3 Days Training on Python3

Day 2 : Module 5

Muhammad Saufy Rohmad

Module 2 (90 minutes)

Objectives

1. Introduction to Functional Programming
2. Higher Order Functions
3. Curried Functions

1. Introduction to Functional Programming

- Functional Programming is not a new idea and indeed goes right back to the 1950s and the programming language LISP
- Here we introduces Functional Programming (also known as FP) and the key concept of Referential Transparency (or RT).
- FP is : ... a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids state and mutable data.
- The first is that it is focussed on the computational side of computer programming

1. Introduction to Functional Programming(2)

- That is these functions only rely on their inputs to generate a new output.
- As an example of a side effect, if a function stored a running total in a global variable and another function used that total to perform some calculation; then the first function has a side effect of modifying a global variable and the second relies on some global state for its result.

1. Introduction to Functional Programming(3)

- Functional Programming aims to avoid side effects.
- Functional Programming avoids concepts such as state
- Functional Programming promotes immutable data
- Functional Programming promotes declarative programming

1.1 Referential Transparency

- For example, let us assume that we have defined the function increment as shown below.

  def increment(num):

  return num + 1

- If we use this simple example in an application to increment the value 5:

  print(increment(5))

  print(increment(5))

- We can say that the function is Referentially Transparent (or RT) if it always returns the same result for the same value (i.e. that increment(5) always returns 6).

1.1 Referential Transparency(2)

- the following code is no longer Referentially Transparent:

```
amount = 1
def increment(num):
return num + amount
print(increment(5))
amount = 2
print(increment(5))
```

- The output from this code is 6 and 7—as the value of amount has changed between calls to the increment() function.

1.1 Referential Transparency(3)

- A closely related idea is that of No Side Effects.
- That is, a function should not have any side effects, it should base its operation purely on the values it receives,and its only impact should be the result returned.
- Any hidden side effects again make software harder to maintain.

2. Higher Order Functions

- These are functions that take as a parameter, or return (or both), a function.
- Function as objects
  - We can get the address of the function without ( ) when call the function.
  - Function is same as data that stored in memory.
- If we run:

  print(type(some_function))

  will produce:

  <class 'function'>
- This some_function is really a type of variables that reference at the function objects in memory which we can execute using the round brackets.

2. Higher Order Functions(2)

- We can pass the reference to the function to another variables.

  another_reference = get_msg

  print(another_reference())

- We did not make a copy of the function, only its address in memory.

2. Higher Order Functions(3)

- A function that takes another function as a parameter is known as a higher order function.
- It take one or more functions as a parameter
- Return as a result a function
- All other function in python are first order functions.
- Many of the functions found in the Python libraries are higher order functions. It is a common enough pattern that once you are aware of it you will recognise it in many different libraries.

2. Higher Order Functions-Example

```
def apply(x, function):
result = function(x)
return result
```

- The function apply is a higher order function because its behaviour (and its result) will depend on the behaviour defined by another function—the one passed into it. We could also define a function that multiplies a number by 10.0, for example:

```
def mult(y):
return y * 10.0
```

- Now we can use the function mult with the function apply, for example:

```
apply(5, mult)
```

- This would return the value 50.0

2. Higher Order Functions- Example(2)

```
def mult_by_five(num):
    return num * 5
def square(num):
    return num * num
def add_one(num):
    return num + 1
```

- All of the above could be used with the following higher order function:

```
def apply(num, func):
return func(num)
```

- For example:

```
result = apply(10, mult_by_two)
print(result)
```

- This would output the value 20.0

2. Higher Order Functions-Example(3)

- The following listing provides a complete set of the earlier sample functions and how they may be used with the apply function:

  print(apply(10,mult_by_five))

  print(apply(10,square))

  print(apply(10,add_one))

  print(apply(10,mult_by_two))

- The output from this is:

  50

  100

  11

  20

2. Function returning function

```
def make_function():
    def adder(x, y):
        return x + y
    return adder
```

- We can then use this make_function to create the adder function and store it into another variable. We can now use this function in our code, for example:

```
f1 = make_function()
print(f1(3, 2))
print(f1(3, 3))
print(f1(3, 1))
```

- Which produce the output

```
5
6
4
```

3. Curried Functions

- Currying is a technique which allows new functions to be created from existing functions by binding one or more parameters to a specific value.
- It is a major source of reuse of functions in Python which means that functionality can be written once, in one place and then reused in multiple other situations.

3. Currying Concepts

- At an abstract level, consider having a function that takes two parameters.
- These two parameters, x and y are used within the function body with the multiply operator in the form x * y. For example, we might have:

  operation(x, y): return x * y

- This function operation() might then be used as follows:

  total = operation(2, 5)

- Which would result in 5 being multiplied by 2 to give 10. Or it could be used:

  total = operation(10, 5)

3. Currying Concepts(2)

- Which would result in 5 being multiplied by 10 to give 50.
- If we needed to double a number, we could thus reuse the operation() function many times, for example:

  operation(2,5)

  operation(2,10)

  operation(2,6)

  operation(2,151)

- All of the above would double the second number. However, we have had to remember to provide the 2 so that the number can be doubled
- However, the number 2 has not changed between any of the invocations of the operation() function.
- What if we fixed the first parameter to always be 2, this would mean that we could create a new function that apparently only takes one parameter (the number to double)

3. Currying Concepts(3)

- For example, let us say we could write something like:

  double = operation(2, *)

- Such that we could now write:

  double(5)

  double(151)

- In essence double() is an alias for operation(), but an alias that provides the value 2 for the first parameter and leaves the second parameter to be filled in by the future invocation of the double function.