## What is a Function

- In Python, function is a 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)

  Above shown is a function definition which consists of following components.
  - Keyword def  which marks the start of function header.
  - A function name to uniquely identify it. Function name can be any unique identifier.
  - 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. Statements must have same indentation level.
  - An optional return statement to return a value from the function.

*Note : The details of """docstring"""  is explained in slide no.6*

# Example of a Function

- def greet(name):

      """This function greets the person  passed in as parameter"""

          print("Hello, " + name + ". Good morning!")

Here, a function of  name greet() is defined. It has one argument : name

O/P :

If we execute this code with the name as Rahul, the output  is :

Hello Rahul. Good Morning!

## How to call a Function inPython

- Once we have defined a function, we can call it from
  - another function,
  - program or
  - Python prompt.
- To call a function we simply type the function name with appropriate parameters. eg: To call the function greet with the name Rahul, we write the following at the Python prompt
- >>> greet(‘Rahul')

## 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 example of the function given in previous slide, 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.
- For example:
- Try running the following into the Python shell to see the output.
  - >>> 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.
- For example:
  - >>> print(greet(“Rahul")) or
  - >>> greet(“Rahul”)
- Hello, Rahul. Good morning!
- None : Here, None is the returned value.

## Example of a Function using Return Statement

```
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))
# Output: 2

print(absolute_value(-4))
# Output: 4
```

## 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 is not visible from outside. Hence, they have a local scope.
- Lifetime of a variable is the period throughout which the variable exits 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 of a Function to Illustrate the use of Scope and lifetime of a Variable

```
def my_func():                    # function is defined
    x = 100
   print("Value inside function:",x)
  x = 200
my_func()                         # function is called
print("Value outside function:",x)
```

O/P

```
Value inside function: 100
Value outside function: 200
```

## Types of Functions

- **Build- In functions :** Functions that are built in Python are called Build- In functions .eg : *split(), sort(), append(), input(), print()*

- **User-Defined Functions:** Functions defined by the users themselves are called User-Defined functions.
  The functions greet() and my_func() explained in the previous slides are examples of user defined functions

# Default Arguments

- Python allows function argumets to have default values.
- If the function is called without the argument, the argument gets its default value.
- Further, arguments can be specified in any order by using named arguments.
- Eg :

```
def  CalArea(height =2, width = 5)
        area = width*height
        print “ Width = “, width, “ \tHeight = “, height, “ \tarea = “, area
CalArea()
CalArea(width = 5.6)
CalArea(height = 9)
CalArea(width = 7, height = 5)
CalArea(4.5, 3.5)
CalArea(5)
```

- The O/P of the previous program will be:

```
Width = 5     Height =2   area = 10
Width = 5.6  Height =2   area = 11.2
Width = 5     Height =9   area = 45
Width = 7     Height =5   area = 35
Width = 3.5  Height =4.5   area = 15.75
Width = 5     Height =5  area = 25
```

## Recursion

- Recursion is the process of defining something in terms of itself.

  We know that 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***.
- Every recursive function should have a terminating or base condition.
- Following is an example of recursive function to find the factorial of an integer.
- Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

## Example of recursive function

```
# find the factorial of a number
def calc_factorial(x):
            """This is a recursive function    to find the
factorial of an integer"""
if x == 1:
             return 1
else:
            return (x * calc_factorial(x-1))
num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

The recursive call is explained as follows:

```
calc_factorial(4) # 1st call with 4
4 * calc_factorial(3) # 2nd call with 3
4 * 3 * calc_factorial(2) # 3rd call with 2
4 * 3 * 2 * calc_factorial(1) # 4th call with 1
4 * 3 * 2 * 1 # return from 4th call as number=1
4 * 3 * 2 # return from 3rd call
4 * 6 # return from 2nd call
24 # return from 1st call
```

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

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

## Recursion Contd..

**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.

## Global and Local Variables

**Global Variables**

- In Python, a variable declared outside of the function or in global scope is known as global variable. This means, global variable can be accessed inside or outside of the function.
- **Example 1: Create a Global Variable**

```
x = "global“
        def foo():
                print("x inside :", x)
        foo()
                print("x outside:", x)
```

O/P

```
x inside : global
x outside: global
```

## Global and Local Variables contd…

- **Local Variables**

A variable declared inside the function's body or in the local scope is known as local variable.

**Accessing local variable outside the scope**

```
def foo():
        y = “local”
foo()
        print(y)
```

**O/P :**

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

The output shows an error, because we are trying to access a local variable `y` in a global scope whereas the local variable only works inside `foo()` or local scope.

## Creating a Local Variable

```
def foo():
        y = "local"
        print(y)
  foo()
```

**O/P :**

```
        local
```

## Using Global and Local variables in same code

```
x = "global“
            def foo():
                  global x
                  y = "local"
                  x = x * 2
                  print(x)
                  print(y)
             foo()
```

**O/P :**

```
global global
local
```

## Global variable and Local variable with same name

```
x = 5
def foo():
        x = 10
        print("local x:", x)
foo()
        print("global x:", x)
```

**O/P**

```
local x: 10
global x: 5
```

## Global Keyword

In Python, global keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

## Rules of global Keyword

The basic rules for global keyword are:

- When we create a variable inside a function, it's local by default.
- When we define a variable outside of a function, it's global by default. You don't have to use global keyword.
- We use global keyword to read and write a global variable inside a function.
- Use of global keyword outside a function has no effect

## Global Keyword contd…

- **Use of global Keyword (With Example)**

```
c = 1 # global variable
def add():
        print(c)
add()
```

**When we run above program, the output will be:**

```
1
```

- **Modifying Global Variable From Inside the Function**

```
c = 1 # global variable
def add():
        c = c + 2 # increment c by 2
        print(c)
add()
```

**When we run above program, the output shows an error:**

```
UnboundLocalError: local variable 'c' referenced before assignment
```

- Changing Global Variable From Inside a Function using global

```
c = 0 # global variable
def add():
        global c
        c = c + 2 # increment by 2
        print("Inside add():", c)
add()
print("In main:", c)
```

When we run above program, the output will be:

```
Inside add(): 2 In main: 2
```