Lab 3: Functions

Learning Objectives

- To learn how to develop **user-defined functions** in Python;
- To understand key concepts in Python functions: **parameters/arguments**, **docstrings**, **returned value**, **global variable**, **local variable**;
- To understand how Python function affects the input arguments under different scenarios;
- To practice more on structured programming: **if … else…** and **while**/**for** loops;
- To practice more on tuple, list, and dictionary;
- To understand the concept of **modules** in Python.

Functions

A function is a unit of code that works with various inputs, and is able to produce concrete result(s) based on the input(s). Python offers many useful **built-in functions** such as **input()** and **print()**. A function is reusable: once you define a function, you can use it multiple times. The following is an example that defines a function **sum()** which takes two input parameters and another function **hello()** which has no input. A defined function will not be executed until a **caller** calls (runs) it.

```
# define a function
def sum(a, b):
    return a + b

#call the sum() function
x = sum(100, 200)
y = sum(2.3, 5.7)
z = 2 * sum(5, 10)  # use function in an expression
s = sum('abc', 'xyz')

# define a function without parameter
def hello():
    print('Hello, COMP1007 Students!')

#call the hello() function
hello()
```

In this example, **sum()** is a **user-defined function** which accepts two input variables (called **parameters**) and returns the summation of the two parameters. **hello()** is another user-defined function which simply prints a hello message.

When should we define a function? (1) If a software program has a routine task that will be repeatedly executed, we may develop a function for this task so that every time we can do the task by simply calling the function. (2) If a software program is very complex, we may decompose it into multiple smaller tasks, and then develop a function for each small task.

Defining a Function

In Python, the user needs to follow the below syntax to define a function:

- Begin with the ***def*** keyword;
- Followed by the function ***name***, which is the identifier of the function and has the same rule as variable names;
- Then place the list of ***parameters*** between ***parentheses*** and followed by a ***colon***. Sometimes a function has no parameter at all;
- The function ***body***: the list of statements to be executed when the function is executed. Indentation is mandatory for the function body.
- Usually the function ***returns*** a result to the caller. If you don't put an explicit **return statement**, the function will automatically return ***None***, which is a keyword in Python. Return statement can appear anywhere in the function body. Once the return statement is executed, it ends the function call.

```
def function_name(param1, param2, … paramN):
    statement 1
    statement 2
    ……
    statement n
    return return_value
```

Indentation

Function Parameters

A function may take one or more input values, which are called **parameters** (or **arguments** when the function is called). The parameters are optional, depending on what the function needs to do. The function name and its list of parameters are called the **signature** of the function.

When we define a function in Python, we need not specify the data type of parameters. However, when you call the function, the parameters should match the body statements of the function. For example:

```
def sum(a, b):
    return a + b # we expect that "a + b" is a legal statement

print( sum(2, 3) ) # 5
print( sum('py', 'thon') ) #python
```

Trial-and-Error Exercise:

- What will happen if you call function `sum(2, 'python')`?

Docstring

The following shows a more complicated example: to display the **Fibonacci series** – 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … until the input argument is reached.

```
def fib(n):
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end = ' ')
        a, b = b, a + b
    print()

fib(1000)
fib(10000)
fib(100000)
```

In this example, the first statement of the function body is a string literal with **triple quotes**; this string literal is the function's documentation string, or **docstring**. It usually explains what this function is about. Providing a docstring for your function is a good **programming style**.

The statement **a, b = 0, 1** is the same as two separate statements **a = 0** followed by **b = 1**. In fact, this multiple assignment statement simply assigns the tuple (0, 1) to the tuple (a, b). For the case of **a, b = b, a + b**, it has the same effect as the following statements:

```
Temp1 = b
Temp2 = a + b
a = Temp1
b = Temp2
```

Question: can the following statements replace **a, b = b, a + b** in the above example?

```
a = b
b = a + b
```

Optional parameters: Python allows you to specify **default values** for a function's parameters.

```
def circle(r, pi = 3.14):
    return pi * r ** 2
```

When we call function **circle()**, we should provide one argument or two arguments. If we provide one argument, its value will be passed to parameter r, and the value of pi will be equal to the default value 3.14. If you need higher precision, you can pass a more accurate value such as 3.14159 to parameter pi.

```
In [1]: circle(9)
```

```
Out[1]: 254.34

In [2]: circle(9, pi = 3.14159)
Out[2]: 254.46878999999998
```

Returned Value

In Python, a function is defaulted to return ***None***. You can omit the return statement if your function has nothing to return.

```
def greet(name):
    print('Hello ', name)
```

Python also allows a function to return more than one value by a single return statement. The multiple returned values will be stored as a tuple. E.g., using the built-in functions **max()** and **min()**:

```
def find_max_min(num_list):
    return max(num_list), min(num_list)

alist = [1,2,3,4,5,6,7,8,9]
m, n = find_max_min(alist)
print(m, n)
```

Python function can also return a list. The following function **reduce_list()** accepts a list as the parameter, then creates a new list that contains the unique elements of the input list and returns it.

```
#%%
def reduce_list(long_list):
    short_list = []
    for i in long_list:
        if not (i in short_list):
            short_list.append(i)
    return short_list

short = reduce_list(['abc', 'jack', '123', 'jack', 'abc', '123'])
print(short)
```

In the above example, the expression "**i in long_list**" is *True* or *False*, depending on whether item **i** can be found in **long_list** or not. Please pay attention to the nested indentations.

The next example shows a function **seq_to_dic()** that converts an indexed sequence (tuple or list) into a dictionary, using the index as the key for each element. It first uses the built-in function **dict()** to create an empty dictionary.

```python
#%%
def seq_to_dic(seq):
    myDic = dict()  # create an empty dictionary
    for i in range(len(seq)):
        myDic[i] = seq[i] # add a new key-value pair i:seq[i] into myDic
    return myDic

li = ['a', 'b', 'c', 'd']
dic1 = seq_to_dic(li)
print(dic1)

tu = ('Ada', 'Bob', 'Catty', 'David')
dict2 = seq_to_dic(tu)
print(dict2)
```

Exercise 1

Write a function that accepts three numbers and returns the minimum one. You are not allowed to use the build-in **`min()`** function. Instead, you can use **if … else** statement.

Exercise 2

Write a function that accepts a Celsius degree and returns the corresponding Fahrenheit degree.

$$Fahernheit = 1.8Celsius + 32$$

Global Variable and Local Variable

Global variable means a variable defined outside a function. Global variables can be accessed inside or outside functions. Variables defined inside a function (i.e., the variable name appears on the left side of an assignment operator) is called the **local variables** of that function. Local variables of a function can only be accessed (i.e., read or write) within that function.

```python
#%%
global_var = 100
test_var = 10

def my_func():
    test_var = 20 # Question: is this test_var local or global?
    print("inside function, global_var = ", global_var)
    print("inside function, test_var = ", test_var)

my_func()
print("outside function, test_var = ", test_var)
```

Remark: In the above example, when function my_func() is called, it assigns 20 to its local variable test_var, which will NOT change the value of the global variable test_var.

If a function wants to change the value of a global variable, it needs to make a declaration before accessing the global variable. E.g.,

```
#%%
global_var = 100
test_var = 10

def my_func():
    global test_var
    test_var = 20
    print("inside function, global_var = ", global_var)
    print("inside function, test_var = ", test_var)

my_func()
print("outside function, test_var = ", test_var)
```

In the above example, the statement **global test_var** inside the body of function my_func() tells the function that the identifier **test_var** refers to the global variable test_var.

Calling Function

While calling the function, the user needs to match all the parameters. Otherwise, Python will pop error to the user. For example, if the function needs to take one parameter, the user must insert exactly one parameter.

```
#%%
def print_list_onebyone(list1):
    for x in list1:
        print(x)
```

In this example, `print_list_onebyone()` needs one ***iterable object*** as the parameter. The user must insert exactly one such iterable object (such as list, tuple, or range).

```
In[1]: print_list_onebyone([1,2,3,4])
1
2
3
4
```

If the user inserts more than one parameter or no parameter, python will pop an error.

```
In[1]: print_list_onebyone([1,2] , [3,4])
Traceback (most recent call last):

File "<ipython-input-7-fb3266eeb63b>", line 1, in <module>
        print_list_onebyone([1,2] , [3,4])

TypeError: print_list_onebyone() takes 1 positional argument but 2 were
given
```

Case Study: Can the input arguments be modified by a function?

The following is a declaration of a function. The function accepts a string s. In the function, s is updated by the **upper()** function (change all characters to upper case).

```
def  change(s):
   s = s.upper()
```

The following function accepts a number n and increases n by 10.

```
def increase(n):
   n = n + 10
```

The following function accepts a list c and append a new element 3.5 to c.

```
def add(c):
   c.append(3.5)
```

The functions above intend to update the input arguments directly. Let's see what will happen if we call them as follows:

```
text = "hello world"
number = 10.4
mylist = [10, 2.4, 22, 1]

change(text)
increase(number)
add(mylist)

print(text)
print(number)
print(mylist)
```

We get the result that the **change()** and **increase()** functions are not able to change the input arguments (i.e., **text** and **number**). But, the **add()** function is able to update the input list **mylist**.

In Python, if the arguments are **immutable** like integers, floats, strings or tuples, the changes are only applicable inside the functions. Outside the functions, their values remain no change. But, if the arguments are **mutable** like lists or dictionaries, **their elements can be changed** in place in the function.

Exercise 3

Write a function that returns a ***tuple*** that contains all odd numbers in the range from 1 to X, where X is a function parameter. If X is less than 1, display an error message。

Exercise 4

Write a function that accepts a string, reverses the string and returns the reversed string. Please use a for loop to implement the reverse function.

Exercise 5

Write a function that accepts two sequences (such as tuple, list, string) as input, and returns a list which is the intersection of the two input lists.

Python Modules

A large software program may consist of many Python script files. It may also use some Python standard library functions and some third-party functions. In order to avoid name clashes (i.e., different Python files use the same identifiers), Python uses **modules** to organize the program codes for reuse, and provides self-contained **namespaces** that minimize variable name clashes across your programs. **Each Python file is a module**, and modules **import** other modules to use the names (such as functions and global variables) they define.

There are two popular ways to use a module:

import: Lets a client (importer) fetch a module as a whole

from: Allows clients to fetch particular names from a module

For instance, suppose a file ***my_module.py*** defines a function called **hello()**, for external use. Suppose ***client.py*** wants to use function hello() defined in ***my_module.py***. To this end, it may contain Python statements such as the following:

```
#file my_module.py
def hello(x):
    print('Hello ' + x)
```

```
#file client.py
import my_module # to access file my_module.py
my_module.hello('Python') # to use the hello() function
```

The import statement in client.py loads the target file ***my_module.py*** to execute. For user programs such as my_module.py, it should be usually stored in the same directory as the client.py so that the Python environment can find it. The follow example shows how to use the **from** statement.

```
#file client.py
from my_module import hello # to access file my_module.py
hello('Python') # to use the hello() function
```

Learning Python includes two stages: (1) to learn the Python language syntax; (2) to learn how to use different existing modules. We will learn many useful Python modules later.