David Goldberg

May 28, 2024

Foundations of Programming: Python

Assignment 08

Link to Github:

# Modules and Unit Tests

## Introduction

In this week's assignment we learned the importance of modules and how to integrate them into your code. We also learned the "main" module, its importance, and how to implement it so it completes each module. On top of that, we also learned how to use Unit Testing and how it is implemented and used in our script.

## Modules

Modules are a way you can break classes down into separate Python files and use them when necessary. Modules help organize and reuse code which makes it easier to manage large programs by breaking them down into smaller components. For example, if your script has 4 classes, you can create a folder where each class is now its own Python file. Using the import function allows you to import the specific module you wish.

```
10  from data_classes import Employee
11  from processing_classes import FileProcessor
12  from presentation_classes import IO
```

*Figure 1: Module example*

In the code above, there are 3 different modules getting imported into the main script. The module's file name, data_classes, imports the specific class inside, Employee. You can also use an alias which would change line 10 to "import data_classes as dc" where dc is the alias. Dc then must get written into every time that module gets called, which can be a fair amount of work. Modules are a great way to organize your code and have a clean script. This is shown in the figure above because the use of modules allows for 3 lines of code rather than a few hundred lines of code.

## Main Method

The purpose of the main method is to enable a script to function both as a standalone program and as a reusable module. The example below shows the use of main method.

```
128  def main():
129      print("Hello, World!")
130
131  if __name__ == "__main__":
132      main()
```

*Figure 2: Main Method*

Typically, you define a function 'main' to encapsulate the main logic of your script, which is shown on line 128 and 129. But line 131 is an idiom that checks weather the script is being run directly or being imported. If it's being run directly, the code within the 'if' block executes. This ensures that the 'main' function is called only when the script is executed directly, not when it is imported as a function.

## Unit Testing

Unit testing is a test where individual units or components of a script are tested separately, in its own Python file, to ensure that they work as intended. The goal of using unit testing is for validation of the script and makes sure that the script performs as it should be. The first step of writing a unit test is to import the built-in module unittest. The rest is shown in the example below

```
1  import unittest  # Library that automates unit tests
2  from data_classes import Person
3  from data_classes import Employee
4
5  class TestPerson(unittest.TestCase):          # anything i
6      def test_person_innit(self):          # test initializ
7          person=Person('David', 'Goldberg')
8          self.assertEqual('David', person.first_name)
9          self.assertEqual('Goldberg', person.last_name)
```

*Figure 3: Unit Testing Example*

As seen above, right away it is important to import the necessary modules we need to test our code. It seems like we'll be needing a Person class and an Employee class as well as the built-in unittest module. The class on line 6 'TestPerson' inherits unittest.TestCasew which allows TestPerson to become a test case that can contain multiple test methods. The function inside of the class creates an instance of Person (line 7). Lines 8 and 9 use assertEqual method to check whether first name and last name is equal to 'David' and 'Goldberg'. If this fails, there will be an error output. By running this test, you can verify that the 'Person' class initializes the proper instances of first_name and last_name.  Overall, unit tests are an extremely important part of the scripting process and ensure that each component in a script is working correctly. *Figure 3* is just one example of a unit test, but scripts that contain many components require a lot more unit tests for more efficiency. This week's assignment required us to write a unit test for major components for each of our modules.

## Writing the Script

Compared to last week's assignment, it is very similar but now uses employees instead of students, and instead of course name, it uses review date and review rating. The goal of this assignment is to create unit tests for the Assigment08 starter file. Before we create the first unit test, it is important to split up each class into its own module to then be called into the main script. The modules are data_classes.py, presentation_classes.py, and processing_classes.py. Data_classes.py contains the class Person and Employee which contain properties for the employee's first name, last name, review date, and review rating. The presentation_classes.py module contains the class IO which is responsible for any data input/output. It takes in the users choices, outputs users choices, and includes error handling. The final module processing_classes.py contains methods for reading from and writing to the json file. All the classes in the modules have very similar functionality to the script from last week. Creating these modules allows for cleanliness in the main method

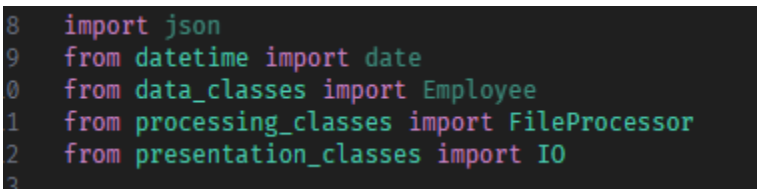


*Figure 4: module imports*

The example above shows imports for each of the modules. What this does is imports a specific class from a module. For example, if we want the IO class from presentation_classes.py, we will write line 12. This is a great way to reduce the amount of code in the main module. Once all the modules are created, I was able to write a unit tests for each module, starting with test_data_classes.py, which is module for unit tests.

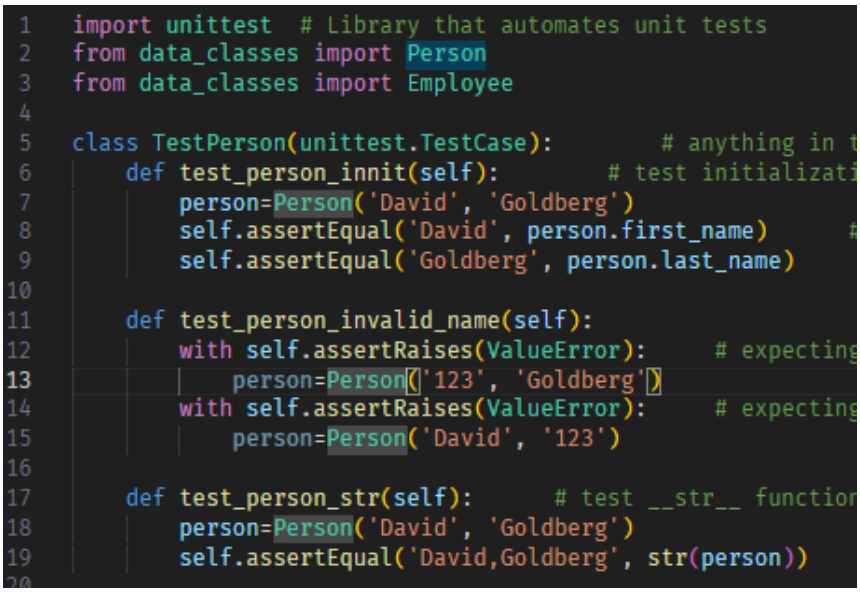


*Figure 5: TestPerson Unit Test*

For the test_data_classes we need to first import the modules we will use, which are the Person and Employee class since we are testing those. The test case is on line 5 which tests the intances of Person, which is first and last name of the employee. If the instance given is an expected input and will tell the test with assertEqual that is the expected instance of first and last name. It will also do a unit test for ValueError and show what instance of Person will cause a Value error.

```
21  class TestEmployee(unittest.TestCase):
22      def test_employee_innit(self):
23          employee=Employee('David', 'Goldberg', '2024-02-26', 4)
24          self.assertEqual('David', employee.first_name)      # automation that is from test case, check if first name i
25          self.assertEqual('Goldberg', employee.last_name)
26          self.assertEqual('2024-02-26', employee.review_date)
27          self.assertEqual(4, employee.review_rating)
28
29      def test_employee_invalid_review_data(self):
30          with self.assertRaises(ValueError):
31              employee=Employee(first_name='David', last_name='Goldberg', review_date='2001/04/26', review_rating= 4)
32
33      def test_employee_rating_out_of_range(self):
34          with self.assertRaises(ValueError):
35              employee=Employee(first_name='David', last_name='Goldberg', review_date='2024-02-26', review_rating= 6)
36
37
38
39
40  if __name__ == '__main__':
41      unittest.main()     # when running the test, run all tests in the file
```

*Figure 6: TestEmployee unit test*

The next test involved the next two employee inputs, review date and review rating. The first method takes a proper set of inputs as an instance. Since we know what inputs don't cause any errors we are able to make this unit test for a correct instance of Employee. The next two methods create a test for improper inputs for the review_data and review_rating. If review date is inputted with an incorrect format and if review rating is set to a rating outside of 1-5, that is a test we want to run. Line 40 is an expression that is used to determine where the script is being run as a main method or is being imported into another script. If this script is being called into another script, the value '__name__' is set to the name of the module.

The next module I ran unit tests on was for the processing_classes.py. This means I needed to create another module called test_processing_classes.py that will hold unit tests for the classes in that module.

```
21      def test_read_data_from_file(self):     # creating sample data in the temp fiile
22          sample_data= [
23              {"FirstName": "David", "LastName": "Goldberg", "ReviewDate": "2024-04-26", "ReviewRating": 4},
24              {"FirstName": "John", "LastName": "Peter", "ReviewDate": "2024-04-26", "ReviewRating": 3}
25              ]
26
27          with open(self.temp_file_name, 'w') as file:  # open temporary file
28              json.dump(sample_data, file)
29
30          employees = FileProcessor.read_employee_data_from_file(self.temp_file_name)     # read in temp file
31          self.assertEqual(len(sample_data), len(employees))      # validation: length of sample data is equal
32
33          self.assertEqual(sample_data[0]['FirstName'], employees[0].first_name)
34          self.assertEqual(sample_data[0]['LastName'], employees[0].last_name)
35          self.assertEqual(sample_data[0]['ReviewDate'], employees[0].review_date)
36          self.assertEqual(sample_data[0]['ReviewRating'], employees[0].review_rating)
37
38          self.assertEqual(sample_data[1]['FirstName'], employees[1].first_name)
39          self.assertEqual(sample_data[1]['LastName'], employees[1].last_name)
40          self.assertEqual(sample_data[1]['ReviewDate'], employees[1].review_date)
41          self.assertEqual(sample_data[1]['ReviewRating'], employees[1].review_rating)
```

*Figure 7: Read File Unit Test*

This unit test takes in sample data, which is correct data we expect to be inputted by the employee. Then we call the read_employee_data_from_file method from FileProcessor and checks if the length of the sample data is the same length as data inputted from the employee. This will fail if the lengths are not the same. Lines 33-41 are assertions to check if each field in the sample data matches the corresponding field in the inputted 'employees' list. This tests validates the read_employee_data_from_file through the use of sample data that has been inputted into a temporary JSON file.

```
43      def test_write_data_to_file(self):      # testing writing to file
44          sample_data= [
45              Employee('David','Goldberg','2024-04-26',4),
46              Employee('John','Peter','2024-04-26',3)
47          ]
48
49          FileProcessor.write_employee_data_to_file(self.temp_file_name, sample_data)
50
51          with open(self.temp_file_name, 'r') as file:
52              file_data = json.load(file)
53
54          self.assertEqual(len(sample_data), len(file_data))
55
56          self.assertEqual(sample_data[0].first_name, file_data[0]['FirstName'])
57          self.assertEqual(sample_data[0].last_name, file_data[0]['LastName'])
58          self.assertEqual(sample_data[0].review_date, file_data[0]['ReviewDate'])
59          self.assertEqual(sample_data[0].review_rating, file_data[0]['ReviewRating'])
60
61          self.assertEqual(sample_data[1].first_name, file_data[1]['FirstName'])
62          self.assertEqual(sample_data[1].last_name, file_data[1]['LastName'])
63          self.assertEqual(sample_data[1].review_date, file_data[1]['ReviewDate'])
64          self.assertEqual(sample_data[1].review_rating, file_data[1]['ReviewRating'])
```

*Figure 8: Write File Unit Test*

This test case has the same functionality as the previous "test_read_data_from_file" which take sample data and make sure lengths that are written to the JSON file are the same or else there will be an error given to the user. This uses a temportary json file to store and write to so it will not interfere with the json file used through the main body of the script.

```
6       def test_get_input(self):       # want input choice to be automated (patch the function)
7           # patch function will overwrite python. override builtins.input
8           # return the value 2
9           with patch('builtins.input', return_value='2'):
10              choice=IO.input_menu_choice()
11              self.assertEqual('2', choice)
12
13      def test_input_employee_data(self):
14          # Simulate user input for student data
15          with patch('builtins.input', side_effect=['David', 'Goldberg', '2024-04-26', 4]):
16              employees = []
17              employees=IO.input_employee_data(employees)
18              self.assertEqual(1, len(employees))
19              self.assertEqual( 'David', employees[0].first_name)
20              self.assertEqual('Goldberg', employees[0].last_name )
21              self.assertEqual('2024-04-26', employees[0].review_date)
22              self.assertEqual(4, employees[0].review_rating)
```

*Figure 9: IO Test Case*

The last module left to test is presentation_classes.py. For this we create a test case module called test_presentation_classes.py. With the patch function being used, it will temporarily replace the 'builtin' input function to always return '2' when called during the test. This replaces the need for a user interaction. This will check if the input_menu_choice will correctly return the input '2'. The next test case is for inputting employee data which also uses the patch function by simulating the user input shown in line 15. The asserting the results section checks if the length of the employee record has been added to the list (on line 18). The rest of the 'self.assertEquals' check if the proper information has been inputted by the employee correctly. The purpose of this test are to verify that the menu choice and the employees inputs work correctly with the user input.

## Summary

Overall, this module was full of valuable information that will make us efficient and awake script writers in the future. The usage of modules creates code reusability and maintainability by allowing us to encapsulate functions inside individual files that can easily be imported. The use of modules makes the use of test cases easier. Unit Tests are extremely crucial because they ensure that an individual component of a script is working properly. It allows for early detection of bugs and increases code quality and reliability. This module will help me create more complicated scripts and increase the quality of them as well.