function body}
  return value *(optional)*

  Example:

```
def hello():
  print("Hello!!!!")
hello()
hello()
hello()
```

  Output:
  Hello!!!!
  Hello!!!!
  Hello!!!!

- **Parameters:**
  If the function contains parameters, then while calling the function we need to send parameters with it compulsorily.
  Example:

```
def hello(name):
  print('Hello',name)
hello('Isha')
```

  Output:
  Hello Isha

- **Return Statement:**
  The output of the function is a return statement. For the user, return statement is optional, however internally, the function also returns some value.

  Example 1:

```
def add(a,b):
 sum= a+b
 return sum
a = add(20,10)
print("The sum is",a)
```

  Output:

```
The sum is 30
```

  Example 2:

```
def odd_even(num):
 if num%2==0:
  print(num, "is even")
 else:
  print(num, "is odd")
a= odd_even(4)
print(a)
```

  Output:

```
4 is even
None
```

  Example 3:

```
def fact(num):
  result=1
  while num>=1:
   result= result*num
   num-=1
  return result
a= fact(5)
print(a)
```

  Output:

```
120
```

Note: Return statement in python can return multiple values as well. But it works in the tuple unpacking concept.

Example:

```
def sum_sub(a,b):
 sum=a+b
 sub=a-b
 return sum, sub
a,b= sum_sub(20,10)
print(a,b)
t=sum_sub(10,5)
print(type(t))
```

Output:

30 10
<class 'tuple'>

- **Arguments:**

Actually, the parameters sent in the function are known as formal parameters whereas the value that we use while calling the function is known as actual arguments.

There are four types of arguments:

- Positional Arguments: The order of actual arguments in which the function call has been made will be assigned to the formal parameters in the same order.

- Keyword Arguments: We can pass the value as keywords while calling the function by assigning the actual arguments with formal parameters. We can mix positional argument with keyword argument but after one keyword argument has been made, we have to assign all as keyword arguments.

- Default Arguments: If we want a default answer in case of no argument while calling the function, then we can set the default argument. Parameter without default always follows parameter with a default.

- Variable Length Arguments: When we don't have a fixed length of arguments that we know of, then we make variable length arguments that accept n number of arguments(positional). It creates tuples. It starts with an *(ex: *args). But if we want variable length keyword, it starts with **(ex: **kwargs) and it creates a dictionary. *args comes before **kwargs.

- **Types of variables (Functional Programming):**

  - **Global Variable:**
    Those variables that are made outside the function and can be accessed from any part of the code whether inside or outside the function are global variables.

  - **Local Variable:**
    Those variables that are made inside the function and can only be accessed within a function are local variables.

Example:

```
a=10
def sum():
  s= a+1
  b=1
  return s
s= sum()
print(s)
print(b)
```

Output:

```
11
Traceback (most recent call last):
 print(b)
       ^
NameError: name 'b' is not defined
```

In order to correct this, we can make the variable ‘b’ global by using the global keyword as ‘global b’ before the variable declaration inside the function sum().

Note: Local variable takes priority over global variable inside a function.

We can use global and local variable together with the same name in a function by using a function globals() as:

```
a=10
def f():
  a=99
  print(a)
  print(globals().get('a'))  //or  print(globals()['a'])
f()
```

Output:

99
10

- **Recursive Functions:**

  The functions that call themselves are recursive functions. Two criterias are needed for any function to be recursive:

  1. There should be a terminating criteria.
  2. After every call, the program should move closer to the termination criteria.

Example:

```
def fact(n):
 if n==0:
  result =1
 else:
  result= n*fact(n-1)
 return result

a= fact(5)
print(a)
```

Output:

120

Recursive depth (the number of loop) in python is about 995.

- **Anonymous Functions (Lambda Functions):**

Those functions which do not have any names are known as anonymous functions. These functions are short term and are made for instant use only.
In almost all cases, lambda functions are used when functions are sent as arguments. Generally, functions that take functions as arguments are filter(), map() and reduce().

Example 1: filter() function takes function as first argument and sequence as second argument.

```
def isEven(n):
 if n%2==0:
  return True
 else:
  return False
l=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

```
l1=list(filter(isEven,l))
print(l1)
```

Output:

[2, 4, 6, 8, 10, 12, 14, 16]

Now, using lambda function with it:

```
l=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
l1=list(filter(lambda x: x%2==0,l))
print(l1)
```

Output:

[2, 4, 6, 8, 10, 12, 14, 16]

Example 2: map() function manipulates/takes actions in a sequence and maps the same number of output in another sequence.

```
l=[1,2,3,4,5]
def double(x):
 return 2*x
l1=list(map(double,l))
print(l1)
```

Output:

[2, 4, 6, 8, 10]

Now, using lambda function with it:

```
l=[1,2,3,4,5]
l1=list(map(lambda x:x*2,l))
print(l1)
```

Output:

[2, 4, 6, 8, 10]

Example 3: reduce() function takes the sequence and reduces it to 1.
from functools import reduce //since reduce() is not an inbuilt function

```
l=[1,2,3,4,5,6,7,8,9,10]
result = reduce(lambda x,y:x+y,l)
print(result)
```

Output:

55

Note: Everything in python is an object. Function is also an object since it holds some address.

- **Function aliasing:**
  Giving another name to a function is function aliasing.

  Example:

```
def hello(name):
 print("Hello", name)
wish = hello
hello("Ram")
wish("Sita")
del hello
wish("Hi")
```

  Output:

  Hello Ram
  Hello Sita
  Hello Hi

- **Nesting of Functions:**

```
def outer():
 print("Outer function execution")
 def inner():
  print("Inner function execution")
 print("Outer function calling inner function")
 inner()
outer()
```

  Output:

  Outer function execution
  Outer function calling inner function
  Inner function execution

  Note: We cannot call inner() function from outside the outer() function.