MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University

COMP7004 - Systems Scripting

Lecture 12: Python Functions

Dr. Vincent Emeakaroha
vincent.emeakaroha@mtu.ie

Semester 2, 2025

www.mtu.ie

Python Function

- A function is like a mini-program within a program.
- We have encountered some built-in functions already
  - print()
  - input()
  - len()
- A major purpose of function is to group code that get executed multiple times.
  - Easy maintenance / fixing of bug
- We will focus on learning how to write our own functions based on an example.

helloFunc.py

```
def hello():
        print("Howdy!")
        print("Howdy!!!")
        print("Hello there.")
hello()
hello()
hello()
```

def Statement

def hello():

- Defines a function named “hello()”
  - Functions names can be chosen freely
  - Must adhere to naming convention
- Note the definition syntax

Function Body

```
print("Howdy!")
print("Howdy!!!")
print("Hello there.")
```

- Presents the body of the function
- Function body code is executed whenever function is invoked
- Note function definition is different from invocation
  - Defining function alone does not execute it

Function Invocation

hello()
hello()
hello()

- Indicates the calling of the function. It could be called multiples times.
- When a function is called, execution jumps to the start of the function body and executes the code there.

helloFunc.py: Execution

```
def hello():
    print("Howdy!")
    print("Howdy!!!")
    print("Hello there.")
hello()
hello()
hello()
```

Output:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

def Statement with Parameter

- Functions can be defined to accept arguments in the form of parameters.
  - Parameters are typed between the parentheses
- Parameters are like variables
  - Contains the value of argument when function is called.
- Example
  - helloFunc2.py

helloFunc2.py

```
def hello(name):
        print(”Welcome "+ name)

hello("Vincent")
hello("John")
```

- Note that the parameter name is only valid in function body and its forgotten after function return.
- If you try this, for example, after *hello(“John”)*
  - Print(“Value of name is “+ name)
  - Will produce error message

Function: return Statement

- Functions can evaluate to a single value
  - Known as return value.
- A return statement specifies the return value.
- Characteristics
  - The `return` keyword
  - The value or expression that the function should return
- An expression could be used as a return statement
  - Will return what the expression evaluates to.

return.py

```
import random

def getAnswer(answerNumber):
    if answerNumber == 1:
        return "It is certain"
    elif answerNumber == 2:
        return "It is decidedly so"
    elif answerNumber == 3:
        return "Yes"
    elif answerNumber == 4:
        return "Reply hazy try again"
    elif answerNumber == 5:
        return "Ask again later"
    elif answerNumber == 6:
        return "Concerntrate and ask again"
    elif answerNumber == 7:
        return "My reply is no"
    elif answerNumber == 8:
        return "Outlook not so good"
    elif answerNumber == 9:
        return "Very doubtful"

r = random.randint(1,9)
fortune = getAnswer(r)
print(fortune)
```

Passing return Value as Argument

- The return value of a function can be passed as argument to another function.
- Example:

```
r = random.randint(1,9)
fortune = getAnswer(r)
print(fortune)
```

- The above code can be shortened to a single equivalent line

```
print(getAnswer(random.randint(1, 9)))
```

The None Data Type

- In Python there is a special value called **`None`**
  - It represents the absence of a value
  - Similar to **`null`**, **`nil`** or **`undefined`** from other programming language
- Python adds **`return None`** to the end of every function definition without a return statement.
  - Also if you use a **`return`** statement without a value, **`None`** is returned.
- Example:

```
>>> spam = print("Hello!")
Hello!
>>> None == spam
True
>>>
```

Keyword Arguments

- Special type of argument identified by the keyword put before them in function calls.
  - Often used for optional parameters
- As an example, the `print()` function has the optional parameter `end` and `sep`
  - `end` specifies what should be printed at the end of arguments
  - `sep` specifies what should be printed between arguments
- Let us consider the default behaviour of the `print()` function:

```
print("Hello")      Hello
print(”World")      World
```

Keyword Arguments

- You can use the optional parameters to change the default behaviour of `print()` function.
- Example:

```
print("Hello”, end=“”)
print(”World")
```

HelloWorld

- Similarly we can change the separator (default = ‘ ‘) as follows:

```
print("cat",  "dog",  "goat",  sep=“;")
```

cat;dog;goat

Local and Global Scope

- Local scope
  - Variables that exist in local scope are called local variables
  - Parameters and variables that are assigned in a called function
- Global scope
  - Variables that are assigned outside all functions
  - Variables that exist in this scope are called global variables
- A scope is like a container
  - When a scope is destroyed all the variables inside it are forgotten
- There is only one global scope and it is created when a program starts
  - When a program terminates, this scope is destroyed
- A local scope is created whenever a function is called
  - All the variables assigned exist within this scope.
  - When the function returns, the scope is destroyed.

Importance of Scope

- Separation of space and variable validity
- Code in the global scope cannot use any local variables
  - However, a local scope can access global variables
- Code in a function's local scope cannot use variables in any other local scope
- You can use the same name for different variables if they are in different scopes
  - Example: There can be a `local` variable named `count` and a `global` variable named `count` as well.
- Improve code maintainability by forcing functions to interact with programs through only parameters and return value.

Local Variable and Global Scope

- Local variable cannot be used in a global scope
- Example:

```
def chicken():
    eggs = 31337

chicken()
print(eggs)
```

```
MP7044-Systems-Scripting/Examples/python/keywordArg.py", line 10, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

- The error is because “eggs” existed in the local scope when “chicken()” was called and when it returned the scope was destroyed.

Two Local Scopes

- A variable in a local scope cannot be used in another local scope
- Example:

```
def chicken():
    eggs = 313
    bacon()
    print(eggs)

def bacon():
    ham = 101
    eggs = 0

chicken()
```

Output:
313

- Local variables in one function is completely separate from local variable in another function.

Global Variable and Local Scope

- Global variables can be read from local scope
- Example:

```
def chicken():
    print(eggs)

eggs = 50
chicken()
print(eggs)
```

Output:
50
50

- Since there is no code that assigns **`eggs`** a value in **`chicken()`** function, Python considers it a reference to global variable **`eggs`**

Same Name Global and Local Variable

- Technically, it is perfectly legal to use the same name for global and local variable.
  - However, to simplify your life, avoid this practice.
- Example:
  - sameName.py

```
def chicken():
    eggs = "chicken local"
    print(eggs)

def bacon():
    eggs = "bacon local"
    print(eggs)
    chicken()
    print(eggs)

eggs = "global"
bacon()
print(eggs)
```

Output:

```
bacon local
chicken local
bacon local
global
```

- There are actually three different variables in that program but with the same name
  - Can be very confusing to keep track of which variable is being used at any moment.
  - Therefore, avoid such practice.

The Global Statement

- The global statement can be used to modify a global variable within a function scope.
- Specifying the `global` keyword before a variable in a function, tells Python not to create a local variable of same name but to modify the global one.
- Example:
  - sameName2.py

```
def chicken():
    global eggs
    eggs = "turkey eggs"

eggs = "global"
chicken()
print(eggs)
```

Output:
turkey eggs

Variable Scope Rule

- There are four rules to tell if a variable is in a local or global scope:
  1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
  2. If there is a global statement for that variable in a function, it is a global variable.
  3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
  4. But if the variable is not used in an assignment statement, it is a global variable.
- Example
  - sameName3.py

Variable Rule Example: sameName3.py

```
def chicken():
    global eggs
    eggs = "turkey eggs" # this is global

def bacon():
    eggs = "bacon" # this is local

def ham():
    print(eggs) # this is global

eggs = 42 # this is global
chicken()
print(eggs)
```

Output:
turkey eggs

- In **`chicken()`** function, **`eggs`** is the global variable because there is a **`global`** statement at the beginning of the function.
- In **`bacon()`** function, **`eggs`** is a local variable because there is an assignment statement.
- In **`ham()`** function, **`eggs`** is global variable because it is just being used and there is no assignment or **`global`** statement.

Functions as Black Boxes

- Often, all you need to know about a function are its input parameters and output value.
  - You do not necessarily have to border about the internal implementation unless you are very curious.
- When thinking of functions in this high-level way, it's common to say that you are treating the function as a black box.
- Writing functions without global variables is encouraged so that you do not have to worry about the function's code interacting with the rest of your program.
- Use function parameters to pass in values to the function.

Exception Handling

- At the moment, getting an error or exception in your Python program means the entire program will crash.
- You do not want this to happen to your real world program. Instead you would want your program to be able to:
  - detect problems / errors
  - handle them
  - continue to run.
- Example
  - zeroDivide.py

Zero Division Error: zeroDivide.py

```
def divide(divideBy):
    return 42 / divideBy

print(divide(2))
print(divide(12))
print(divide(0))
print(divide(3))
```

Output:
21.0
3.5

```
zeroDivide.py", line 3, in divide
    return 42 / divideBy
ZeroDivisionError: division by zero
```

Error Handling Statement

- Errors can be handled with `try` and `except` statements. The code that could potentially have an error is put in a `try` clause. The program execution moves to the start of the `except` clause if an error happens.
- The Syntax is as follows:

```
Try:
    <normal program code>
except <error code>:
    <error handling code>
```

Handling Error: zeroDivide2.py

```
def divide(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print("Error: Invalid argument.")

print(divide(2))
print(divide(12))
print(divide(0))
print(divide(3))
```

Output:

```
21.0
3.5
Error: Invalid argument.
None
14.0
```

Function Call Error Handling

- A try clause can also be specified when calling functions. Any errors that occur in the function will be caught.
- Example:
  - zeroDivide3.py

```
def divide(divideBy):
    return 42 / divideBy

try:
    print(divide(2))
    print(divide(12))
    print(divide(0))
    print(divide(3))

except ZeroDivisionError:
    print("Error: Invalid argument.")
```

Output:

```
21.0
3.5
Error: Invalid argument.
```

Guess Number Program

```
# This is a guess the number game
import random

secretNumber = random.randint(1, 20)
print("I am thinking of a number between 1 and 20.")

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print("Take a guess.")
    guess = int(input())

    if guess < secretNumber:
        print("Your guess is too low!")
    elif guess > secretNumber:
        print("Your guess is too high!")
    else:
        break # This condition is the correct guess!

if guess == secretNumber:
    print("Good job! You guessed my number in "+ str(guessesTaken) + " guesses!")
else:
    print("Nope. The number I was thinking of was " + str(secretNumber))
```

```
I am thinking of a number between 1 and 20.
Take a guess.
15
Your guess is too high!
Take a guess.
11
Your guess is too high!
Take a guess.
8
Your guess is too high!
Take a guess.
4
Your guess is too low!
Take a guess.
6
Good job! You guessed my number in 5guesses!
```

Summary of Functions

- Functions are primary way to compartmentalise your code into logical groups. Since the variables in a function exist in their own local scopes, the code in one function cannot directly affect the values of variables in other functions.
- Functions are great tools to help you organise your code. You can think of them as black boxes. They have inputs in the form of parameters and outputs in the form of return values.
- Programs can be made more resilient to common errors by using try and except statements, which can detect errors and control program execution.

MTU
Ollscoil Teicneolaíochta na Mumhan
Munster Technological University