CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computer Science

http://www.cit.ie

Programming for Data Analytics

Week3: Functions in Python

Dr. Haithem Afli

Haithem. afli@cit.ie

@AfliHaithem

2018/2019

Content – Part 1

- Introduction to Functions
- Defining and Calling a Function
- Designing a Program to Use Functions
- Local Variables
- Passing Arguments to Functions
- Global Variables and Global Constants

Content – Part 2

- Introduction to Value-returning Functions:
- Writing Your Own Value-Returning Functions
- The math Module
- Storing Functions in Modules

Introduction to Functions

- Most programs perform tasks that are large enough to be broken down into **several subtasks**.

- For this reason, programmers usually break down their programs into small manageable pieces of code known as functions.

Function: Group of statements within a program that perform as specific task

Using functions to divide and conquer a large task

This program is one long, complex sequence of statements.

```
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
```

In this program the task has been divided into smaller tasks, each of which is performed by a separate function.

```
def function1():
    statement
    statement
    statement
```
function

```
def function2():
    statement
    statement
    statement
```
function

```
def function3():
    statement
    statement
    statement
```
function

```
def function4():
    statement
    statement
    statement
```
function

Benefits of Modularizing a Program with Functions

- The benefits of using functions include:
  - More organized code
  - Easier to read code
  - Code reuse
    - Write the code once and call it multiple times
  - Better testing and debugging
    - Can test and debug each function individually
  - Easier facilitation of teamwork
    - Different team members can write different functions
    - Faster development

Defining and Calling a Function

- Function name should be **descriptive** of the task carried out by the function
  - Often includes a verb

- Function definition: specifies what a function does

```
def function_name():
        statement
        statement
```

Function header: first line of function.
Includes keyword **def** and **function name**, followed by parentheses and colon

Under the function header will appear an **indented** block of statements that belong together as a group

Blank lines that appear in a block are **ignored**

Defining and Calling a Function

- Call a function to execute it
  - When a function is called:
    - Interpreter jumps to the function and executes statements in the block
    - Interpreter jumps back to part of program that called the function
      - Known as function return

```
def helloWorld():
    print ('hello world')


helloWorld()
```

This statement calls the helloWorld function and causes it to execute

Defining and Calling a Function

- It is common for a program to have a main function that is called when the program starts.

- `main` function: called when the program starts
  - Calls other functions when they are needed

```
def main():
    helloWorld()
    print ("GoodBye")

def helloWorld():
    print ("hello world")

main()
```

Notice that the main function is called first. The main function then calls the helloWorld function.

The interpreter then returns to the main function.

Positioning of Functions

- The following code will **not work**. It will produce the following error:
  - NameError: name 'main' is not defined
- The interpreter executes the code from top to bottom. The first instruction it encounters is to execute a function called main. However, it has not yet seen or loaded this function.
- Therefore, you should make sure that the function definition is loaded by the interpreter before it is called.

```
main()

def main():
    helloWorld()
    print ("GoodBye")

def helloWorld():
    print ("hello world")
```

Task

- Write a program with a main function that then calls a function called squareNumbers.
- The function squareNumbers should ask the user for a number and print out the value of the number squared.

Task

- Write a program with a main function that then calls a function called squareNumbers.
- The function squareNumbers should ask the user for a number and print out the value of the number squared.

```
def squareNumbers():
    num = int(input("Please enter a number"))
    print (num**2)

def main():
    squareNumbers()

main()
```

Local Variables

- Local variable: variable that is assigned a value inside a function
  - Belongs to the function in which it was created
    - Only statements inside that function can access it, error will occur if another function tries to access the variable

- Scope: the part of a program in which a variable may be accessed
  - The scope of a local variable is the function in which it is created

Local Variables

```
def getName ( ) :
    name = input("Enter your name :")

def main():
    getName()
    print ( name )

# Call the main function.
main()
```

Line will cause an error. The variable name is not visible in the method main.

Different functions may have local variables with the **same name**

– Each function does not see the other function's local variables, so no confusion

```
def main():
    name = 'John'
    print ('hello', name)
    newGreeting()
    print ('hello', name)

def newGreeting():
    name = 'Fred'
    print ('hello', name)

# Call the main function.
main()
```

Two separate variables called name. One is local to the main and the other is local to newGreeting.

```
hello John
hello Fred
hello John
```

Passing Arguments to Functions

- Notice in all the function we have use so far when we call the function we give the function name and an empty brackets.

- However, consider the **input** function, which we have used extensively.

- Notice when we call the input function we place some data between the open and close brackets. This data is called an argument and is passes to and used by the function we call.

Passing Arguments to Functions

- **Argument**: piece of data that is sent into a function
  - Function can use argument in calculations
  - When calling the function, the **argument** is placed **in parentheses** following the function name
  - `functionName(argument)`

- **Parameter** variable: variable that is **assigned the value of an argument** when the function is called.
  - The parameter and the argument reference the same value
  - General format:

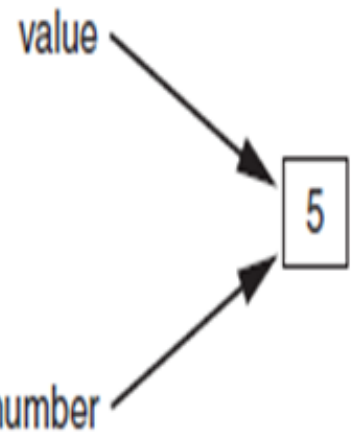
    ```
    def function_name(parameter):
    ```

  - Scope of a parameter: the function in which the parameter is used

Passing Arguments to Functions

```
def main():
    value = 5
    show_double(value)


def show_double(number):
    result = number * 2
    print(result)
```



In the main function, local variable called *value* is assigned value of 5.

When the show_double function is called the variable value appears inside the parentheses. This means that value is being passed as an **argument** to the show_double function

When this statement executes, the show_double function will be called and the ***number* parameter will be assigned the same value as the *value* variable**.

```
def printGreeting(name):
    print ("Hello", name)

def main():
    name = input("Enter name: ")
    printGreeting(name)

main()
```

Passing Multiple Arguments

- Python allows writing a function that accepts **multiple arguments**
  - Parameter list replaces single parameter
    - Parameter list items separated by comma

- Arguments are passed ***by position*** to corresponding parameters
  - First parameter receives value of first argument, second parameter receives value of second argument, etc.

- In the example on the next slide we write a function that will calculate the area of a rectangle. It will take in two parameters and print out their product.

Passing Multiple Arguments (cont'd.)

```
def main():
    length = int(input("Enter length of rectangle"))
    width = int(input("Enter width of rectangle"))
    calculateArea(length, width)


def calculateArea(recLength, recWidth):
    print (recLength* recWidth)

main()
```

You can think of the process as the value of the argument length being copied and stored in parameter recLength. Value of width is copied to recWidth.

Passing Multiple Arguments

```
def main():
    firstName = input("Enter your first name:")
    lastName = input("Enter your last name:")
    print ("Your name reversed is")
    reverseName(firstName, lastName)

def reverseName(first, last):
    print (last, first)

# Call the main function.
main()
```

Data types

- In general, data types in Python can be distinguished based on whether objects of the type are mutable or immutable.
- The content of objects of immutable types cannot be changed after they are created.

Immutable
int
float
str
Boolean

Data types

- It is important to understand that variables in Python are really just references to objects in memory. If you assign an object to a variable as shown below, this variable num just points to the int object in memory.

```
num = 10
```

- If you reassign a variable as we do with num below you create a new int object with the value 12. The variable now points to this (different) object in memory.

```
num = 10
num = 12
```

Making Changes to Parameters

- Important: Changes made to an immutable data type parameter within the function do not affect the original argument itself

- Changes made to a mutable data type parameter will be reflected in the original argument.

```
def main():
    userAge = 25
    print ('User Age is ', userAge)

    updateAge(userAge)
    print ('User Age is now', userAge)

def updateAge(age):
    age+=1
    print ('Updated Age to ', age)


# Call the main function.
main()
```

The output of the program is:

User Age is 25
Updated Age to 26
User Age is now 25

Notice the change made to the age variable in updateAge has no impact on the value of the userAge variable in main()

Global Variables

- Global variable: created by assignment statement written **outside all the functions**
  - Can be accessed by any statement in the program file, including from within a function

```
# Create a global variable.
myValue = 10


# The show-value function prints
# the value of the global variable.
def showValue():
    print (myValue)


# Call the show-value function.
showValue()
```

Global Variables

- Important: If a function needs to assign a value to the global variable, the global variable must be re-declared within the function

  - General format: `global` *`variable_name`*

- **Important**: If a function needs to assign a value to the global variable, the global variable must be re-declared within the function

  - General format: `global` *`variable_name`*

```
# Create a global variable.
myValue = 10

def main():
    showValue()
    print (myValue)

def showValue():
    myValue = 12

# Call the main function.
main()
```

Program will just print out =10

Notice the myValue variable created in the function showValue is a local variable and not the global variable.

Changing a Global Variable Value

```
# Create a global variable.
myValue = 10


def main():
    showValue()
    print ('= ', myValue)



def showValue():
    global myValue
    myValue = 12


# Call the main function.
main()
```

Program will print out = 12

Change made here is made to the local variable myValue

Discussion

Content – Part 2

- Introduction to Value-returning Functions:
- Writing Your Own Value-Returning Functions
- The math Module
- Storing Functions in Modules

Introduction to Value-Returning Functions:

- **Simple function**: group of statements within a program for performing a specific task
  - Call function when you need to perform the task

- **Value-returning function**: similar to simple function, **returns a value**
  - A value-returning function is a function that returns a value back to the part of the program that called it.
  - The value that is returned from a function can be used like any other value: it can be assigned to a variable, displayed on the screen, used in a mathematical expression (if it is a number), and so on.

Standard Library Functions and the `import` Statement

The Python interpreter has a number of functions built into it that are always available. https://docs.python.org/3/library/functions.html

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Standard Library Functions and the `import` Statement

- Standard library: library of pre-written functions that comes with Python (https://docs.python.org/3/library/)
  - *Library functions* perform tasks that programmers commonly need
    - Viewed by programmers as a "black box"

A library function viewed as a black box

Modules

- The functions in the **standard library** are stored in files that are known as **modules**.
  - For example, functions for performing math operations are stored together in a module, functions for working with files are stored together in another module, and so on.

- In order to call a function that is stored in a module, you have to write an **import statement** at the top of your program.

- An import statement tells the interpreter the name of the module that contains the function.

- Format: ***import module_name***

Dot Notation

- If within our program we want to call a method from an imported module we can use dot notation.
- Dot notation: notation for calling a function belonging to a module
- Format: ***module_name.function_name()***

Using the random module

- `random` module: includes library functions for working with random numbers
- To use these functions we must first import the random module by including an import statement at the top of our class.
  - This statement causes the interpreter to load the contents of the random module into memory.
  - This makes all of the functions in the random module available to your program

```
import random
```

Using the random module

- The first function we will examine is named *randint*.

- `randint` function: generates a random number in the range provided by the arguments
  - Returns the random number to the part of program that called the function
  - Returned integer can be used anywhere that an integer would be used

```
number = random.randint(1, 100)
```

- Notice that two arguments appear inside the parentheses: 1 and 100.
- These arguments tell the function to return an integer random number in the range of 1 through 100.

Using the random module

- This method returns a value. We can assign this value to a variable (see below)

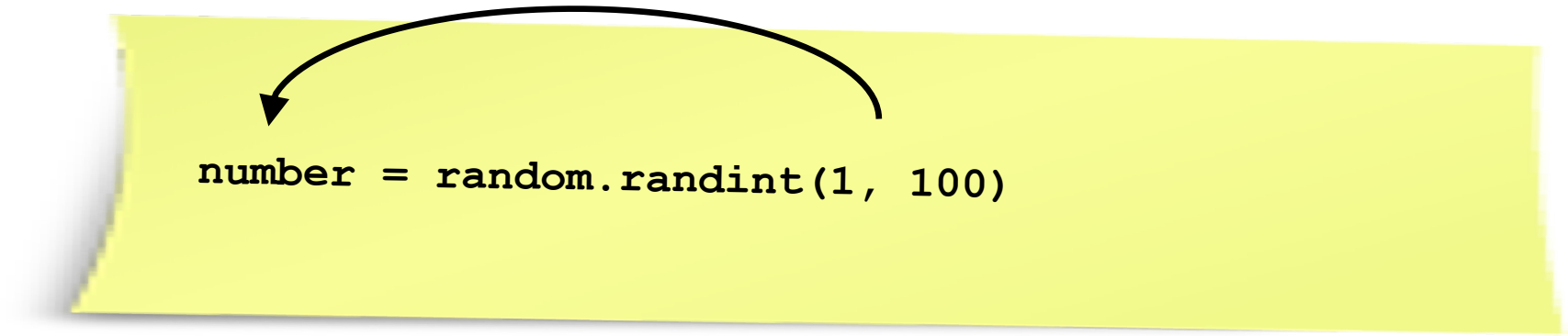
- The number that is returned will be assigned to the *number* variable

```
number = random.randint(1, 100)
```

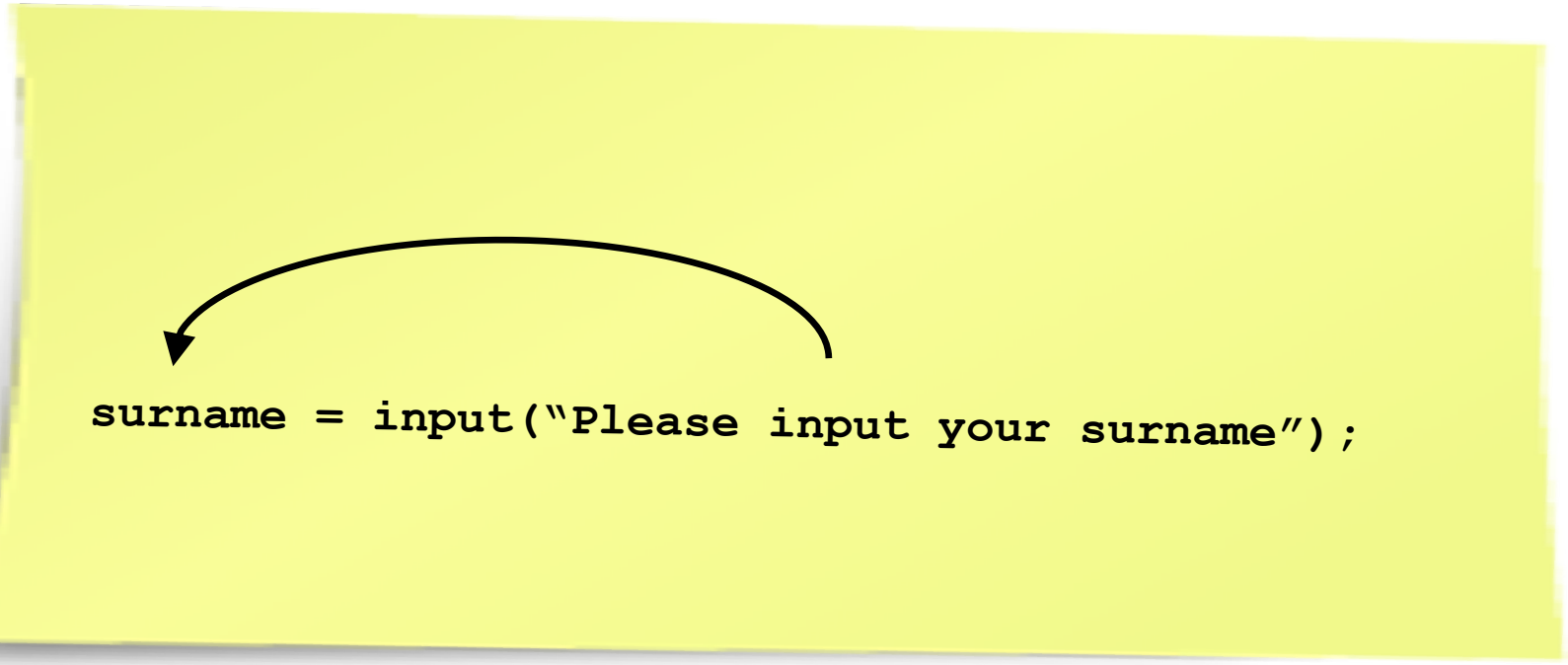
Using the random module

- This method returns a value. We can assign this value to a variable (see below)

- The number that is returned will be assigned to the *number* variable



Value Returning Function



The `math` Module

- `math` module: part of standard library that contains basic functions that are useful for performing mathematical calculations
  - Typically accept one or more values as arguments, perform mathematical operation, and return the result
  - Use of module requires an `import math` statement

The `math` Module (cont'd.)

Many of the functions in the `math` module

`math` Module Function	Description
`acos(x)`	Returns the arc cosine of `x`, in radians.
`asin(x)`	Returns the arc sine of `x`, in radians.
`atan(x)`	Returns the arc tangent of `x`, in radians.
`ceil(x)`	Returns the smallest integer that is greater than or equal to `x`.
`cos(x)`	Returns the cosine of `x` in radians.
`degrees(x)`	Assuming `x` is an angle in radians, the function returns the angle converted to degrees.
`exp(x)`	Returns e^x
`floor(x)`	Returns the largest integer that is less than or equal to `x`.
`hypot(x, y)`	Returns the length of a hypotenuse that extends from (0, 0) to (`x`, `y`).
`log(x)`	Returns the natural logarithm of `x`.
`log10(x)`	Returns the base-10 logarithm of `x`.
`radians(x)`	Assuming `x` is an angle in degrees, the function returns the angle converted to radians.
`sin(x)`	Returns the sine of `x` in radians.
`sqrt(x)`	Returns the square root of `x`.
`tan(x)`	Returns the tangent of `x` in radians.

Math Module Example

```
import math


def main():
    a = int(input('Enter the length of side A: '))
    b = int(input('Enter the length of side B: ' ))

    # Calculate the length of the hypotenuse.
    c = math.hypot(a, b)
    # Display the length of the hypotenuse.
    print ('The length of the hypotenuse is ' , c)



main()
```

The `math` Module (cont'd.)

- The `math` module defines constants `pi` and `e`, which are assigned the mathematical values for *pi* and *e*
  - Can be used in equations that require these values, to get more accurate results

- Variables must also be called using the dot notation
  - Example:

```
circle_area = math.pi * radius**2
```

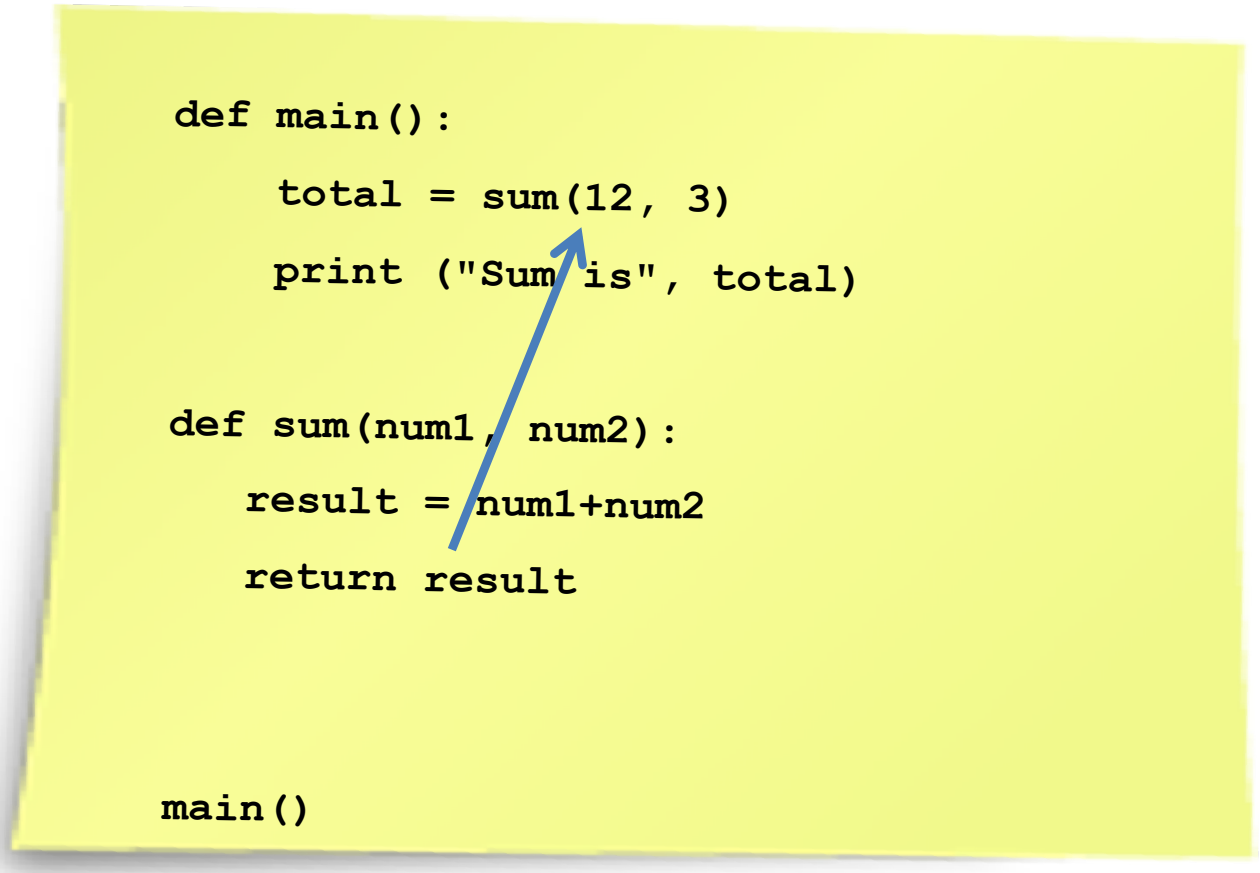
Writing Your Own Value-Returning Functions

- We will now look at writing our own value-returning functions in the same way that you write a simple function, with one exception: a value-returning function must have a return statement.
- Here is the general format of a value-returning function definition in Python:

```
def function-name ( ):
    statement
    statement
    etc.
    return expression
```

Writing Value-Returning Methods

- Following is a simple example of a sum method:



```
def main():
    total = sum(12, 3)
    print ("Sum is", total)

def sum(num1, num2):
    result = num1+num2
    return result

main()
```

Returning Boolean Values

- Boolean function: returns either `True` or `False`
  - Normally used to test a condition such as for decision and repetition structures
    - Common calculations, such as whether a number is even, can be easily repeated by calling a function

Boolean Example

```
def isEven(number):
    if (number % 2) == 0:
        status = True
    else:
        status = False
    return status


def main():
    number = 122
    answer = isEven(number)
    print ("Is even is ", answer)


main()
```

Boolean Example

```
def isEven(number):
    if (number % 2) == 0:
        return True
    else:
        return False


def main():
    number = 122
    answer = isEven(number)
    print ('Is even is ', answer)

main()
```

Notice isEven has multiple return statements but only one can ever be executed.
You should make sure that there is not path that allows the function to finish without returning a value.

Boolean Example

```
def isEven(number):
    return ( (number % 2) == 0 )


def main():
    number = 122
    answer = isEven(number)
    print ('Is even is ', answer)

main()
```

Notice in this variant the expression inside the brackets evaluates to either True or False and this value is returned.

Returning Multiple Values

- In Python, a function can return multiple values
  - Specified after the return statement separated by commas
    - Format: `return` *`expression1, expression2, etc.`*
  - When you call such a function in an assignment statement, you need a separate variable on the left side of the = operator to receive each returned value

```
def getName():
    first = input( 'Enter your first name:' )
    last = input( 'Enter your last name: ' )
    return first, last


def main():
    firstName, lastName = getName()
    print ("First Name: ", firstName, " Second Name: ", lastName)

main()
```

Storing Functions in Modules

- In large, complex programs, it is important to keep code organized
- **Modularization**: grouping related functions in modules
  - Makes program easier to understand, test, and maintain
  - Make it easier to reuse code for multiple different programs
- Module is a file that contains Python code
  - Contains function definition but does not contain calls to the functions
    - Importing programs will call the functions
- Rules for module names:
  - File name should end in `.py`
  - Cannot be the same as a Python keyword
- Import module using `import` statement in the normal way

Storing Functions

- Example: Module called rectangle.py

```
def area(width, length):
    return width * length

def perimeter(width, length):
    return 2 * (width + length)
```

Storing Functions

```
import rectangle



def main():
    sideA = 12;
    sideB = 10;
    recArea = rectangle.area(sideA, sideB)
    print (recArea)


main()
```

Programming Task A

- Write a method that will ask a user for their age and return the value.

- Write another method that takes in a single int parameter and returns true if the int parameter is greater than 18 or false if it is less than 18.

- Using the above methods write a program that will ask a user for their age and will inform the user if they are over 18 or not

```
def getAge():
    age = int(input("Please input your age"))
    return age

def validateAge(userAge):
    if(userAge>=18):
        return True
    else:
        return False


def main():
    age = getAge();
    isValidAge = validateAge(age)

    print ('User valid age = ', isValidAge)

main()
```

Programming Task B

- Part A
  - Write a program that will generate two random numbers between 1 and 100.
  - Write a function that will return the sum of the two numbers.
  - Write a function that will ask the user to guess the value and will return the guessed value.
  - The program should then print out a message indicating if the user has guessed correctly or not.

- Part B
  - Make this program iterative. Allow the user to continue until they chose to exit.

```
import random

def main():
    numberA = random.randint(1, 100)
    numberB = random.randint(1, 100)

    result = sum(numberA, numberB)
    guess = askUser(numberA, numberB)
    if (result == guess):
        print ("Correct")
    else:
        print ("Incorrect")


def sum(num1, num2):
    return num1+num2

def askUser(num1, num2):
    guess = int(input("What is the sum of the following numbers: "+str(num1)+" + "+str(num2)))
    return guess

main()
```

```
import random

def main():
    continueGame = 'y'

    while continueGame == 'y':
        numberA = random.randint(1, 100)
        numberB = random.randint(1, 100)

        result = sum(numberA, numberB)
        guess = askUser(numberA, numberB)
        if (result == guess):
            print ("Correct")
        else:
            print ("Incorrect")
        continueGame = input("Would you like to continue y/n")


def sum(num1, num2):
    return num1+num2

def askUser(num1, num2):
    guess = int(input("What is the sum of the following numbers: "+str(num1)+" + "+str(num2)))
    return guess
```

```
def checkNumber(number):
    if number < 50:
        return 1
    elif number < 100:
        return 2
    else:
        print ("Invalid Value")


def main():
    number = 120
    result = checkNumber(number)
    print ("Result", result)

main()
```

Notice the final else doesn't have a return statement associated with it.

Python will operate anyway and simply return None if the number is greater than or equal to 100

This is very poor practice. All paths in a value returning function should have a return keywords associated with them.

Discussion

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computer Science

http://www.cit.ie

Thank you

Haithem. afli@cit.ie

@AfliHaithem