Session 3: Functions

Prepared by: Harith Zulfaizal

March 2020

1 Introduction

An efficient code is one that maximizes reusability. Code reuse is a great way to reduce error and helps in making debugging easier. Function is one of the greatest way of code reuse where when you are performing the same set of codes for an operation for a number of times, it is best to encapsulate it in a function. This function can be called anytime inside the code to perform the aforementioned encapsulated operations, thus introducing modularity in your code.

The actions performed by the functions depends on the definition given by the user and what values passed as the arguments inside the functions. The function may or may not return values as their output. To define a function, the keyword `def` is used followed by the function name and parenthesis. The function name follows the same rules as variables, ie. they can contain alphabets, numbers and underscores. The statement `return` exits a function, optionally passing back the output to the caller. To call a function, simply write the function name followed by parenthesis.

```
def function_name():
    print("Not nice")

def doNothing():
    pass

def func():
    return 96

function_name() #calling the function
print(function_name())

x=func() #calling the function by reassigning to a variable
print(x)
```

1.1 Arguments

So far we have been dealing functions with no arguments, or what some other users would call parameters. Arguments may be represented as variables separated by commas that may be used in the function body and parameterized the function. Functions can have as many arguments as one desires, separated by commas. Python handles arguments usually in a positional manner, hence the order of the arguments passed is important when calling the functions.

```
def test(x,y,z):
    return x**2+y**2+z**2

def power(base,x) #this is a reimplementation of an already existing global function
    return base**x

r=test(2,3,3)
print(r)
print(test(2,3,3))

m=power(5,4)
print(m)
```

It is also possible to return more than one value. If you do not determine what type the function returns, it will return the values as a tuple. The output may differ on how you call them.

```
def two(a,b):
    x=a**2
    y=b**2
    return x, y

s=two(2,3)
d, f=two(2,3)
print(s)
print(d)
print(f)
```

The arguments need not be just numbers, strings can also be arguments.

```
def name(first, last):
    print("My name is " + first + " " + last)

name("Harith", "Akmal")
```

1.1.1 Namespaces

Now, when you have functions, the Python program will have a namespace specific to each function and to the main body. So, a variable defined in one namespace would not be available in another unless it is passed as an argument or the other namespace is local to the one the variable is residing (global variables).

In a Python script, you are automatically in the global namespace. Therefore, any variables declared are global variables. These variables are obtainable in functions you declare later as well. If you try to change variables inside a function, that variable will be specific only in that function's namespace. To change it within a function, you need to access the global variable by writing `global`.

```
topic = "quantum chromodynamics"

def changeLocal():
    topic = "chaos theory"
    print(topic, id(topic))

def changeGlobal():
    global topic
    topic = "general relativity"
    print(topic, id(topic))

changeLocal()
print(topic, id(topic))
changeGlobal()
print(topic, id(topic))
```

If you want to access what variables are available in a specific namespace, you use the locals() method for variables in the local namespace and globals() for global variables. It will return a dictionary.

```
topic = "quantum chromodynamics"

def changeLocal():
    topic = "chaos theory"
    print("locals:", locals())


changeLocal()
print("globals:", globals())
```

1.1.2 Keyword Arguments

Arguments can be defined by a value which can act as a standard behavior. This is called as keyword arguments where the order of the argument doesn't matter and not every arguments have to be passed by the user. The syntax of a function with default keyword arguments is as follows:

```
def func(<arg1>, <arg2>, ...., <kwarg1>=<val1>, <kwarg2>=<val2>, ...):
    <body>
```

As an example, try out this polynomial equation function with keyword arguments:

```
def polyn(x, a=1.0, b=3.0):
    print(a*x+b)
```

The *polyn*() function can be called with neither a or b, either a or b, or both. The position of the arguments a an b does not matter as long as we define the a and b in the call statement. The argument x is a positional argument with no value, thus x must be in the designated order and be passed through the function. Here are a few variations of how the *polyn*() function works when called, check them out to understand how it works:

```
polyn(3)
polyn(3,2)
polyn(3,2,5)
polyn(3, a=2, b=4)
polyn(3, b=4, a=2)
```

We can also use `None` as an argument in a function, when there is nothing to be passed but you still want to use the function.

```
def FuncName(string, var=None):
    if var is None:
        print(string)
    else:
        print(string+var)

FuncName("Hi!")
FuncName("Hi!", "My name is Harith!")
```

Mutable data tyes such as lists, sets and dictionaries should be avoided to be passed as an argument in a function since they retain their state from one call to another. It can be done, but one needs to be aware of how to manipulate them according to their need.

1.2 Variable Number of Arguments

The arguments for a function can have arbitrary number. This is especially useful if you don't exactly know how many arguments you are dealing with, like in the case of dealing with lists.

To write a function that takes a variable number of arguments, a special character asterisk (*) must be added before the name of the argument. The variable argument needs also be written later in the case of multiple arguments. The syntax is as follows:

```
def func(<arg1>,<arg2>,...,<kwarg1>=<val1>,...,*<args>):
    <body>
```

What it does is that the args variable is a tuple with extra arguments packed inside. For example, we can reconstruct a minimum function that checks the minimum value over a set of data.

```
def printArgs(*args):
    print("Positional argument tuple:", args

def printSomeStuff(required1, required2, *args):
    print(required1)
    print(required2)
    print(*args

def minimum(*args):
    m=args[0]
    for x in args[1:]:
        if x < m:
            m=x
    return m

num=[3,52,12,5,32,9,2]
print(minimum(*num))
```

A variable number of unknown keyword arguments can also be done by adding two asterisk (*) before the name of the argument. The keyword arguments will be packed into a dictionary with string keys. The syntax of such a functions is:

```
def func(<arg1>,<arg2>,...,<kwarg1>=<val1>,...,*<args>,*<kwargs>):
    <body>
```

Here are some example that shows args and kwargs are tuple and dictionaries:

```
def blender(*args, **kwargs):
    print(args, kwargs)

blender("no", 96)
blender(z=3, y=96)
blender("no", [1], "yes", z=6, x=24)

t=("no",)
d={"sick": "wack"}
blender("yes", k="rad", *t, **d)
```

NB: Never forget to document your code whenever possible! Sometimes you made so many and the script becomes so sophisticated that you forget the purpose of some stuff you wrote before. Make it a habit/

1.3 Exercises

Construct a quadratic formula function to find the root solution(s). Make it able to identify the type of roots by using the discriminant $b^2 - 4ac$.(Hint: Use if-elif statements inside the function)

2 Recursions

In Python, a defined function is able to call itself from within its own body. This is known as recursion. The classic example of a recursion function is the Fibonacci sequence, as learned last week:

```
def fib(n):
    if n==0 or n==1:
        return n
    else:
        return fib(n-1) + fib(n-2)

print(fib(10))
```

From the Fibonacci sequence, we know that it starts with two initial values of 0 and 1. In the Fibonacci function, if n is neither 0 nor 1, the function will return fib(n-1)+fib(n-2) which will call the function itself. In this recursion sequence, the only needed values are 0 and 1, any other values of n will be traced back through recursion to fib(0)=0 and fib(1)=1.

2.1 Exercises

Get the factorial for n=5, 10, 32 using the recursion function.

$$n! = n(n-1)(n-2)...(2)(1) \tag{1}$$

3 Lambda Function

Lambdas or anonymous functions are special ways to create single-line functions and got its name from the way they are defined such that they are considered as expressions rather than statements. A lambda function can take any number of arguments but only one expression. Lambda function is defined by using the lambda keyword followed by the arguments, a colon (:) and an expression. The format of the lambda is:

```
lambda <args> : <exprs>
```

The expression can be as simple or complex as you want as long as it is a single expression. A lambda function can also be assigned to a variable and be called like a normal function. As previously mentioned, the lambda function can take any number of arguments. Examples:

```
x = lambda a : a + 10
print(x(2))

y= lambda a, b, c: a+b+c
print(y(1,2,3))

def func(n):
  return lambda a : a * n

double = func(2)

print(double(11))
```

Now, it seems like using `Lambda` is a just like any other functions but minus the name right. You might be wondering, when you would need to use them? Any time you need a direct function on the spot. A good example would be the sort() method for lists. For example:

```
x =[-5, 4, 6, -7, 9, -8, 10, -1, -2]

#
# Now for whatever reason, you want to order the list above
# based on the individual member's squared value instead.
#

x.sort(key= lambda x: x**2)
print(x)
```