Functions

Ch-9

contents

- What is function?
- Difference between function and method
- How to define function
- How to use(call) function

What is function?

- Function
  - A group of statements
    - that are intended to do a specific task.
- Reusability
- Modularity
- Easy code maintenance

Difference between function and method

- Function:
  - A function is called using its name, only.
- Method:
  - When a function is defined inside a class,
    - It becomes a method.
  - A method is called using, any one of the following ways:
    - objectName.methodName()
    - ClassName.methodName()

Syntax

```
def functionName():
    function statement
    function statement
    function statement
```

Example

```
def greet():
    print(“Entered in function.”)
    print(“Quitting from function.”)
```

Passing parameters

```
def functionName(para1, para2,para3,…):
    function statement
    function statement
    function statement
```

Example

```
def sum(num1,num2):
    ans = num1 + num2
    print(ans)
```

Pass by value

- Copy of object is passed to the function
- Changes made inside the function do not get reflect to calling function

Pass by pointer

- Memory address or reference of object is passed to the function.
- Changes made in the function reflect to the calling function.

Python : How parameters are passed?

- In python
  - Neither pass by value nor pass by pointer is used
- Rather, values are sent to functions by means of the object references.
  - We know that in python,
    - everything is considered as object.
      - All numbers
      - Strings
      - Tuples
      - Lists
      - dictionaries

Python : variable creation

[function-parameters.ipynb]

- In other programming languages:
  - x = 10:
    - A variable with a name ‘x’ is created first and some memory is allocated to the variable.
    - Then the value 10 is stored into the variable ‘x’.
    - Image a ‘x’ as a box where 10 is stored.
- In Python:
  - x = 10:
    - That is not the case in Python. In Python everything is an object.
    - An object can be imagined as a memory block where we can store some value.
    - In this case an object with a value ’10’ is created in memory for which a name ‘x’ is attached.
    - So, 10 is the object and ‘x’ is the name or tag given to that object.
    - Object is created on heap memory. Use id() function to know address of object.

Python: parameter passing

[function-parameters.ipynb]

- When we pass values like: numbers, strings, tuples or lists to a function,
  - the references of these objects are passed to the function.
  - Ex.

```
def modify(x):
    x = 9
    print(x, id(x))
##
x = 7
modify(x)
print(x, id(x))
```

Python: parameter passing..

[function-parameters.ipynb]

- When we pass values like: numbers, strings, tuples or lists to a function,
  - the references of these objects are passed to the function.
  - Ex…. (previous slide)
  - Here, the object is 7; and its reference is ‘x’. this ‘x’ is being passed to the modify() function.
  - Inside the function we are using x=9:
    - This means another object 9 is created in heap, and that object is referenced by the name ‘x’.
    - The reason why another object is created in memory is that the integer objects are immutable.
    - And hence, in modify(), we get output x = 9.

Remember

- In python,
  - Integers, floats, strings, tuples are immutable.
    - i.e. their data can’t be modified.
  - When we try to change their value, a new object is created..
    - with the modified value.
  - Whereas, lists and dictionaries are mutable.
    - i.e. when we change their data, the same object gets modified..
      - and new object is NOT created.

‘list’ as function parameter

[function-parameters.ipynb]

```
def modify(x):
    x.append(9)
    print("In modify(), x= ", x)
##

x = [7,7,7]
print("x=",x)
print("Calling modify()..")
modify(x)
print("Came back from modify().")
print("x=",x)
```

‘list’ as function parameter...

[function-parameters.ipynb]

```
def modify(x):
    x = [9,9,9]  # notice the assignment statement
    print("In modify(), x= ", x)
##

x = [7,7,7]
print("x=",x)
print("Calling modify()..")
modify(x)
print("Came back from modify().")
print("x=",x)
```

Formal and actual arguments

[function-parameters.ipynb]

- Formal parameters:
  - Used in the function definition
    - Useful to receive values from outside of the fuction
- Actual parameters:
  - Used in the function call

```
def sum(x,y):      # x and y are formal parameters
    z = x + y
    print(z)
##

a = 7; b = 9
sum(a,b)  # a and b are actual parameters
```

Actual parameters/arguments

- 4 types of actual arguments:
  - Positional arguments
  - Keyword arguments
  - Default arguments
  - Variable length arguments

Positional arguments

- These are the arguments
  - passed to a function in correct positional order
  - Here,
    - the number of arguments, and their positions
    - In the function definition should match exactly with the number and position of the argument in the function call.

    - [function-parameters.ipynb]

Keyword arguments

[function-parameters.ipynb]

- Keyword arguments
  - Identify the parameters by their names.

```
def divide(dividend,divisor):
    return dividend/divisor
##
divide(dividend = 70, divisor = 7) # notice order
divide(divisor = 7, dividend = 70) # notice order

def printName(firstname,surname):
    print(firstname, ' ', surname)
##
printName(firstname='narayan',surname='joshi’)
printName(surname='joshi',firstname='narayan')
```

Default arguments

- We can mention some default value for
  - the function parameters in the definition.

```
def divide(dividend,divisor=10):
    return dividend/divisor
##
divide(70, 7)
divide(70)
divide(dividend = 70)
```

  - [function-parameters.ipynb]

Variable number of arguments [function-parameters.ipynb]

- Variable number of arguments
  - Sometimes, programmer does not know
    - how many parameters a function may receive.
    - In that case, the programmer cannot decide how many arguments to be given in the function definition.

```
def printNames(*vargs):
    print(vargs)
##
printNames(1,2,3)
printNames(1,2,3,4)
printNames(1,2,3,4,5)
```

Local and global variables

[local-global.ipynb]

- Local variables
  - When we declare a variable inside a function,
    - It becomes a local variable.
    - Limited scope to that function where it is created.
      - i.e. not available outside that function.

```
def fun():
    x = 7
    print(x)
##
fun()
print(x)
```

Local and global variables

[local-global.ipynb]

- Global variables
  - When we declare a variable above a function,
    - It becomes a global variable.
    - Such variables are available to
      - all the functions/code which are written after it.

```
y = 9
def fun():
    x = 7
    print(y)
    print(x)
##
fun()
print(y)
print(x)
```

- Remember:
  - Scope of local variable is
    - limited only to the function in which it is declared.
  - Scope of global variable is
    - the entire program body written below it.

The global keyword

[local-global.ipynb]

- Sometimes, the global variable and the local variable
  - may have the same name.
  - In that case, the function, by default
    - refers to the local variable and ignores the global variable.
  - So, the global variable is not accessible inside the function

```
y = 9 # global y
def fun():
    y = 7 # local y
    print('y=',y) # prints local y
##
fun()
print(y)  # prints global y
```

The global keyword

[local-global.ipynb]

- ..

```
y = 9 # global y
def fun():
    global y   # this is blobal y
    print('global y=',y) # prints global y
    y = 99
    print("modified global y=",y)  # prints global new y
##
fun()
print(y)  # prints global y
```

How to access global in presence of local?

- How to access global variable in presence of local when they have same name:

```
y = 9 # global y
def fun():
    y = 7   # y is local
    print('local y=',y) # prints local y
    print('local y=',globals()['y']) # prints global y
    globals()['y'] = 99 # Update global y
##
fun()
print(y)  # prints global y
```

[local-global.ipynb]

How to pass group of elements to function

- Store elements in a list; pass that list as parameter to function
  - Note: list is immutable
- Store elements in a list; pass that list as parameter to function
  - Note: tuple is mutable

Anonymous functions

- A function without a name
- Named functions are defined using the '`def`' keyword
- But, anonymous functions do not use 'def'
  - They are defined using the '`lambda`' keyword
    - Also known as lambda functions
- Example: named function:

```
def square(x):
      return x*x
```

- Example: lambda function:

```
lambda x: x*x
```

Anonymous functions

[lambda-function.ipynb]

- Example of lambda function:

  ```
  lambda x: x*x
  ```

  - Here, the '`lambda`' keyword represents an anonymous function.
  - The '`x`' represents argument to the nameless function.
  - The '`:`' represents the beginning of the nameless function
    - that contains an expression `x*x`.
- Syntax:

  ```
  lambda argument_list : expression
  ```
- Lambda functions return a function and hence,
  - they should be assigned to a function as:

    ```
    f = lambda x : x * x
    ```

    Here '`f`' is the function name to which the lamda expression is assigned.
  - How to call the function 'f'?

    ```
    value = f(7)
    ```

Lambda functions

- Lambda function
  - contain only one statement/expression, and
  - return the result implicitly (i.e. no explicit return statement)
- Example: lambda function to calculate sum of two numbers
  - lambda-functions.ipynb
- Example: lambda function to calculate product of two numbers
  - lambda-functions.ipynb
- Example: lambda function to find maximum of two numbers
  - lambda-functions.ipynb