University of
South Australia

Problem Solving and Programming

Week 10 – Functions *…continued…*

Python Books

Course Textbook

Gaddis, Tony. 2012, *Starting Out with Python*, 2nd edition, Pearson Education, Inc.

Free Electronic Books

There are a number of good free on-line Python books. I recommend that you look at most and see if there is one that you enjoy reading. I find that some books just put me to sleep, while others I enjoy reading. You may enjoy quite a different style of book to me, so just because I say I like a book does not mean it is the one that is best for you to read.

- The following three books start from scratch - they don't assume you have done any prior programming:
  - The free on-line book "**How to think like a Computer Scientist: Learning with Python**", by Peter Wentworth, Jeffrey Elkner, Allen B. Downey and Chris Meyers, provides a good introduction to programming and the Python language. I recommend that you look at this book.
  - There is an on-line book "**A Byte of Python**" that is quite reasonable. See the home page for the book, or you can go directly to the on-line version for Python 3, or download a PDF copy of the book. This book is used in a number of Python courses at different universities and is another I recommend you look at.
  - Another good on-line book is "**Learning to Program**" by Alan Gauld. You can download the whole book in easy to print PDF format, and this is another book that would be good for you to look at.
- If you have done some programming before, you may like to look at the following:
  - **The Python Tutorial** - this is part of Python's documentation and is updated with each release of Python. This is not strictly an e-Book, but is book-sized.
  - **Dive into Python 3**, by Mark Pilgrim is a good book for those with some programming experience. I recommend you have a look at it. You can download a PDF copy.

Problem Solving and Programming

- More on writing programs:
  - User-defined Functions… *continued…*

Functions

- Revision Example:

  - Write a function which will take three numbers as parameters, sum them, and return the result.

```
def functionName(parameters):
    function_body_suite
```

Functions

Solution:

Functions

Example – calling the function once it has been defined:

Functions

- Why use functions?

  - Program development more manageable

    The divide-and-conquer approach makes program development more manageable. Construct program from smaller pieces/modules.

  - Simpler code

    Typically simpler and easier to understand when code is broken down into functions. Several small functions are much easier to read than one long sequence of statements.

  - Software reusability

    Use existing functions as building-blocks to create new programs.

  - Avoid repeating code

    Reduce the duplication of code within a program. Write code to perform a task once and then reuse it each time you need to perform the task.

  - Better testing

    Testing and debugging becomes simpler when each task within a program is contained in its own function. Test each function in a program individually to determine whether it correctly performs its operation. Easier to isolate and fix errors.

Functions

Like most other languages, you may return only one value/object from a function in Python.

One difference is that in returning a container type (such as a list), it may seem as if you can actually return more than a single object.

The following function returns a list (one object).

```
def example_function():
    return [1, 2, 3]

result = example_function()
print(result)
```

Output:

```
[1, 2, 3]
```

Functions

When lists are returned from a function, they can be saved in a number of ways.

```
def example_function():
    return [1, 2, 3]
```

As a list:

```
result = example_function()
print(result)
```

Output:

```
[1, 2, 3]
```

As individual variables:

```
no1, no2, no3 = example_function()
print(no1, no2, no3)
```

Output:

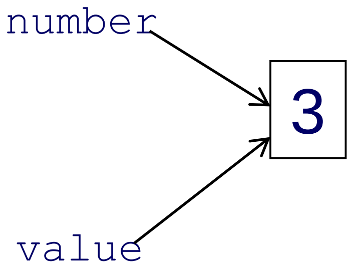
```
1, 2, 3
```

Each variable will receive its corresponding return value in the order the values are returned.

Passing Arguments to Functions

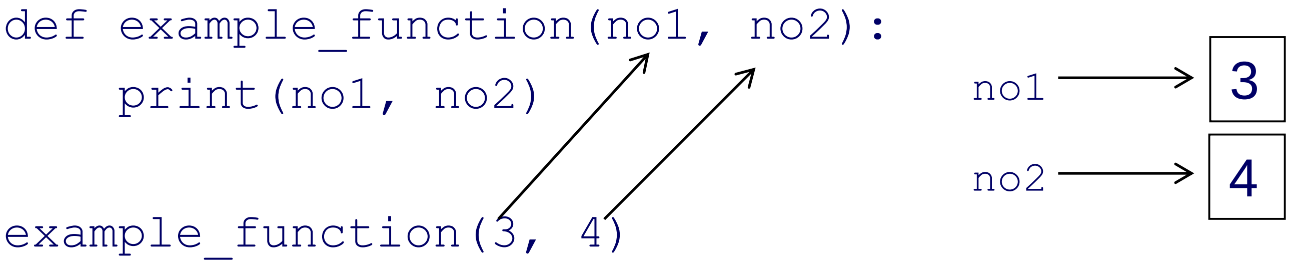
- A function is invoked by a call which specifies;
  - A function name,
  - May provide parameters/arguments.
- An argument is any piece of data that is passed into a function when the function is called.
  - A function can use its arguments in calculations or other operations.
  - When calling the function, the argument is placed in parentheses following the function name.
- A parameter is a variable that receives an argument that is passed into a function.
  - A variable that is assigned the value of an argument when the function is called.
  - The parameter and the argument reference the same value.

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

value = 3
example_function(value)
```



Passing Arguments to Functions

- As we have seen, a function may accept multiple arguments.
  - Parameter list items are separated by a comma.
  - Arguments are passed by position to the corresponding parameters.
    - First parameter receives the value of the first argument, second parameter receives the value of the second argument, etc.
    - Positional arguments:
      - Arguments must be passed in the exact order in which they are defined.
      - The exact number of arguments passed to a function call must be exactly the number defined.

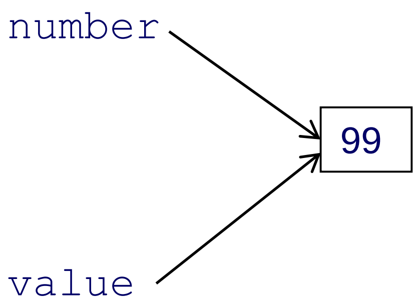


```
def example_function(no1, no2):
    print(no1, no2)

example_function(3, 4)
```

Passing Arguments to Functions

- Changes made to a parameter value within a function do not affect the argument.
  - The `value` variable passed to the `change_me` function cannot be changed by the function.
  - For example:

```
def change_me(number):
    print('Attempting to change the value!')
    number = 0
    print('Now the value is:', number)


value = 99
print('The value is:', value)
change_me(value)
print('Back from function call and the value is still:', value)
```



number

value

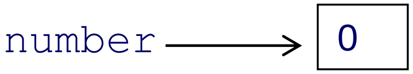
99

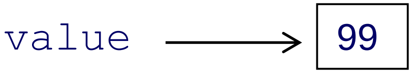
Passing Arguments to Functions

- The `value` variable passed to the `change_me` function cannot be changed by the function.

- For example:

```
def change_me(number):
    print('Attempting to change the value!')
    number = 0
    print('Now the value is:', number)
```





```
value = 99
print('The value is:', value)
change_me(value)
print('Back from function call and the value is still:', value)
```

Output:

```
The value is: 99
Attempting to change the value!
Now the value is: 0
Back from function call and the value is still: 99
```

Passing Arguments to Functions

- Functions can change the values of arguments if they are mutable non-primitive values (pass by reference).

```
def change_me(aList, aNum):
    aList.append(aNum)
    aNum = aNum + 1

numList = [1, 2]
num = 3
print(num, numList)
change_me(numList, num)
print(num, numList)
change_me(numList, num)
print(num, numList)
```

Output:

```
3 [1, 2]
3 [1, 2, 3]
3 [1, 2, 3, 3]
```

Passing Arguments to Functions

- Functions can change the values of arguments if they are mutable non-primitive values (pass by reference). Another example:

```
def change_me1(aList):
    for index in range(0,len(aList)):
        aList[index] = index * 2

def change_me2(aNum):
    aNum = 777


numList = [0, 1, 2, 3, 4]
num = 3

# List before call to function change_me1
print("List is (before change_me1): ", numList)

# Call function change_me1 to update the list
change_me1(numList)

# List after call to function change_me1
print("List is (after change_me1): ", numList)

# Num before call to function change_me2
print("Num is (before change_me2): ", num)

# Call function change_me2 to update num
change_me2(num)

# Num after call to function change_me2
print("Num is (after change_me2): ", num)
```

Output:

```
List is (before change_me1):  [0, 1, 2, 3, 4]
List is (after change_me1):  [0, 2, 4, 6, 8]
Num is (before change_me2):  3
Num is (after change_me2):  3
```

Passing Arguments to Functions

- Keyword arguments
  - Identify the arguments by parameter name in a function call.
  - Allows for arguments to be missing (related to functions with default arguments) or out-of-order.

  - For example:

```
def example_function(no1, no2):
    print(no1, no2)
```

  - Naturally, we can call the function giving the proper arguments in the correct positional order in which they were declared:

```
example_function(3, 4)
```

Output:

```
3 4
```

Passing Arguments to Functions

- Keyword arguments allow out-of-order parameters, but you must provide the name of the parameter as a ‘keyword’ to have your arguments match up to their corresponding argument names:

    - For example:

```
def example_function(no1, no2):
    print(no1, no2)



example_function(no2=3, no1=4)
```

Output:

```
4 3
```

Passing Arguments to Functions

- Default arguments
    - Arguments declared with default values.
    - Parameters are defined to have a default value if one is not provided in the function call for that argument.
    - Definitions are provided in the function declaration header.
    - Positional arguments must come before any default arguments.

```
def example_function(no1=7, no2=2):
    print(no1, no2)

example_function()
```

Output:

```
7 2
```

Passing Arguments to Functions

```
def f(a, b=3, c=4):
     print(a, b, c)


f(1)             Output:    1 3 4


f(1, 2)          Output:    1 2 4


f(1, c=5)        Output:    1 3 5


f(c=7, a=3)      Output:    3 3 7
```

Functions - Scope

- Scope of variables
  - The part of a program in which a variable may be accessed.
  - Variables either have local or global scope.
  - Variables defined within a function have local scope.
  - Variables defined at the highest level in a module/file have global scope.
  - Global variables have a lifespan that lasts as long as the program is executing and whose values are accessible to all functions.
  - Local variables live as long as the functions they are defined in are active and are accessible within the function only.
  - Example:

```
def example_function():
    local_str = 'local'
    print(global_str + ' ' + local_str)

global_str = 'global'

example_function()
```

Output:

```
global local
```

Functions - Scope

- Variables can be defined:
  - At the top level in a program.
  - In a block.
- If you define a variable in a block:
  - It is distinct to any variable with the same name defined outside the block.
  - When you leave the block, you no longer have access to the variable.
- Parameters to functions are treated as variables defined in the block of the function, initialised using the parameter passed when the function is called.
- When searching for an identifier/variable, Python searches the local scope first. If the name is not found within the local scope, then an identifier must be found in the global scope (or an error occurs).
- If possible, you should avoid the use of global variables.
  - Typically makes programs difficult to maintain.
  - Only use the information passed to functions via the parameters or defined within the function.

Functions - Scope

- Local Variables
  - A variable that is assigned a value inside a function.
    - When you assign a value to a variable inside a function, you create a local variable.
    - Belongs to the function in which it was created.
    - Only statements inside that function can access it, an error will occur if another function tries to access the variable.

      Example:

```
def example_function():
    local_str = 'local'

example_function()
print(local_str)   # This cases an error!
                   # NameError: name 'local_str' is not defined
```

  - Different functions may have local variables with the same name.
    - Each function does not see the other function’s local variables, so no confusion.

Functions - Scope

- Global Variables
  - A global variable is accessible to all the functions in a program file.
  - When a variable is created by an assignment statement that is written outside all the functions in a program file, the variable is global.
  - A global variable can be accessed by any statement in the program file, including the statements in any function.

  - For example:

```
def example_function():
    print('String in function is:', global_str)


global_str = 'global'

example_function()
```

Functions - Scope

- global statement
  - Global variable names can be overridden by local variables.
  - For example:

```
def example_function():
    global_str = 'local'
    print('String in function is:', global_str)



global_str = 'global'

print('String is:', global_str)
example_function()
print('String is:', global_str)
```

Output:

```
String is: global
String in function is: local
String is: global
```

Functions - Scope

- global statement
  - An additional step is required if we want a statement in a function to assign a value to a global variable. You must declare the global variable.
  - We modify the code so that the global version of global_str is used.
  - To reference/access a global variable, use the global statement.
  - For example:

```
def example_function():
    global global_str
    global_str = 'local'
    print('String in function is:', global_str)


global_str = 'global'

print('String is:', global_str)
example_function()
print('String is:', global_str)
```

Output:

```
String is: global
String in function is: local
String is: local
```

Functions - Scope

- Global Variables and Global Constants
  - Most programmers agree that you should restrict the use of global variables, or not use them at all. A few reasons why:
    - Make debugging difficult.
      - Many locations in the code could be causing a wrong variable value.
    - Functions that use global variables are usually dependent on those variables.
      - Makes functions harder to transfer to another program.
    - Make a program hard to understand.
  - **In most cases, you should create variables locally and pass them as arguments to the functions that need access to them.**
  - **Only use the information passed to functions via the parameters or defined within the function.**

  - Although you should try to avoid the use of global variables, it is permissible to use global constants in a program.
  - A global constant is a global name that refers to a value that cannot be changed.
  - Although Python does not allow you to create true global constants, you can simulate them with global variables.
  - For example:

```
MAX_NUMBER = 10
```

  - It is common practice to write a constant's name in uppercase letters.
    - Reminder that the value referenced by the name is not to be changed in the program.

Functions - Scope

- Scope exercise:
  - What output does the following code produce?

```
j = 1
k = 2

def function1():
    j = 3
    k = 4
    print('j is:', j, 'k is', k)

def function2():
    j = 6
    function1()
    print('j is:', j, 'k is', k)

k = 7
function1()
print('j is:', j, 'k is', k)

j = 8
function2()
print('j is:', j, 'k is', k)
```

End of Week 10