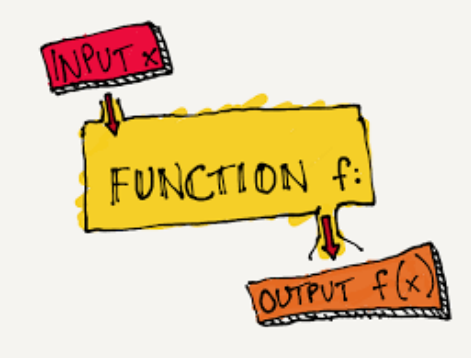
Jawaharlal Nehru Technological University Hyderabad

Kukatpally, Hyderabad - 500 085, Telangana, India



Functions in Python

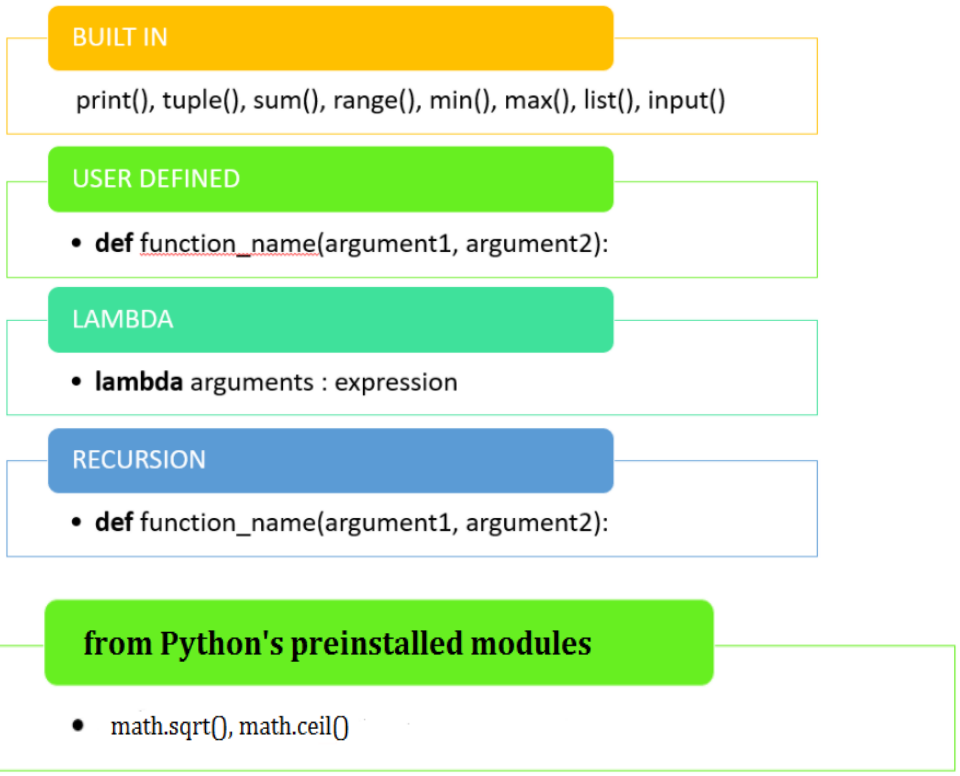
Session 5 , 28 Sep 2022

Dr N V Ganapathi Raju
Professor and HOD of IT
GRIET

Functions in Python

- Functions allow us to group a number of statements into a logical block.
- In programming function refers to a named sequence of operations that perform a computation.
- Used to provide modularity to complex applications and are defined to be re-usable / manageable also saves time.
- We communicate with a function through a clearly defined interface, providing certain parameters to the function, and receiving some information back.
- Apart from this interface, we generally do not how exactly a function does the work to obtain the value it returns.

Various kinds of Functions in Python



In general, Python supports 4 kinds of functions

- **Built-in functions**: which are an integral part of Python (print(), input()). Always available without any additional effort on behalf of the programmer.
- **User-defined functions** which are written by users for users - you can write your own functions and use them freely in your code.
- A **lambda function** is a small function containing a single expression. Helpful when we have to perform small tasks with less code.
- **from Python's preinstalled modules** - a lot of functions, very useful used significantly less often than built-in ones, are available in several modules installed together with Python.

User defined functions

Syntax of a function definition

- **Syntax** :

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

- def keyword: This marks the beginning of the function header.
- function_name: This is a unique name that identifies the function.
- parameters or arguments: Values are passed to the function by enclosing them in parentheses (). optional.
- The colon (:) marks the end of the function header.
- statement(s):There must be one or more valid statements that make up the body of the function.

Notice that the statements are indented (typically tab or four spaces).

- **An optional return statement to return a value from the function.**

Syntax of function call

- To call a function, simply type the function name with appropriate parameters.
- Syntax of function call :

```
function_name (parameters)
```

- Arguments are the actual value that is passed into the calling function.
- Note 1: There must be a one-to-one correspondence between the formal parameters in the function definition and the actual arguments of the calling function.
- Note 2: When we call a function, the control flows from the calling function to the function definition.
- Note 3: Once the block of statements in the function definition is executed, then the control flows back to the calling function and proceeds with the next statement.

return statement

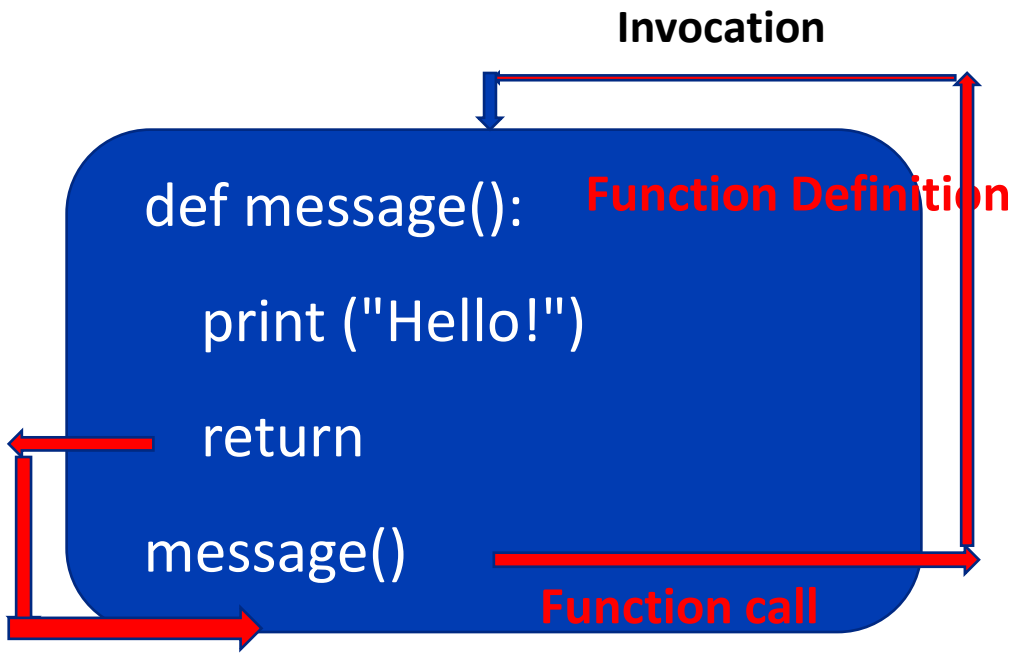
- The return statement is used to exit a function and go back to the place from where it was called.

```
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 a function, then the function will return the None object.

How function works



You mustn't invoke a function which is not known at the moment of invocation.

Raises NameError: exception

- when you **invoke** a function, Python remembers the place where it happened and *jumps* into the invoked function;
- the body of the function is then **executed**;
- reaching the end of the function forces Python to **return** to the place directly after the point of invocation

functions for passing various type of data to function as arguments

```
def msg1(s):
    print (s)

def msg2(i):
    print (i)

def msg3(f):
    print (f)

msg1(“programming")

msg2(100)

msg3(1500.50)
```

```
programming
100
1500.5
```

functions for addition of two numbers

```
def sum(x,y):
    s=x+y
    print ("Sum is inside function" ,s)
sum(10,20)


def sum(x,y):
    s=x+y
    return s
total=sum(10,20)
print ("Sum is outside function“, total)
```

```
1  def sum(x,y):
2      s=x+y
3      print ("Sum is inside function" ,s)
4  sum(10,20)
5
6  def sum(x,y):
7      s=x+y
8      return s
9
10 total=sum(10,20)
11 print ("Sum is outside function",total)
```

```
Sum is inside function 30
Sum is outside function 30
```

function for finding biggest of two numbers

```
def max_2( x, y ):
    if x > y:
        return x
    else :
        return y
print(max_2(10,20))
```

```
1 def max_2( x, y ):
2     if x > y:
3         return x
4     else :
5         return y
6 print(max_2(10,20))
```

```
20
```

function for checking prime number

```
def prime(num):
    n=1
    count=0
    while (n<=num):
        if((num%n)==0):
            count=count+1
        n=n+1
    if(count==2):
        return ("Prime")
    else:
        return ("not a prime")
res = prime(5)
print(res)
```

```
1  def prime(num):
2      n=1
3      count=0
4      while (n<=num):
5          if((num%n)==0):
6              count=count+1
7          n=n+1
8
9      if(count==2):
10         return ("Prime")
11     else:
12         return ("not a prime")
13 res = prime(5)
14 print(res)
```

```
Prime
```

function for finding factorial of a given number

```
def factorial(num):
    n=1
    while num>0:
        n=n*num
        num=num-1
    return n
res=factorial(5)
print(res)
```

```
1  def factorial(num):
2      n=1
3      while num>0:
4          n=n*num
5          num=num-1
6      return n
7
8  res=factorial(5)
9  print(res)
```

```
120
```

Scope and Lifetime of variables

Scope and Lifetime of variables

- Python programs have two scopes: **global** and **local**.
- A variable is a **global variable** if its value **is accessible and modifiable throughout your program**.
- Global variables have a global scope.
- A variable that is defined inside a **function definition is a local variable.**
- The **lifetime** of a variable refers to the **duration of its existence**.
- The local variable is created and destroyed every time the function is executed, and it cannot be accessed by any code outside the function definition.
- Local variables inside a function definition have local scope and exist as long as the function is executing.

Local variables

```
print('-----local var---------')
def message():
        a=10
        print ("Value of a is",a)
        return


message()
#print (a)
```

```
1 print('-----local var---------')
2 def message():
3         a=10
4         print ("Value of a is",a)
5         return
6 message()
7 #print (a)
```

```
-----local var---------
Value of a is 10
```

```
1 print('-----local var---------')
2 def message():
3         a=10
4         print ("Value of a is",a)
5         return
6 message()
7 print (a)
```

```
-----local var---------
Value of a is 10
---------------------------------------------
NameError                                 Tr
<ipython-input-26-241609dbf007> in <module>
      5         return
      6 message()
----> 7 print (a)

NameError: name 'a' is not defined
```

Global variables

```
print('-----global var---------')
b=20
def message():
    a=10
    print ("Value of a is",a)
    print ("Value of b is",b)


message()
#print (a)
print (b)
```

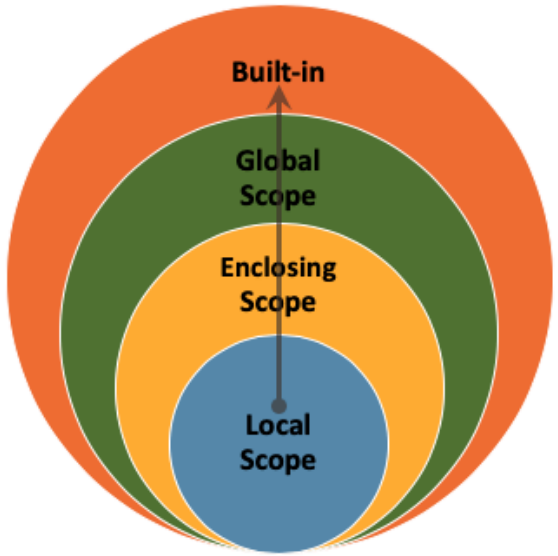
```
1  print('-----global var---------')
2  b=20
3  def message():
4          a=10
5          print ("Value of a is",a)
6          print ("Value of b is",b)
7
8  message()
9  #print (a)
10 print (b)
```

```
-----global var---------
Value of a is 10
Value of b is 20
20
```

Official : Scopes as per LEGB rule

- There are four major types of variable scope and is the basis for the LEGB rule.
- LEGB stands for **Local -> Enclosing -> Global -> Built-in**.
- **Local scope means, define a variable within a function.**
- Enclosing scope means, defining variables in nested functions
- Global scope means, Whenever a variable is defined outside any function,

  it becomes a global variable, and its scope is anywhere within the program.
- Built-in Scope means, All the special reserved keywords fall under this scope.




```
# Global scope
x = 0

def outer():
    # Enclosed scope
    x = 1
    def inner():
        # Local scope
        x = 2
```

Local vs Enclosing vs Global variables

```
test_var = 5
def outer_function():
    test_var = 60
    def inner_function():
        test_var = 100
        print(f"Local var {test_var} in inner fun")
    inner_function()
    print(f"Local var {test_var} in outer fun")
outer_function()
print(f"Global var {test_var} in global")
```

```
Local var 100 in inner fun
Local var 60 in outer fun
Global var 5 in global
```

Function Arguments in Python

Function Arguments in Python

In Python, We can call a function using various types of formal arguments:

- Default arguments.
- Keyword arguments
- Required arguments/Positional arguments
- Flexible arguments (***args** and ****kwargs**)

Default arguments.

- In some cases, we have a function with multiple parameters and we have a **common value for some of them**. We can specify default arguments for some of the function parameters.
- In these cases, we can call our function without specifying the values for the parameters with default arguments. To do this in Python, we can use the = sign followed by the default value.

```
def raise_power(number, power = 2):
    return number ** power

print(raise_power(9))

print(raise_power(2, 3))
```

```
1  def raise_power(number, power = 2):
2      return number ** power
3
4  print(raise_power(9))
5
6  print(raise_power(2, 3))
```

```
81
8
```

Keyword arguments

- Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the **caller identifies the arguments by the parameter name**.
- This allows you to **skip arguments** or **place them out of order** because the Python interpreter is able to use the keywords provided to match the values with parameters.
- We can use the keyword arguments using the argument name and the = sign.

```
def raise_power(number, power):
    return number ** power
print(raise_power(2, 3))
print(raise_power(number = 2, power = 3))
print(raise_power(power = 2, number = 3))
```

```
1  def raise_power(number, power):
2
3      return number ** power
4
5  print(raise_power(2, 3))
6
7  print(raise_power(number = 2, power = 3))
8
9  print(raise_power(power = 2, number = 3))
```

```
8
8
9
```

Required arguments (positional arguments)

- Required arguments must be passed to the function in the exact positional order to match the function definition.
- If the arguments are not passed in the right order, or if the arguments passed are more or less than the number defined in the function, a syntax error will be encountered.

```
def sum(a,b):
        c=a+b
        print (c)
sum(10,20)
#sum(20)
```

```
1  def sum(a,b):
2          c=a+b
3          print (c)
4  sum(10,20)
5  #sum(20)
```

```
30
```

Flexible arguments

- We may want to define a function which accepts **more arguments than we have specified** in the function. We may need to pass any number of arguments to our function.
- We can use the **special syntax *args** and ****kwargs** in our function definitions to achieve that.
- ***args :** These arguments are called **non-named variable-length arguments**.

```
def add(*num):
    sum = 0
    for n in num:
        sum = sum + n
    print("Sum:",sum)

add(10,20)
add(10,20,30,40)
```

```
1  def add(*num):
2      sum = 0
3      for n in num:
4          sum = sum + n
5      print("Sum:",sum)
6
7  add(10,20)
8  add(10,20,30,40)
```

```
Sum: 30
Sum: 100
```

Flexible arguments

- ****kwargs :** These arguments are called **named variable-length arguments**.
- In the function, we use the double asterisk ** before the parameter name to denote this type of argument.
- The arguments are passed as a **dictionary** and these arguments make a dictionary inside function with name same as the parameter excluding double asterisk **.

```
def kwargs_example(**kwargs):
    print(type(kwargs))
    print(kwargs)

kwargs_example(age = 25, position = "Data Scientist")

kwargs_example(name = “abc", email = "support@abc.com", position = "ML Engineer")
```

Built-in functions

Built-in Functions

- The Python interpreter has a number of functions that are built into it and are always available.

Built-in Functions				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

https://docs.python.org/3/library/functions.html

Examples of built-in functions

```
print(abs(-3))
print(min(1, 2, 3, 4, 5))
print(max(4, 5, 6, 7, 8))
print(pow(3, 2))
print(len("Conduira Online"))
```

```
1 print(abs(-3))
2 print(min(1, 2, 3, 4, 5))
3 print(max(4, 5, 6, 7, 8))
4 print(pow(3, 2))
5 print(len("Conduira Online"))
```

```
3
1
8
9
15
```

Preinstalled modules

Preinstalled modules

- Modules in Python are reusable libraries of code having *.py* extension, which implements a group of methods and statements. Python comes with many built-in modules as part of the standard library.
- To use a module in your program, import the module using *import* statement. All the *import* statements are placed at the beginning of the program.

import module_name where import is a keyword

- Example : import math
- The ***math* module** is part of the Python standard library which provides access to various **mathematical** functions and is always available to the programmer
- The syntax for using a function defined in a module is, ***module_name.function_name()***

math module

```
import math
print(math.ceil(5.4))
print(math.sqrt(4))
print(math.pi)
print(math.cos(1))
print(math.factorial(6))
print(math.pow(2, 3))
```

```
1 import math
2 print(math.ceil(5.4))
3 print(math.sqrt(4))
4 print(math.pi)
5 print(math.cos(1))
6 print(math.factorial(6))
7 print(math.pow(2, 3))
```

```
6
2.0
3.141592653589793
0.5403023058681398
720
8.0
```

random module

- Another useful module in the Python standard library is the *random* module which generates random numbers.

```
import random
print(random.random())
print(random.randint(5,10))
```

```
1  import random
2  print(random.random())
3  print(random.randint(5,10))
```

```
0.8787676695752806
9
```

- *random()* function generates a random floating-point number between 0 and 1 and it produces a different value each time.
- *random randint(start, stop)* which generates a integer number between start and stop argument numbers (including both).

Recursive functions

Recursive Functions

- A **recursive function** is a function defined **in terms of itself via self-referential expressions**.
- The **function will continue to call itself and repeat its behavior until some condition is met to return a result.**
- All recursive functions share a common structure made up of two parts: **base case and recursive case.**

Examples using recursive functions in Python

```
def rec_cout(n):
    if n <= 0:
        print(“Python!”)
    else:
        print(n)
        rec_cout(n-1)

rec_cout(3)
```

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
print_n(“Python",3)
```

Recursive function for factorial of a given number

```
def factorial_recursive(n):
    # Base case: 1! = 1
    if n == 1:
        return 1

    # Recursive case: n! = n * (n-1)!
    else:
        return n * factorial_recursive(n-1)
factorial_recursive(5)
```

```
1  def factorial_recursive(n):
2      # Base case: 1! = 1
3      if n == 1:
4          return 1
5
6      # Recursive case: n! = n * (n-1)!
7      else:
8          return n * factorial_recursive(n-1)
9  factorial_recursive(5)
```

```
120
```

Lambda functions

lambda functions

- An anonymous function is a function that is defined **without a name**.

- While normal functions are defined using the def keyword in Python, *anonymous functions are defined using the **lambda** keyword.*

Syntax for lambda functions: **lambda arguments: expression**

  - Lambda functions can have any **number of arguments** but return only one **expression**. The **expression** is **evaluated** and **returned**.
  - Lambda functions are syntactically restricted to return a single expression
  - We can use lambda functions as an anonymous functions inside other functions
  - Lambda functions can be used **wherever function objects are required**.

Examples of lambda functions

```
v1 = lambda x : x * 2
print(v1(5))

v2 = lambda x: x * 2
print(v2(5.0))

v3 = lambda x, y: x + y
print (v3 (5,10))

x="lambda functions"
(lambda x : print(x))(x)

Name = lambda first, second: first +' '+ second
Name('Lambda', 'Functions')
```

```
10
10.0
15
lambda functions
'Lambda Functions'
```

Limitations

- Since we can evaluate single expressions, features like

  iteration,

  conditionals,

  exception handling cannot be specified.

- But very useful in the place of one-line functions that evaluate single expressions.

Addition , multiplication and power operations

```
add = lambda a,b,c : a+b+c
print(add(5,3,2))

multiply = lambda x,y:x * y
print(multiply(3,7))

power = lambda m,n: m**n
print(power(6,2))
```

```
1  add = lambda a,b,c : a+b+c
2  print(add(5,3,2))
3
4  multiply = lambda x,y:x * y
5  print(multiply(3,7))
6
7  power = lambda m,n: m**n
8  print(power(6,2))
```

```
10
21
36
```