Introduction to Python

Sana Rasheed

AL NAFI,
A company with a focus on education, wellbeing and renewable energy.

Section Three

Overview

- Testing
- Python Debugger
- Decorators
- Lambda
- Code Profiling

Introduction to Testing

- Once you write a code, it is never the final version. There is always room for improvement.
- Hence, you are always looking into refactoring code, making it more efficient, faster, cleaner or Pythonic.
- Although, sometimes, in an effort to improve the code, we drift away from the actual intent of the code in the first place.
- Therefore, there exists a way to ensure your code does what it was always intend to do, while you go on to improve it.
- This is known as testing or unit testing.

Introduction to Testing

- Unit testing is a practice for every programming language, where you create test cases to check whether your code behaves always in the way it is supposed to.
- These tests ensure the integrity of the code and reduce chances of error on part of the coder.
- In Python, you can perform tests using a library called unittest. It provides an entire framework to built unit test for your code.

Introduction to Testing

- Suppose, you have a piece of code that takes argument from the user their first and last name and returns each capitalized and joined as a single string.

```python
def join_names(first, last):
    return ' '.join([first.capitalize(), last.capitalize()])


return_data =  join_names('john', 'doe')
print(return_data)
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

```
E:\Projects\Sana\Course - Python>python section_three.py
John Doe
```

Introduction to Testing

- Now, the method is suppose to receive two input and return one string with space separated two words, each capitalized. Let’s write a test case to ensure that our code does the following things it is supposed to:
  - Return a data type of string
  - The string contains two words that are separated by space
  - Each of those words is capitalized

- Intention is to ensure our code behaves exactly how we want them to. Hence, you can work on improving your code as long as it returns two strings which are capitalized.

assertion methods for testing validity of condition. In this case, whether the string is capitalized. If true, the case passes, if false, it throws an assertion error and the case fails.

```python
return_data =  join_names('john', 'doe')
print(return_data)


class TestStringMethods(unittest.TestCase):

    def test_is_capitalized(self):
        temp1, temp2 = return_data.split(' ')
        self.assertTrue(temp1.istitle())
        self.assertTrue(temp2.istitle())

    def test_length(self):
        self.assertTrue(len(return_data.split(' ')), 2)

    def test_split(self):
        self.assertEqual(type(return_data), str)


if __name__ == '__main__':
    unittest.main()
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

```
E:\Projects\Sana\Course - Python>python section_three.py
John Doe
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

The user-defined class TestStringMethods inherits the TestCase class. This is part of the unittest library which instructs python that the class is used for unittesting. Each method defined within a class acts as a test. Code does not break when an error is encountered in any of the methods but collected as failed test.

The overall result for all of the test cases and the status is then displayed when the main() method from the library is run. The interpreter automatically identifies it as a unittest.

Introduction to Testing

- Suppose we intentionally provided incorrect data for testing. Let’s set the return_data to ‘john doe’. Our first test test_is_capitalized() should fail and thus happens.
- As you notice, it prompts an assertion error and the status is FAILED.

```python
# return_data =  join_names('john', 'doe')
# print(return_data)
return_data = 'john doe'
print(return_data)
```

```
======================================================================
FAIL: test_is_capitalized (__main__.TestStringMethods)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "section_three.py", line 17, in test_is_capitalized
    self.assertTrue(temp1.istitle())
AssertionError: False is not true

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)

E:\Projects\Sana\Course - Python>
```

Introduction to Testing

- Testing ensures the behavior of your functions, classes and variables are as intended.
- In our example, we used two important testing methods assertTrue and assertEqual, each of which tests fulfill condition and for equality of parameters, respectively.
- The unittest library comes with additional methods for testing a variety of conditions. You can use whichever suits your circumstances.

Introduction to Testing

Method	Description
assertEqual(a, b)	Tests equality of a and b
assertNotEqual(a, b)	Tests inequality of a and b
assertTrue(x)	Tests if condition evaluates to True
assertFalse(x)	Tests if condition evaluates to False
assertIs(a, b)	Tests if a is the same as b
assertIsNot(a, b)	Tests if a is not the same as b
assertIsNone(x)	Tests if x is None
assertIsNotNone(x)	Tests if x is not None
assertIn(a, b)	Tests if a exists in b
assertNotIn(a, b)	Tests if a does not exist in b
assertIsInstance(a, b)	Tests if a is an instance of class b
assertNotIsInstance(a, b)	Test if a is not an instance of class b

Debugger

- Debugging is a very useful skill for any programmer.
- It is the ability to assess where the problem in your code is whenever it fails.
- Usually, error messages in Python are very descriptive and easy to understand, however, sometimes when your code works fine but produces the wrong result, you need to step into the code, line by line, to see where the problem is arising.
- For this, you can use the pdb library. Most IDEs now come with interactive ways to debug, however, its good to know the bare bones way to do it.

Debugger

The -> indicates the line on which the debugger is at.

```python
import pdb

def sum_values(a, b):
    return a + b

def test_function():

    pdb.set_trace() # we have added a breakpoint here. The code pause execution here.
    print('This is the first line')
    print("This is the second line.")
    value  = sum_values(2, 3)
    print('The code is done!')
    return value

test_function()
```

The debugger stops the code execution right after this line and waits until user input is provided.

The s command instructs the debugger to step into the specific method, in our case, sum_values.

```
E:\Projects\Sana\Course - Python>python section_three.py
> e:\projects\sana\course - python\section_three.py(76)test_function()
-> print('This is the first line')
(Pdb) n
This is the first line
> e:\projects\sana\course - python\section_three.py(77)test_function()
-> print("This is the second line.")
(Pdb) n
This is the second line.
> e:\projects\sana\course - python\section_three.py(78)test_function()
-> value  = sum_values(2, 3)
(Pdb) s
--Call--
> e:\projects\sana\course - python\section_three.py(70)sum_values()
-> def sum_values(a, b):
(Pdb) n
> e:\projects\sana\course - python\section_three.py(71)sum_values()
-> return a + b
(Pdb)
```

You can access the internal variables of sum_values here.

Debugger

- The first line of our test_method, has the trace. The debugger stops right after executing that line. You can use controls such as n to move to the next line, s to step into the function being called (as we do to access the sum_values() method).
- While inside the debugger, you can access the internal variables as they are created. For instance, while inside the sum_values method, you can access the values of a and b.
- There are more controls that you can use while interacting with the debugger in the console as shown in the next slide.

Debugger

Command	Key	Description
Next	n	Execute the next line
Print	p	Print the value of the variable following p
Repeat	Enter	Repeat the last entered command
List	l	Show few lines above and below the current line
Step	s	Step into a subroutine
Return	r	Run until the current subroutine returns
Continue	c	Stop debugging the current breakpoint and continue normally
Quit	q	Quit pdb abruptly

TESTING AND DEBUGGING

EXERCISE - 1

- Create code for a calculator that preforms the following operations:
  - add, subtract, multiply, divide, power, exponent.
- Write unit tests for each of the method of operations.
- Add debug traces for each method and log results.

Decorators

- Decorator is a powerful feature that allows programmer to modify the behavior of functions or classes. It works by wrapping itself around an existing method and executing sequentially.
- You can define certain checks or behaviors that does some, say, pre-requisite work or additional work before your method does.
- In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.
- Let’s have a look at an example. Here, we will use a decorator to retrieve and display the name of the method being called.

Decorators

We define our decorator as a nested function. The inner method does the computation of the intended method passed as the argument along with additional work, in our case display the name of the function.

```
def extract_function_name(func):
    def internal_method(*args, **kwargs):
        print('The method called is:', func.__name__)
        returned_value = func(*args, **kwargs)
        print('The method execution is complete.')
        return returned_value
    return internal_method

# adding decorator to the function
@extract_function_name
def sum_two_numbers(a, b):
    print("This is called inside the function")
    return a + b

@extract_function_name
def product_two_numbers(a, b):
    print("This is called inside the function")
    return a*b

a, b = 3, 4
# getting the value through return of the function
print('Sum function value:', sum_two_numbers(a, b))
print('Product function value', product_two_numbers(a, b))
```

The @ symbol is used to refer to the decorator. Notice how it is written right on top of the function being attached to.

Decorators

- By attaching the decorator method (using the @ symbol), at the top of each of these methods, we instruct Python to run the decorator method the moment this method is called.
- Hence, it retrieves the method name inside decorator, prints it, runs the actual method, and then ends the execution.

Decorators

This statement is printed by the decorator.

```
E:\Projects\Sana\Course - Python>python section_three.py
The method called is: sum_two_numbers
This is called inside the function
The method execution is complete.
Sum function value: 7
The method called is: product_two_numbers
This is called inside the function
The method execution is complete.
Product function value 12
```

This statement is printed by the decorator.

Lambda

- Lambda is a way of defining anonymous functions in Python.
- It helps you write methods in line expressions which is not just neat but comes with its own advantages.
- Since they are anonymous, you can define them inside of a method, extend its functionality and the functionality of the method. Let’s have a look at an example.
- Let’s define a lambda function that raises any provided number by the power n. We will then define functions using the lambda function like cube_it, square_it, etc.

Lambda

The lambda function accepts one argument, here a, and then the logic to be performed on the argument are separated by “ : “

```python
def my_power_raiser(n):
    return lambda a : a ** n

square_it = my_power_raiser(2)
cube_it = my_power_raiser(3)
quad_it = my_power_raiser(4)

input_number = 3
print('Our input is:', input_number)
print('square_it function raises input by power 2: ', square_it(input_number))
print('cube_it function raises input by power 3: ', cube_it(input_number))
print('quad_it function raises input by power 4: ', quad_it(input_number))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
E:\Projects\Sana\Course - Python>python section_three.py
Our input is: 3
square_it function raises input by power 2:  9
cube_it function raises input by power 3:  27
quad_it function raises input by power 4:  81

E:\Projects\Sana\Course - Python>
```

DECORATORS AND LAMBDA

EXERCISE - 2

- Recreate the calculator as a class.
  - Treat it as a parent class.
  - Create a class called ScientificCalculator and inherit Calculator
- Create scientific functions: log, power, exponent and factorial as lambda functions.
- Set decorator function which prints the inputs and the operation being performed. [e.g. for addition, the decorator function must print the arguments of the function as ‘1 + 3 = 4‘]

Code Profiling

- Efficiency, memory consumption and speed are the dimensions on which your code is usually evaluated.
- If in any terms, you wish to improve, you would need to assess how your code really performs in terms of statistics.
- You would need to measure time each line or method takes, the number of times it is called and so on.
- This technique is referred to as profiling. Python has a built-in library that allows you to profile your code easily providing key statistics about your program. The library is cProfile.
- Let’s have a look at some example code and run profiling.

Code Profiling

```python
import cProfile

def internal_method():
    temp_var = 0
    for ind in range(10):
        temp_var += 1
    return temp_var

def external_method():
    counter = 0
    for val in range(10):
        counter += internal_method()
    print('Total iterations:', counter)
    return

cProfile.run('external_method()')
```

- Our code contains two methods, internal_method() and an external_method().
- External_method() calls the internal_method() inside it, in a loop that iterates 10 times.
- Internal_method() itself runs a loop on iteration of length 10.

The run method accepts one argument as string, which is the call to the method to be profiled.

Code Profiling

```
E:\Projects\Sana\Course - Python>python section_three.py
Total iterations: 100
         15 function calls in 0.002 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.002    0.002 <string>:1(<module>)
       10    0.000    0.000    0.000    0.000 section_three.py:87(internal_method)
        1    0.000    0.000    0.002    0.002 section_three.py:93(external_method)
        1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
        1    0.002    0.002    0.002    0.002 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}



E:\Projects\Sana\Course - Python>
```

- There are total 15 function calls, (inclusive of built-in methods print and exec).
- The total run time is 0.02 seconds. The method that took the most time was the built-in method, print.

Exercise

- Create both calculators in exercise 2 (Calculator and Scientific Calculator).

- Write two methods, each creates object of each of the above class and runs all the corresponding methods.
  - Which runs faster and why?
  - Which aspect of each calculator takes the longest?
  - Which operation is performed the fastest?

Recap

- Testing
- Python Debugger
- Decorators
- Lambda
- Code Profiling

جزاك الله

To ask questions, Join the Al Nafi Official Group

https://www.facebook.com/groups/alnafi/

(This group is only for members to ask questions)