Fundamentals of Programming 1

Lecture 9

Aurelia Power, ITB, 2018

Attributes of Variables

✓ So far, we have used consistently several attributes of a variable; they include:

- **Name/Identifier**
- **Value and data type once a value is assigned**

✓ Example:

```
inumberOfCoffees  = 10;
```

Name/Identifier

Data type and value: int 10

✓ But each variable has other attributes, and we will discuss next :

- **Scope**

Scope

- An variable's **scope** refers to where the variable can be referenced in a program/script, that is, it refers to where the program can use that variable.
- In other words, the scope of an identifier is the portion of the program in which the identifier can be referenced/used.
- Some identifiers/variables can be referenced/used throughout the entire program and, in this case, they have global scope.
- Others can be referenced from/used only in a function, and in this case they have local scope.
- A **local variable is a variable** defined in a function and it can be referenced/accessed only in that function.
- So, any variable assigned a value in a function has local scope, restricted to that function.

Local Scope

- Function parameters and variables assigned a value inside a function all **have local scope and they can only be accessed inside the defining function**.

```
def avg(n1, n2, n3):
    average = (n1 + n2 + n3) / 3
    return average
print(n1) ## will give a NameError: name 'n1' is not defined
print(average) ## will also give NameError: name 'average' is not defined
```

The function's scope which applies to *n1, n2, n3*, and *average*

- So n1, n2, n3 and average can only be accessed inside the avg function

- Local scope begins from the name of the function and goes to (and including) the last statement which are denoted **by at least one different level of indentation.**

- The period of time a variable exists in memory is called **lifetime**.
- Local variables and parameters have a restricted lifetime: they are created (allocated memory) when the residing function is called and once the function terminated (returns a value or control) they are destroyed (deallocated).

Global Scope

- But we can also have global variables which are defined outside a function and are said to have global scope: their scope begins from where they were first defined and assigned a value since their lifetime begins from the time :

```
print('hello world')
average = 0  ## a global variable average is created and it's scope starts here, and
             ## lasts till the end of the program/script
def avg(n1, n2, n3):
        average = (n1 + n2 + n3) / 3 ## another variable also called average will be
                                     ## created here; this has local scope
        return average
print(average) ## will output 0
average = avg(3, 2 , 4) ## will re-assign a diferent value to the variable average: 3.0
print(average) ## will output 3.0
```

Global scope of the variable *average*

- In the case of a function defining and assigning a value to a variable that has the same name, **inside the function, the local variable will be used**.
- **Outside the function, the global variable is used.**

Global Scope

- But we can also have a global variable which is defined outside a function and can be accessed inside a function as long as there is no local variable with the same name:

```
print('hello world')
minim = 0
def f1(n):
        if n < minim: ## here the global variable minim is used
                ## do something here
def f2(x):
        while x < minim: ## again, here the global variable minim is used
                ## do something here repeatedly
```

- So the variable *minim* can be accessed directly by both functions f1 and f2 and thus can be shared by both functions
- However, because they can access it directly, any of the functions defined in a program may be able to alter a global variable, even though this action may not be intended by both functions
- So, generally it is bad practice to use global variables.

Global Scope

- Another reason why global variables are not good practice is code re-usability: if a function relies on a global variable defined in a program, these functions may not be reliably used in another program since that global variable does not exist in the second program

- So the previous script can be modified in such a way that instead of using a global variable, an additional parameter may be defined by each function which will have local scope:

```
def f1(n, minim):
        if n < minim: ## here the local parameter minim is used
                ## do something here
def f2(x, minim):
        while x < minim: ## again, here the local parameter minim is used
                ## do something here repeatedly
```

```
a = 'global'
def g1(n=3, s='*'):
    return s * n

def g2(n=1):
    for i in range(n):
        print(a)

def g3(s='hello'):
    total=0
    for i in range(0, len(s), 2):
        total += i
    return total

print(g1())
print(g1(2))
print(g1(10, '-'))
g2(3)
g2()
print(g3('fop'))
print(g3())
```

Exercises:

- What does g1 do?
- What does g2 do?
- What does g3 do?

- What is the scope of the variable a?
- What is the scope of the variable n?

- Are s in g1 and s in g2 the same parameters?

- Can the variable i be accessed by g1?

- Identify the arguments and their respective kind in each function call.

- What is the code on the left going to output??

Re-usable Functions and Modular Design

- So far we have used/invoked/called individual functions over and over again in the same program/ script.
- However, in complex applications, programs/scripts are organised at a higher level as a set of modules.
- Each module contains a set of related functions (and/or objects – which are not covered here).
- So a **Python module** is a file containing related definitions and statements.
- An example is the Python standard library which is organised into modules, each module dealing with a specific area.
- For instance, we have used the **random** module (the source code is found in Lib/random.py), and one of its functions **randint**

Re-usable Functions and Modular Design

- Each module can be used only after it has been explicitly imported as follows:

  import module_name

- In addition, we need to provide the fully qualified name of the function when we invoke it as follows: name of the module, followed by a dot, followed by the name of the function

  module_name.function_name()

- EXAMPLE:

```
import random
print(random.randint(2, 10)) ## may output 7
```

- Other commonly used modules are:
  - **math** which defines mathematical functions
  - **string** which defines common string operations
  - **statistics** which defines statistical functions
  - and many more…
- EXERCISE: identify a module from the standard library and describe 2 functions of your choice in terms of arguments and returned value. Then create a script and invoke those 2 functions

Re-usable Functions and Modular Design

- Python provides an alternative way of importing items from different modules and it uses 2 keywords: from and import

  from module_name import function_name

- In this case, when invoking the function name we don't need to provide the fully qualified name, just the name of the function:

  function_name()

EXAMPLE:

```
from math import factorial
print(factorial(7)) ## outputs 5040
```

- You can also import all the members of a module by using *: **from math import ***

- In Python, each module has is own **namespace** which is a container that provides the same named context for a set of related identifiers, including those that apply to functions and global variables.
- The namespace is provided by the name of the file/module.
- Namespaces avoid any potential name clashes because they allow us to use the fully qualified name of an item.

Re-usable Functions and Modular Design

- We can create our own modules to define related functionalities, and then we can use them later in a different script using the keyword import and the fully qualified name of the function.

module_1.py – contains 2 functions that display welcome messages

File Edit Format Run Options Window Help

```
def print_simple_welcome():
    print("Hello students of ITB!")
    print("Welcome to the FOP1 module!")
    print("This module will teach you how to program using Python.")
    print("And, really, programming is awesome!!!")

def print_formatted_welcome():
    print(format("Hello students of ITB!", "^100"))
    print(format("Welcome to the FOP1 module!", "^100"))
    print(format("This module will teach you how to program using Python.",
                 "^100"))
    print(format("And, really, programming is awesome!!!", "^100"))
```

```
def print_odd_numbers(a, b):
    if a % 2 == 0:
        a += 1
    for i in range (a, b + 1, 2):
        print(i, end=' ')
    print()

def maxim(n1, n2, n3):
    m = n1
    if m < n2:
        m = n2
    if m < n3:
        m = n3
    return m

def display_cumulative_sums(a, b):
    total = 0;
    for x in range(a, b + 1):
        total += x
        print(total, end=' ')
    print()
```

- **module_2.py** contains 3 functions:

1. for printing odd numbers within a given range,
2. for returning the maximum value among 3 numerical values,
3. for displaying the cumulative sums of numerical values within a given range

```
##import module_1
import module_1
## use the fully qualified names of the functions
module_1.print_simple_welcome()
print()
module_1.print_formatted_welcome()

## import everything from module_2
from module_2 import *
## use only the name of the functions
print_odd_numbers(2, 12)
print(maxim(-10, 77, 34))
display_cumulative_sums(4, 7)
```

```
======= RESTART: C:/Users/Aurelia Power/Desktop/2018_2019/FOP1/main.py =======
Hello students of ITB!
Welcome to the FOP1 module!
This module will teach you how to program using Python.
And, really, programming is awesome!!!

                    Hello students of ITB!
                Welcome to the FOP1 module!
            This module will teach you how to program using Python.
                And, really, programming is awesome!!!
3 5 7 9 11
77
4 9 15 22
>>>
```

- **main.py** contains the *import statements*, as well as the *function calls* from each module: module_1 and module_2
- **main.py** is the module that needs to be run and when run it will display the output shown on the left

Your turn… TRUE or FALSE??

- Every function must have at least one mutable parameter
- All value-returning functions must contain at least one parameter
- Function calls may contain arguments that are function calls.
- An expression may contain more than one function call
- A global variable is a variable that is defined outside of any function definition.
- The use of global variables is a good way to allow different functions to access and modify the same variables.
- With the "import moduleName" form of import, any utilized entities from the imported module must be prefixed with the module name.

Your turn…

- Define 2 different modules
- The first module should contain 2 functions :
  - One that returns true if a given number is negative, false if it is negative
  - One that returns the absolute value of a number.
- The second module should contain 3 functions :
  - One that displays the squared values of all numbers within a given range (both ends inclusive).
  - One that finds the factorial value, given an integer.
  - One that looks takes in a number n and calculates how many possible combinations of digits can be generated using n amount of digits.
- The main module should contain at least 2 calls for each function defined in the previous module; you should appropriately use import statements and/or fully qualified names when invoking the functions.

Sample Practical Test

1. Write a program that contains **2 functions**:

- One that takes in an whole number n and prints a square of dashes, with the size of rows and columns determined by that number n; for instance, if 2 is passed as argument, the following should be displayed:

  \- -

  \- -

- One that takes in a whole number n and generates the multiplication table nXn; for instance, if the user enters 3:

  1 2 3

  2 4 6

  3 6 9

- Call these functions twice each with different values.

Sample Practical Test ctd.

2. Write another program that also has **2 functions**:

- One that takes in a string and returns whether it contains any vowels; for instance, if 'hello' is passed as argument, it should return True because it contains 2 vowels: 'e' and 'o', but if 'sshhhhh' is passed, the function should return False because it has no vowels (for simplicity reasons, you can assume that we deal only with lower case).
- One that takes in a string and returns whether it contains the sequence "ar"; for instance, if 'car' is passed as argument, the method should return True, if 'johnny' is passed, it should return False (again, for simplicity reasons, you can assume that we deal only with lower case).

- Then call/invoke each function twice with different values.