

# **Python Programming - IV**

- By Nimesh Kumar dagur, CDAC Noida, India

# Python Functions

- In Python, function is a named group of related statements that perform 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 and makes code reusable.

# Syntax of Function

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

## Components of function definition:

- Keyword `def` marks the start of function header
- A function name to uniquely identify it.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon `(:)` to mark the end of function header.
- Optional documentation string (docstring) to describe what the function does
- One or more valid python statements that make up the function body.
- An optional return statement to return a value from the function.

## Example of a function

```
def greet(name):  
    """This function greets to  
    the person passed in as  
    parameter"""  
    print("Hello, " + name + ". Good morning!")
```

```
>>> greet('Paul')  
Hello, Paul. Good morning!
```

# Function Call

- 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.

```
>>> greet('Paul')  
Hello, Paul. Good morning!
```

# Docstring

- The first string after the function header is called the docstring and is short for documentation string.
- It is used to explain in brief, what a function does.
- Although optional, documentation is a good programming practice.
- In the previous example, we have a docstring immediately below the function header.
- We generally use triple quotes so that docstring can extend up to multiple lines.
- This string is available to us as `__doc__` attribute of the function.

# Docstring

```
>>> print(greet.__doc__)  
This function greets to  
    the person passed into the  
    name parameter
```

# The return statement

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

## Syntax of return

```
return [expression_list]
```

- This statement can contain expression which 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.



# Example of return

```
def absolute_value(num):  
    """This function returns the absolute  
       value of the entered number"""  
  
    if num >= 0:  
        return num  
    else:  
        return -num  
  
print(absolute_value(2))  
print(absolute_value(-4))
```

Output

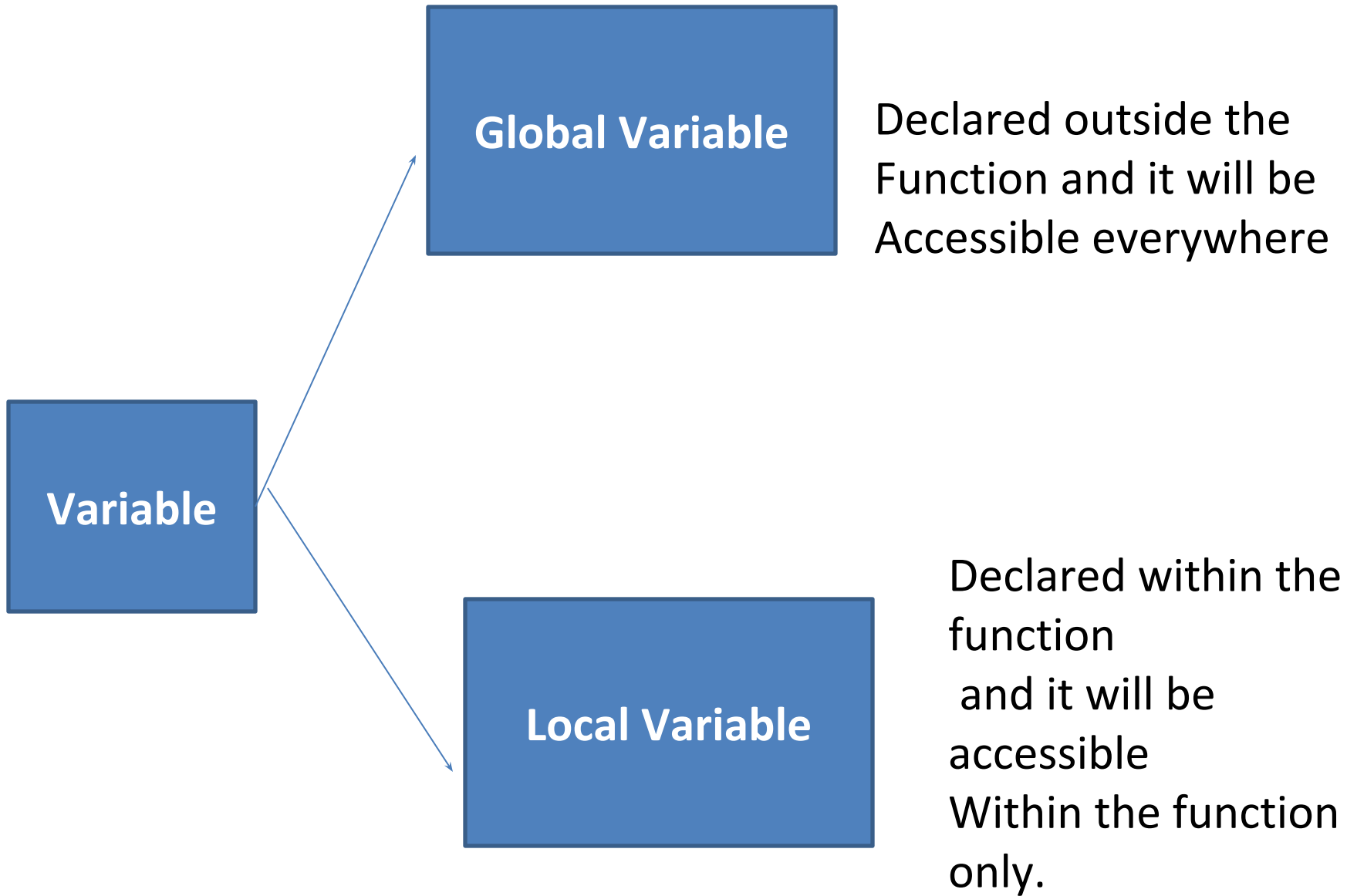
2  
4

## Example of return None

```
>>> print(greet("May"))  
Hello, May. Good morning!  
None
```

# Scope and 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 is not visible from outside. Hence, they have a local scope.
- 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.



## Example

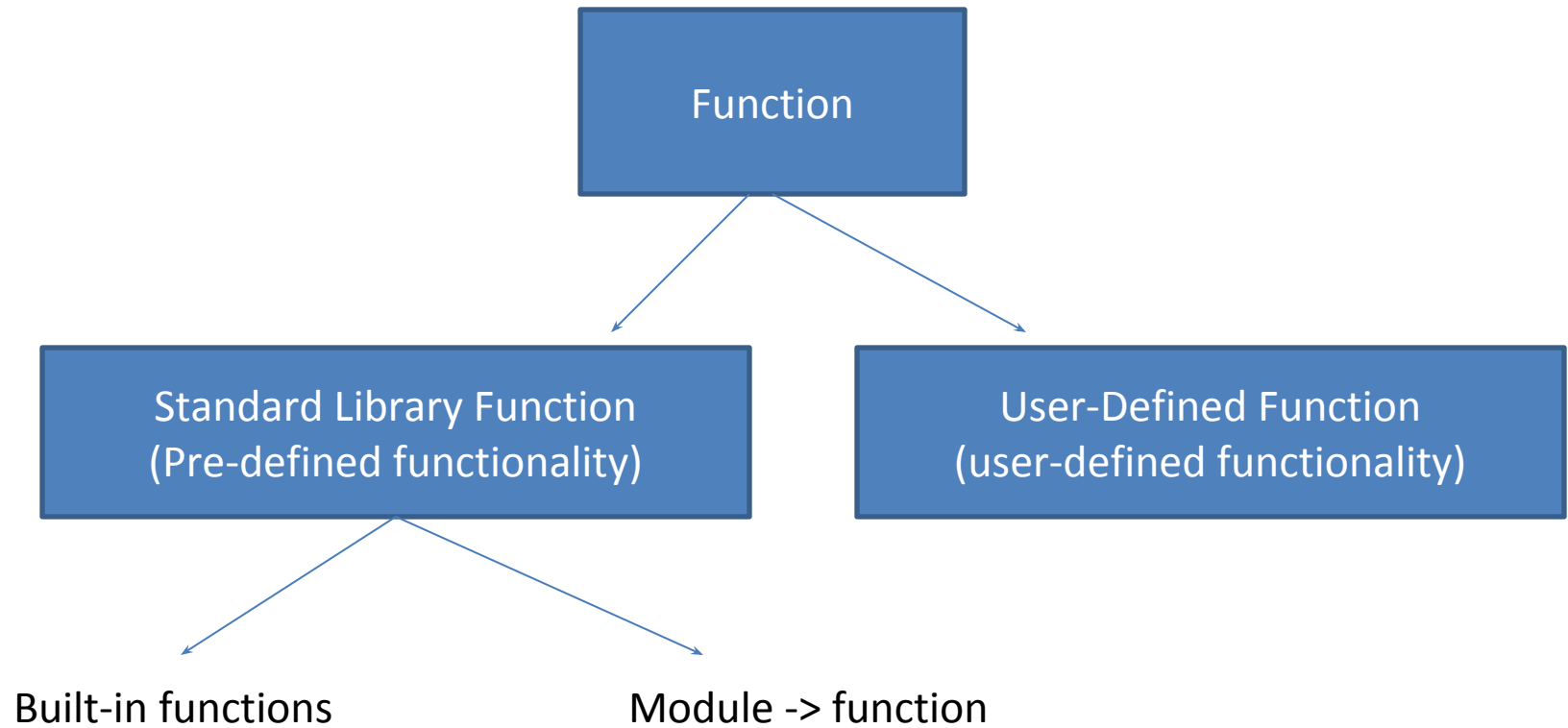
### Scope of a variable inside a function

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

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

# Types of Functions

- Basically, we can divide functions into the following two types:
  1. **Built-in functions** - Functions that are built into Python.
  2. **User-defined functions** - Functions defined by the users themselves.



# Python Programming User-defined Functions

- Functions that we define ourselves to do certain specific task are referred as user-defined functions.
- Functions that readily come with Python are called built-in functions.
- If we use functions written by others in the form of library, it can be termed as library functions.
- All the other functions that we write on our own fall under user-defined functions.
- So, our user-defined function could be a library function to someone else.



# Advantages of user-defined functions

- User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmers working on large project can divide the workload by making different functions.

## Example of a user-defined function

```
def my_addition(x,y):  
    """This function adds two  
    numbers and return the result"""  
    sum = x + y  
    return sum  
  
num1 = float(input("Enter a number: "))  
num2 = float(input("Enter another number: "))  
print("The sum is", my_addition(num1,num2))
```



```
Enter a number: 2.4  
Enter another number: 6.5  
The sum is 8.9
```

# Python Function Arguments

```
def greet(name,msg):  
    """This function greets to  
    the person with the provided message"""  
    print("Hello",name + ', ' + msg)  
  
greet("Monica","Good morning!")
```

## Output

```
Hello Monica, Good morning!
```

# Python Function Arguments

- if we call **greet** function with different number of arguments, the interpreter will complain.
- Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> greet("Monica")      # only one argument
TypeError: greet() missing 1 required positional argument: 'msg'
```

```
>>> greet()              # no arguments
TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'
```

# Variable Function Arguments

- In previous example, function had fixed number of arguments.
- In Python there are other ways to define a function which can take variable number of arguments.
- Three different forms of this type are:
  1. **Default Arguments**
  2. **Keyword Arguments**
  3. **Arbitrary Arguments or Variable-length arguments**

# Default Arguments

- Function arguments can have default values in Python.
- We can provide a default value to an argument by using the assignment operator (=).

```
def greet(name, msg = "Good morning!"):
    """This function greets to
    the person with the provided message.
    If message is not provided, it defaults
    to "Good morning!" """

    print("Hello", name + ', ' + msg)
```

## Default Arguments

```
>>> greet("Kate")
```

```
Hello Kate, Good morning!
```

```
>>> greet("Bruce", "How do you do?")
```

```
Hello Bruce, How do you do?
```

# Default Arguments

- Any number 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.

```
def greet(msg = "Good morning!", name):
```

We would get an error as:

```
SyntaxError: non-default argument follows default argument
```



# Keyword Arguments

- When we call a function with some values, these values get assigned to the arguments according to their position.
- For example, in the above function `greet()`, when we called it as `greet("Bruce", "How do you do?")`, the value "Bruce" gets assigned to the argument *name* and similarly "How do you do?" to *msg*.
- 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.

# Keyword Arguments

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

```
def greet(name, msg = "Good morning!"):
    """This function greets to
    the person with the provided message.
    If message is not provided, it defaults
    to "Good morning!" """

    print("Hello", name + ', ' + msg)
```

`greet(name = "Bruce", msg = "How do you do?")`

# 2 keyword arguments

`greet(msg = "How do you do?", name = "Bruce")`

# 2 keyword arguments (out of order)

`greet("Bruce", msg = "How do you do?")`

# 1 positional, 1 keyword argument

# Keyword Arguments

- we can mix positional arguments with keyword arguments during a function call.
- But we must keep in mind that keyword arguments must follow positional arguments.
- Having a positional argument after keyword arguments will result into errors.
- Example:

```
greet(name="Bruce", "How do you do?")
```

Will result into error as:

```
SyntaxError: non-keyword arg after keyword arg
```

# Arbitrary Arguments or Variable-length arguments

- Sometimes, we do not know in advance the number of arguments that will be passed into a function.
- Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.
- In the function definition we use an asterisk (\*) before the parameter name to denote this kind of argument.

## Example.

```
def greet(*names):  
    """This function greets all  
    the person in the names tuple."""  
  
    # names is a tuple with arguments  
    for name in names:  
        print("Hello",name)  
  
greet("Monica","Luke","Steve","John")
```



```
Hello Monica  
Hello Luke  
Hello Steve  
Hello John
```

# Python Recursion

- Recursion 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, a function can call other functions.
- It is even possible for the function to call itself.
- These type of construct are termed as recursive functions.

```
5*4*3*2*rec(1)
```

```
5*4*3*rec(2)
```

```
5*4*rec(3)
```

```
5*rec(4)
```

```
rec(5)=> stack will create
```

## Example of recursive function:

To find the factorial of an integer:

```
def recur_fact(x):  
    """This is a recursive function  
    to find the factorial of an integer"""  
  
    if x == 1:  
        return 1  
    else:  
        return (x * recur_fact(x-1))
```

```
num = int(input("Enter a number: "))  
if num >= 1:  
    print("The factorial of", num, "is", recur_fact(num))
```



```
Enter a number: 4  
The factorial of 4 is 24
```



# Advantages of recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

# Disadvantages of recursion

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

# Python Anonymous/Lambda Function

- In Python, 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.
- Hence, anonymous functions are also called lambda functions.

# Syntax of Lambda Function

```
lambda arguments: expression
```

- Lambda functions can have any number of arguments but only one expression.
- The expression is evaluated and returned.
- Lambda functions can be used wherever function objects are required.

## Example of Lambda Function

Here is an example of lambda function that doubles the input value.

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

## Example of Lambda Function

- A lambda can take multiple arguments and can return (like a function) multiple values.

```
>>> fn = lambda x, y, z: (x ** 2) + (y * 2) + z
>>> fn(4, 5, 6)
32
```

```
>>> a = lambda x, y: (x * 3, y * 4, (x, y))
>>> a(3, 4)
(9, 16, (3, 4))
```

# Use of Lambda Function

- We use lambda functions when we require a nameless function for a short period of time.
- In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments).
- Lambda functions are used along with built-in functions like `filter()`, `map()` etc.

## Example use with map()

- The map() function in Python takes in a function and a list.
- The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

**use of map() function to double all the items in a list.**

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)
```



[2, 10, 8, 12, 16, 22, 6, 24]



## Example use with filter()

- The filter() function in Python takes in a function and a list as arguments.
- The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

**use of filter() function to filter out only even numbers from a list:**

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)
```



```
[4, 6, 8, 12]
```