Functions

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/Python

Functions

- Is a block of organized, reusable code that is used to perform a single, related action.
- Provide better modularity for application and a high degree of code reusing.
- As already know, Python gives many built-in functions like `print()`, etc.
- But can also create own functions.
- These functions are called user-defined functions.

Defining a Function

- Can define functions to provide the required functionality.
- Syntax

```
def function_name (parameters ) :
        """function_docstring"""
        function_suite
        return [expression]
```

- Here are simple rules to define a function in Python.

Defining a Function (Cont.)

- Function blocks begin with the keyword **`def`** followed by the function name and parentheses (**`( )`**).
- Any input parameters or arguments should be placed within these parentheses.
- Can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or **`docstring`**.

Defining a Function (Cont.)

- The code block within every function starts with a colon (`:`) and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller.
- A return statement with no arguments is the same as return `None`.

Defining a Function (Cont.)

- The following function takes a string as input parameter and prints it on standard screen.
- Example

```
def printme( str ) :
"""This prints a passed string into this function"""
    print (str)
    return
```

Calling a Function

- Defining a function gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, can execute it by calling it from another function or directly from the Python prompt.

Calling a Function (Cont.)

```
1  # Function definition is here
2
3  def  printme (str):
4      "This prints a passed string into this function"
5      print (str)
6      return
7
8  # Now you can call printme function
9  printme("This is first call to the user defined function!")
10 printme("Again second  call to the same function")
```

Console

<terminated> demo.py [/usr/bin/python3.6]

```
This is first call to the user defined function!
Again second  call to the same function
```

Pass by Reference vs Value

- All parameters (arguments) in the Python language are ***passed by reference***.
- It means if change what a parameter refers to within a function, the change also reflects back in the calling function.

Pass by Reference vs Value (Cont.)

```
1  # Function definition is here
2
3  def changeme (mylist):
4      "This changes a passed list into this function"
5      print ("Values inside the function before change : ", mylist)
6      mylist[2] = 50
7      print ("Values inside  the function after change : ", mylist)
8      return
9
10 # Now you can call changeme function
11 mylist = [10, 20, 30]
12 changeme(mylist)
13 print ("Values outside the function : ", mylist)
```

Console

<terminated> demo.py [/usr/bin/python3.6]

```
Values inside the function before change :  [10, 20, 30]
Values inside  the function after change :  [10, 20, 50]
Values outside the function :  [10, 20, 50]
```

Pass by Reference vs Value (Cont.)

- The parameter **`mylist`** is ***local*** to the function **`changeme`**.
- Changing **`mylist`** within the function does ***not*** affect **`mylist`**.
- The function accomplishes nothing and finally this would produce the next slide result.

Pass by Reference vs Value (Cont.)

```
1  # Function definition is here
2
3  def changeme (mylist):
4      "This changes a passed list into this function"
5      mylist = [1, 2, 3, 4]  # This would assign new reference in mylist
6      print ("Values inside the function : ", mylist)
7      return
8
9  # Now you can call changeme function
10 mylist = [10, 20, 30]
11 changeme(mylist)
12 print ("Values outside the function : ", mylist)
```

Console

<terminated> demo.py [/usr/bin/python3.6]

```
Values inside the function :  [1, 2, 3, 4]
Values outside the function :  [10, 20, 30]
```

Function Arguments

- Can call a function by using the following types of formal arguments :
  - Required arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments

Required Arguments

- Are the arguments passed to a function in correct positional order.
- Here, the number of arguments in the function call should match exactly with the function definition.
- To call the function `printme()`, definitely need to pass one argument, otherwise it gives a syntax error.

Required Arguments (Cont.)

```
1 # Function definition is here
2
3 def printme (str):
4     print (str)
5
6
7 # Now you can call changeme function
8 printme()
```

Console

<terminated> demo.py [/usr/bin/python3.6]

```
Traceback (most recent call last):
  File "/home/instructor/PythonHome/0829/demo.py", line 8, in <module>
    printme()
TypeError: printme() missing 1 required positional argument: 'str'
```

Keyword Arguments

- Are related to the function calls.
- When use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- Allows 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.
- Can also make keyword calls to the `printme()` function in the following ways.

Keyword Arguments (Cont.)

```
1 # Function definition is here
2
3 def printme (str):
4     print (str)
5
6
7 # Now you can call changeme function
8 printme( str = "My string")
```

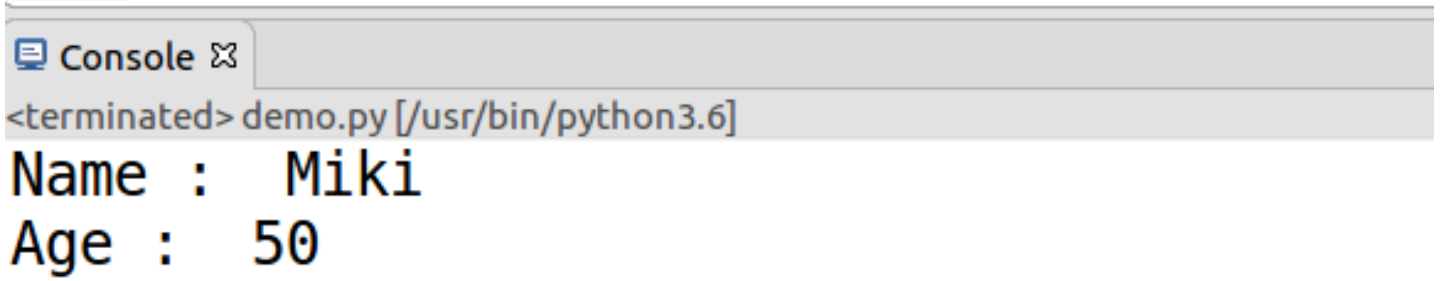
Console

<terminated> demo.py [/usr/bin/python3.6]

```
My string
```

Keyword Arguments (Cont.)

```
1 # Function definition is here
2
3 def printinfo (name, age):
4     print ("Name : ", name)
5     print ("Age : ", age)
6
7
8 # Now you can call changeme function
9 printinfo ( age = 50, name = "Miki")
```



Default Arguments

- Is an argument that assumes a default value if a value is not provided in the function call for that argument.
- The next slide example gives an idea on default arguments, it prints default `age` if it is not passed.

Default Arguments (Cont.)

```
1  # Function definition is here
2
3  def printinfo (name, age = 35):
4      print ("Name : ", name)
5      print ("Age : ", age)
6
7
8  # Now you can call printinfo function
9  printinfo ( age = 50, name = "Miki")
10 printinfo ( name = "Sally")
```

Console

<terminated> demo.py [/usr/bin/python3.6]

```
Name :  Miki
Age :  50
Name :  Sally
Age :  35
```

Variable-length Arguments

- May need to process a function for more arguments than specified while defining the function.
- Are called variable-length arguments.
- Are not named in the function definition, unlike required and default arguments.

Variable-length Arguments (Cont.)

- Syntax for a function with non-keyword variable arguments is given below :

```
def function_name ([formal_args,] *var_args_tuple) :
      function_suite
      return [expression]
```

Variable-length Arguments (Cont.)

- An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments.
- This *tuple* remains empty if no additional arguments are specified during the function call.

Variable-length Arguments (Cont.)

```
1  # Function definition is here
2
3  def printinfo ( arg1, *vartuple):
4      print ("Output is : ")
5      print (arg1)
6      for var in vartuple :
7          print (var)
8
9
10 # Now you can call printinfo function
11 printinfo ( 10 )
12 printinfo ( 70, 60, 50)
```

Console

<terminated> demo.py [/usr/bin/python3.6]

```
Output is :
10
Output is :
70
60
50
```

The Anonymous Functions

- These functions are called anonymous because they are not declared in the standard manner by using the `def` keyword.
- Can use the `lambda` keyword to create small anonymous functions.

The Anonymous Functions (Cont.)

- Lambda forms can take any number of arguments but return just one value in the form of an expression.
- They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because `lambda` requires an expression.

The Anonymous Functions (Cont.)

- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function, during invocation for performance reasons.

The Anonymous Functions (Cont.)

- Syntax
  - The syntax of **`lambda`** functions contains only a single statement.

```
lambda [arg1 [, arg2, …. Argn]] : expression
```

The Anonymous Functions (Cont.)

```
1 # Function definition is here
2
3 sum = lambda arg1, arg2 : arg1 + arg2
4
5
6 # Now you can call printinfo function
7 print ("Value of total : ", sum(10 ,20))
8 print ("Value of total : ", sum(20, 20))
```

Console

<terminated> demo.py [/usr/bin/python3.6]

```
Value of total :  30
Value of total :  40
```

The return Statement

- The statement `return [expression]` exits a function, optionally passing back an expression to the caller.
- A `return` statement with no arguments is the same as return `None`.

The return Statement (Cont.)

```
1  # Function definition is here
2
3  def sum (arg1, arg2):
4      #Add both the parameters and return them.
5      total = arg1 + arg2
6      print ("Inside the function : ", total)
7      return total
8
9
10 # Now you can call printinfo function
11 total = sum(10, 20)
12 print ("Outside the function : ", total)
```

Console

<terminated> demo.py [/usr/bin/python3.6]

```
Inside the function :  30
Outside the function :  30
```

Scope of Variables

- All variables in a program may not be accessible at all locations in that program.
- This depends on where have declared a variable.
- The scope of a variable determines the portion of the program where you can access a particular identifier.
- There are two basic scopes of variables in Python :
  - Global variables
  - Local variables

Global vs. Local variables

- Variables that are defined inside a function body have a ***local*** scope, and those defined outside have a ***global*** scope.
- This means that ***local*** variables can be accessed only inside the function in which they are declared, whereas ***global*** variables can be accessed throughout the program body by all functions.
- When call a function, the variables declared inside it are brought into scope.

Global vs. Local variables (Cont.)

```
1  # Function definition is here
2
3  total = 0        # This is global variable.
4
5  # Function definition is here
6  def sum (arg1, arg2):
7      # Add both the parameters and return them.
8      total = arg1 + arg2    # here total is local variable
9      print ("Inside the function local total : ", total)
10     return total
11
12
13 # Now you can call sum function
14 sum (10, 20)
15 print ("Outside the function global total : ", total)
```

Console

<terminated> demo.py [/usr/bin/python3.6]

```
Inside the function local total :  30
Outside the function global total :  0
```

Lab

```
1
2  def greet_user() :
3      """간단한 환영 인사를 표시한다"""
4      print ('Hello!')
5
6  greet_user()
7
```

Lab (Cont.)

```
1
2 def greet_user( username ) :
3     """간단한 환영 인사를 표시한다"""
4     print ('Hello, ' + username.title() + '!')
5
6 greet_user('Jesse')
7
```

Lab (Cont.)

```
1 def add_number(n1, n2):
2     ret = n1 + n2
3     return ret
4
5 def add_txt(t1, t2):
6     print(t1 + t2)
7
8 ans = add_number(10, 15)
9 print(ans)                    # 25가 출력됨
10 text1 = 'Hello '
11 text2 ='World'
12 add_txt(text1, text2)        # 'Hello World'가 출력됨
13
```

Lab (Cont.)

```
1
2 def describe_pet (animal_type,  pet_name):
3     """애완동물에 관한 정보를 출력한다"""
4     print ('\nI have a ' + animal_type + '.')
5     print ("My " + animal_type + "'s name is " + pet_name.title() + ".")
6
7 describe_pet('hamster', 'harry')
8 describe_pet('dog', 'willie')
9
```

Lab (Cont.)

```
1
2 def describe_pet (animal_type,  pet_name):
3     """애완동물에 관한 정보를 출력한다"""
4     print ('\nI have a ' + animal_type + '.')
5     print ("My " + animal_type + "'s name is " + pet_name.title() + ".")
6
7 describe_pet('harry', 'hamster')
8
```

Lab (Cont.)

```
1
2 def describe_pet (animal_type,  pet_name):
3     """애완동물에 관한 정보를 출력한다"""
4     print ('\nI have a ' + animal_type + '.')
5     print ("My " + animal_type + "'s name is " + pet_name.title() + ".")
6
7 describe_pet(animal_type = 'hamster', pet_name = 'harry')
8 describe_pet(pet_name = 'harry', animal_type = 'hamster')
9
```

Lab (Cont.)

```
1
2  def describe_pet (pet_name, animal_type = 'dog'):
3      """애완동물에 관한 정보를 출력한다"""
4      print ('\nI have a ' + animal_type + '.')
5      print ("My " + animal_type + "'s name is " + pet_name.title() + ".")
6
7  describe_pet(pet_name = 'willie')
8
```

Lab (Cont.)

```
1
2  def describe_pet (pet_name, animal_type = 'dog'):
3      """애완동물에 관한 정보를 출력한다"""
4      print ('\nI have a ' + animal_type + '.')
5      print ("My " + animal_type + "'s name is " + pet_name.title() + ".")
6
7  # 윌리라는 개
8  describe_pet('willie')
9  describe_pet(pet_name = 'willie')
10
11 # 햄스터 해리
12 describe_pet('harry', 'hamster')
13 describe_pet(pet_name = 'harry', animal_type = 'hamster')
14 describe_pet(animal_type = 'hamster', pet_name = 'harry')
15
```

Lab (Cont.)

```
1
2  def build_person ( first_name, last_name):
3      """사람에 관한 정보 딕셔너리를 반환하기"""
4      person = {'first' : first_name, 'last' : last_name}
5      return person
6
7
8  musician = build_person('Jimi', 'Hendrix')
9  print (musician)
10
```

Lab (Cont.)

```
1
2  def build_person ( first_name, last_name, age = ''):
3      """사람에 관한 정보 딕셔너리를 반환하기"""
4      person = {'first' : first_name, 'last' : last_name}
5      if age : person['age'] = age
6      return person
7
8
9  musician = build_person('Jimi', 'Hendrix', age = 27)
10 print (musician)
11
```

Lab (Cont.)

```
1
2 def make_pizza(*toppings):
3     """주문 받은 토핑 리스트 출력하기"""
4     print (toppings)
5
6 make_pizza('pepperoni')
7 make_pizza('mushrooms', 'green peppers', 'extra cheese')
8
```

Lab (Cont.)

```
1
2  def make_pizza(*toppings):
3      """만들려고 하는 피자를 요약한다"""
4      print ("\nMaking a pizza with the following toppings : ")
5      for topping in toppings :
6          print("- " + topping)
7
8  make_pizza('pepperoni')
9  make_pizza('mushrooms', 'green peppers', 'extra cheese')
10
```

Lab (Cont.)

```
1  def add_txt(t1, t2='Python'):
2      print(t1 + ' : ' + t2)
3
4  add_txt('Best language : ')
5  add_txt(t2 = 'Java', t1='Good Language : ')
6
7  def func1(*args):
8      print(args)
9
10 def func2(width, height, **kwargs):
11     print(kwargs)
12
13 func1()                              # 빈 튜플 ( ) 이 출력됨
14 func1(3, 5, 1, 5)                     # (3, 5, 1, 5)가 출력됨
15
16 func2(10, 20)                                 # 빈 사전 { }이 출력됨
17 func2(10, 20, depth=50, color='blue')  #{'depth':50, 'color':'blue'} 이 출력됨
18
```

Lab (Cont.)

```
1  param = 10
2  strdata = '전역변수'
3
4  def func1():
5      strdata = '지역변수'
6      print(strdata)
7
8  def func2(param):
9      param = 1
10
11 def func3():
12     global param
13     param = 50
14
15 func1()                    # '지역변수'가 출력됨
16 print(strdata)              # '전역변수'가 출력됨
17 print(param)                # 10이 출력됨
18 func2(param)
19 print(param)                # 10이 출력됨
20 func3()
21 print(param)                # 50이 출력됨
22
```

Lab (Cont.)

```
1 def reverse(x, y, z):
2     return z, y, x
3
4 ret = reverse(1, 2, 3)
5 print(ret)                          # (3, 2, 1)이 출력됨
6
7 r1, r2, r3 = reverse('a', 'b', 'c')
8 print(r1); print(r2); print(r3)      # 'c', 'b', 'a' 순으로 출력됨
9
```

Lab (Cont.)

```
1 a = [1,2,3]
2 b = a
3 a[0] = 100
4 print (a)
5 print (b)
6
7 print("id(a) = ", id(a), ", id(b) = ", id(b))
8
```

Lab (Cont.)

```
1 a = [1,2,3]
2 b = a[:]
3 a[0] = 100
4 print (a)
5 print (b)
6
7 print("id(a) = ", id(a), ", id(b) = ", id(b))
8
```

Lab (Cont.)

```
1 import copy
2
3 a = [1,2,3]
4 b = copy.deepcopy(a)
5 a[0] = 100
6 print (a)
7 print (b)
8
9 print("id(a) = ", id(a), ", id(b) = ", id(b))
10
```

Lab (Cont.)

```
1 def change(var):
2     print ("Before change in function : ", var)
3     var[0] = 100
4     print ("After change in function : ", var)
5
6 a = [1, 2, 3]
7 print ("Before call function : ", a)
8 change(a)
9 print ("After call function : ", a)
10
```

Lab (Cont.)

```
1 def change(var):
2     print ("Before change in function : ", var)
3     var[0] = 100
4     print ("After change in function : ", var)
5
6 a = [1, 2, 3]
7 print ("Before call function : ", a)
8 change(a[:])
9 print ("After call function : ", a)
10
```

Lab (Cont.)

```
2  def times(a, b):
3      return a * b
4
5  print (times)
6  print (times(10 ,10))
7
8  print(globals())
9
10 myTimes = times
11
12 r = myTimes(10, 10)
13 print (r)
14
15 print (globals())
16
```

```
2 g = lambda x, y : x * y
3
4 print (g(2, 3))
5
6 print ((lambda x : x * x)(3))
7
8 print (globals())
9
```