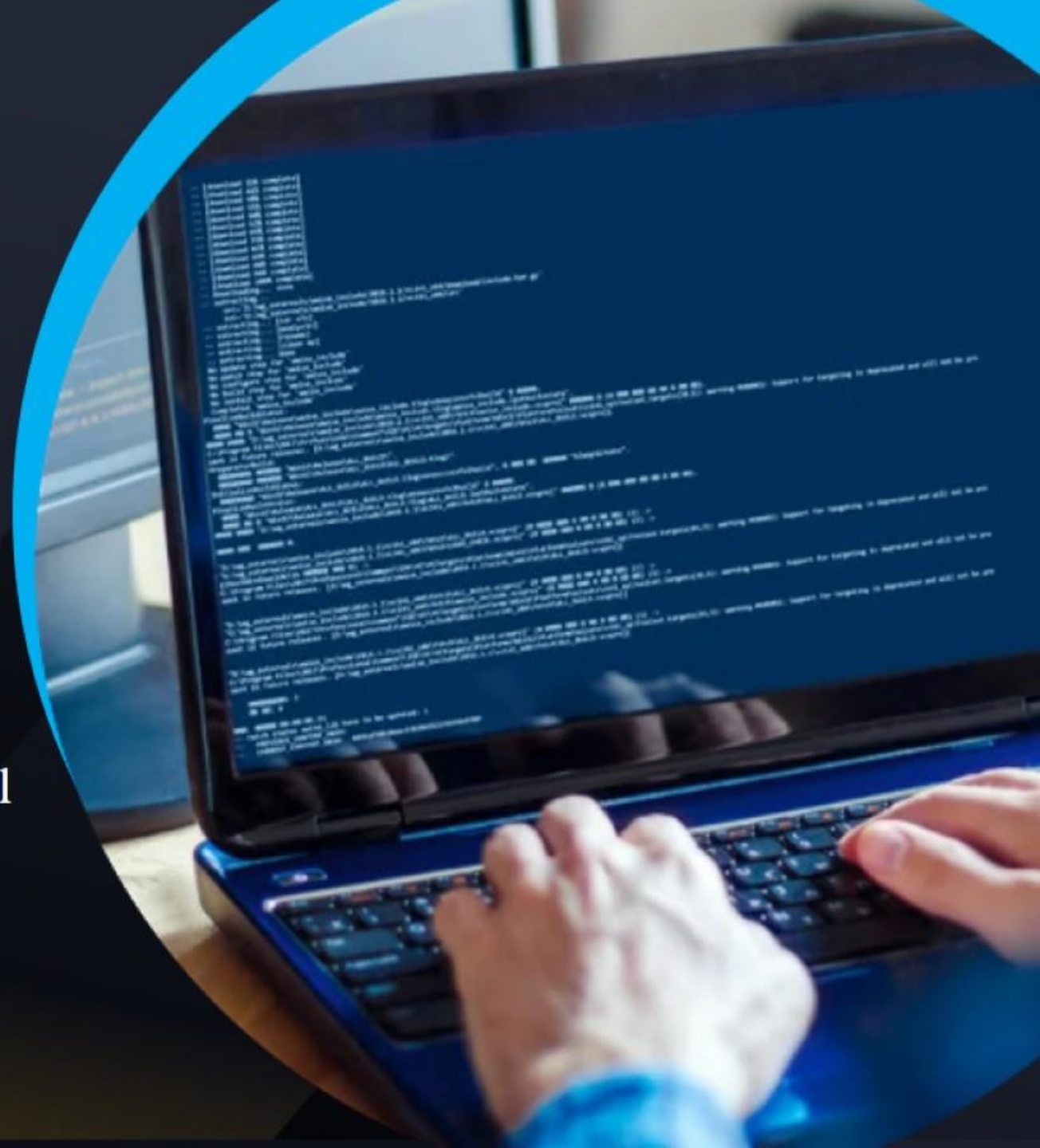


Programming Foundations in Python

Prepared by Dr. Hamada I. AbdulWakel

Lecture 7



Functions and Modules

Lecture 7

Prepared by
Dr. Hamada I. AbdulWakel

www.midocean.edu.km

Lecture

07

RECAP: LECTURE 6

- The creation of the user-defined functions.
- The local and global variables.
- The top-down program design.

Lecture

07

LECTURE MAIN POINTS

- The function names and the default arguments.
- The inner and high-order functions.
- The use of anonymous functions
- The use of modules.

FUNCTION NAMES

Cont'd

- Function names are references to functions and that we can assign multiple names to the same function.
- Functions are stored in variables like everything else.

```
def increment(x):  
    return x + 1  
Myincrease = increment  
print(Myincrease)  
print(increase(5))
```

Lecture

07

FUNCTION WITH DEFAULT
ARGUMENTS

- Arguments provide the function's caller with the means of transmitting information to the function.

```
def function(x, y):  
    pass    # pass keyword used to ignore the body of the function
```

- Programmer can specify arguments with default values in any function definition using the **= operator** to provide default values.

```
def function( x , y = 5):  
    pass
```

FUNCTION WITH DEFAULT ARGUMENTS

Cont'd

- Following the required arguments are one or more default arguments.

```
def function(y = 5, x): #SyntaxError  
    pass
```

- When function is called with these arguments, default values are **overridden** by caller's values.

```
function(2)  
function(2, 3)
```

FUNCTION WITH DEFAULT ARGUMENTS

Cont'd

- Example1:

```
def greet(name, msg):  
    print( "Hello" , name + ', ' + msg)
```

```
greet("Monica", "Good morning!")
```

```
greet() #TypeError: greet() missing 2 required positional arguments
```

FUNCTION WITH DEFAULT ARGUMENTS

Cont'd

- Example2:

```
def greet(name, desc="Mr."):
    print("Hello,", desc, name)
```

```
greet("Ahmed")
greet("Ahmed", "Dr.")
```

FUNCTION KEYWORD ARGUMENTS

Cont'd

- 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**.

```
def display_info(first_name, last_name):  
    print('First Name:', first_name)  
    print('Last Name:', last_name)
```

```
display_info(last_name = 'MIDOCEAN University', first_name = 'ESU')
```

FUNCTION KEYWORD ARGUMENTS

Cont'd

- Example3:

```
def accessArguments(x, y = 1, z = 2):  
    print("X= ", x, "Y= ", y, "Z= ", z)
```

```
accessArguments(3)
```

```
accessArguments(3, 10)
```

```
accessArguments(3, z = 5)
```

```
accessArguments(3, z = 4, y = 6)
```

FUNCTION WITH ARBITRARY ARGUMENTS

Cont'd

- Sometimes, we **do not know** in advance the number of arguments that will be passed into a function.
- In the function definition, we use an asterisk `*` before the parameter name to denote this kind of argument.
- The `*args` syntax allows a function to receive any number of positional arguments as a **tuple**.

FUNCTION WITH ARBITRARY ARGUMENTS

Cont'd

- Example4:

```
def greet(*names):  
    for name in names:  
        print("Hello," , name)
```

```
greet("Ahmed", "Mohamed")  
greet("Ahmed", "Mohamed", "Aly")  
greet("Ahmed", "Mohamed", "Aly", "Mahmoud")
```

FUNCTION WITH ARBITRARY ARGUMENTS

Cont'd

- Example5:

```
def multiply(*numbers):  
    result = 1  
    for num in numbers:  
        result *= num  
    return result
```

```
multiply(2, 3)
```

```
multiply(2, 3, 4, 5, 6, 7, 8, 9) #and more
```

INNER/NESTED FUNCTIONS

Cont'd

- Inner/nested functions, are functions that you define inside other functions.
- This kind of function has direct access to variables and names defined in the enclosing function.

INNER/NESTED FUNCTIONS

Cont'd

```
def outer():      ← step 2
    b = 20        ← step 3
    def inner():  ← step 4, 6
        c = 30    ← step 7
        print(b + c) ← step 8
    inner()       ← step 5
```

#main() function

```
outer()          ← step 1
```

INNER/NESTED FUNCTIONS

Cont'd

- Example6:

```
def outer_func(from):  
    def inner_func():  
        print("Hello from,", from)  
    inner_func()  
  
outer_func("MIDOCEAN University")
```

INNER/NESTED FUNCTIONS

Cont'd

- Example7: calculate factorial to a number using inner function.

```
def factorial(number):  
    def inner_factorial(number):  
        if number <= 1:  
            return 1  
        return number * inner_factorial(number - 1)  
    return inner_factorial(number)  
factorial(5)
```

HIGHER ORDER FUNCTIONS

Cont'd

- A function is called a higher-order function in these cases:
 - When it **takes** one or more functions as arguments.
 - When it **returns** as a result of another function.

HIGHER ORDER FUNCTIONS

Cont'd

- Example8:

```
def loud(text):  
    return text.upper()  
def quiet(text):  
    return text.lower()  
def hello(func):  
    text = func("Hello")  
    print(text)  
hello(loud)
```

HIGHER ORDER FUNCTIONS

Cont'd

- Example9: $10 \text{ (dividend)} / 2 \text{ (divisor)} = 5 \text{ (quotient)}$

```
def divisor(x):  
    def dividend(y):  
        return y / x  
    return dividend
```

```
divide = divisor(2)  
print(divide(10))
```

HIGHER ORDER FUNCTIONS

Cont'd

- Example10:

```
def add_two_nums(x, y):  
    return x + y  
def add_three_nums(x, y, z):  
    return x + y + z  
def get_appropriate_function(num_len):  
    if num_len == 3:  
        return add_three_nums  
    else:  
        return add_two_nums
```

```
args = [1, 2, 3]
```

```
num_len = len(args)
```

```
res_function = get_appropriate_function  
(num_len)
```

```
print(res_function)
```

```
print(res_function(*args))
```

ANONYMOUS FUNCTIONS

Cont'd

- An anonymous function is a function that is defined **without a name**.
- Anonymous functions are defined using the **lambda** keyword.
- Lambda functions can have any number of arguments but **only one expression**.

lambda arguments: expression

ANONYMOUS FUNCTIONS

Cont'd

- A lambda function is used when we require a nameless function for a short period of time.
- It is used as an argument to a higher-order function.

ANONYMOUS FUNCTIONS

Cont'd

- Example11:

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

- This function has no name. It returns a **function object** which is assigned to the identifier **double**. We can now call it as a normal function.

ANONYMOUS FUNCTIONS

Cont'd

- Example12:

```
double = lambda x: x * 2
```

is equal to

```
def double(x):  
    return x * 2
```

ANONYMOUS FUNCTIONS

Cont'd

- Example13: $f(x, y) = x + 2y$

```
f = lambda x, y: x + 2 * y  
print( f(2, 3) )
```

ANONYMOUS FUNCTIONS

Cont'd

- Example14: write a lambda function to **filter** out only the even items 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)
```

ANONYMOUS FUNCTIONS

Cont'd

- Example15: write a lambda function to double each item in a list using **map()**

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

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

```
print(new_list)
```

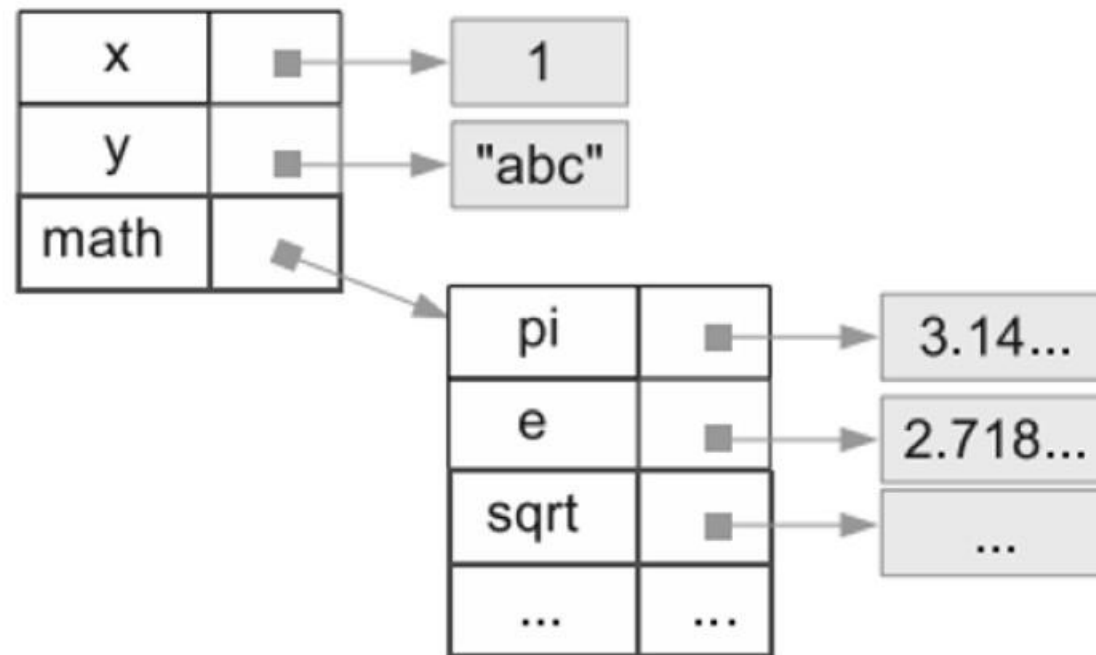
MODULES IN PYTHON

Cont'd

- Modules are assemblies of code that provide specific functionalities and constants for **reuse**.
- Adding a module to your program helping you doing some tasks to save time and effort, where you can add/call multiple modules.
- Constants and functions defined in a module can be accessed by using the module name and **the dot (.) operator**.
`<module>.<identifier>`

MODULES IN PYTHON

Cont'd



MODULES IN PYTHON

Cont'd

- Calling modules in a program:

```
import module_name1, module_name2, .....
```

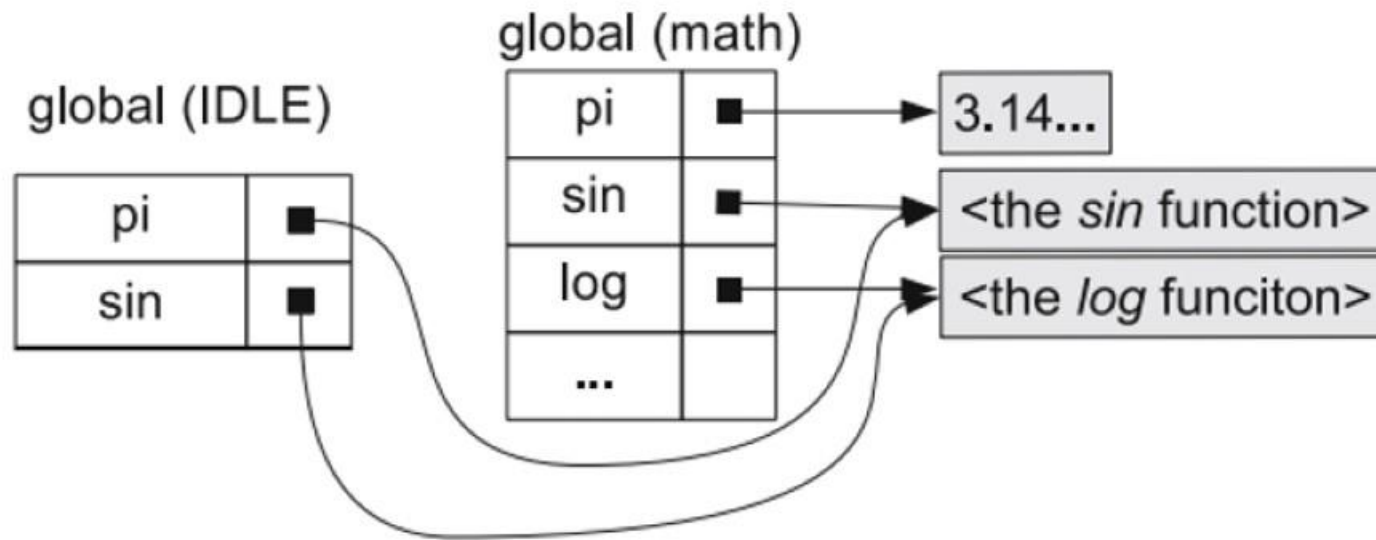
- Example 16.

```
import random  
print(random)  
print(random.random())  
print(dir(random))
```

MODULES IN PYTHON

Cont'd

- Sometimes only a small number of functions or variables needs to be imported from a module. The reason is to minimize the size of the code



MODULES IN PYTHON

Cont'd

- Calling a specific function from a module:

```
from module_name import function_name1, ....
```

- Example 17.

```
from random import randint  
print(random.randint(100, 500)) #get an error  
print(randint(100, 500))
```

- This module provides functions for:
 - Creating and removing a directory (folder),
 - Fetching its contents, and
 - Changing and identifying the current directory.

- Getting current working directory:

```
import os  
print(os.getcwd())
```

- Creating a directory:

```
os.mkdir("/path/to/NewFolder")  
os.mkdir("NewFolder")
```

- Changing the current working directory:

```
os.chdir("/path/to/folder")
```

```
os.getcwd()
```

- Removing a directory:

```
os.rmdir("/path/to/folder")
```

#you can't remove the current working directory

- Listing files and sub-directories in the current working directory:

```
os.listdir("/path/to/folder")
```

```
os.listdir()
```

- Removing a directory:

```
os.rmdir("/path/to/folder")
```

```
#you can't remove the current working directory
```

Lecture

07

QUESTIONS FOR ONLINE DISCUSSION

- Write code to select a random element from a list. Use `random.choice()`.
- Write code to square and cube every number in a given list of integers using Lambda.
- Use `map` to create a new list by changing each country to uppercase in the countries list.
- Use `filter` to filter out countries having exactly six characters.



Thank You

