1 Introduction to unit testing and refactoring in common script languages

Table of Contents

1.0 Background to unit testing in Python

This lab will introduce the unit testing framework "unittest" in Python https://docs.python.org/3/library/unittest.html. Other testing frameworks in Python are doctest, nose2: https://github.com/nose-devs/nose2 and pytest: https://docs.pytest.org/.

We will begin with following Mark Pilgrims example from the book "Dive into Python 3" found at some mirror as: https://github.com/diveintomark/diveintopython3

Before you begin make sure you got everything installed on your computer so you can run and develop Python software as well as JavaScript. I recommend Visual Studio Code (User Installer): https://code.visualstudio.com/download

VS Code with the MS Python Extension have since February 2019 Test Explorer functionality: https://devblogs.microsoft.com/python/category/visual-studio-code/

MS docs for VS Code, Python and testing: https://code.visualstudio.com/docs/python/testing

1.1 Intro to unit testing in Python

Report:

1. Hand in the full source code of your final solution, which can be tested
2. and also run as a normal script.
3. Find, implement and report at least one important missing test case.
4. Also report your test coverage output for statements and branches etc.

Task:

1) This task **should be done thoroughly** so you understand the concept of automated software testing with unit testing and TDD (Test Driven Development). The lab will lay the

Högskolan Dalarna
Röda vägen 3
781 88 BORLÄNGE

Telefon: 023-77 80 00
Telefax: 023-77 80 50
URL: http://www.du.se/

foundation to your software testing knowledge and should preferable be understood to a high degree in order to understand similar concepts further on in the course.

Get the free book ”Dive into Python 3” and the source code for all examples in it. Local book copies (pdf and html) are found at: http://users.du.se/~hjo/cs/gmi2j3/ > docs.and.code

Navigate to chapter 9, Unit Testing and follow the walk-thru lab dealing with Roman Numerals. You may have to fresh up you Python knowledge during the lesson, so preferable use the books HTML version which allows easy jumping back and forth in the text.

To discover Unit Test on your storage drive in VS Code use: Ctrl+Shift+P. I strongly suggest you read the MS Python and testing docs or devblog articles describing Test Explorer (VSCode links are given above) to understand how Test Explorer fully works. It is however not required to use Test Explorer in the lab. If you follow the “Dive into Python 3” book issuing console commands it works fully OK as well. Or execute with:

```
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

You should also study the “unittest” Unit testing framework documentation at: https://docs.python.org/3/library/unittest.html. Especially the assert methods.

When you have finished chapter 9, continue with the lessons in chapter 10, Refactoring. Renaming (which is a common refactoring task) in “Visual Studio Code” is made by marking the text you want change, press F2, do the change and then press enter. This may not work so you may have to fix it: https://www.google.com/search?q=vscode+f2+rename+not+working

When finished with chapter 10, make sure you have grasped at least the unit testing concept. Achieving this you can perform the same work with code you have created by yourself.

2) Add some code so an ordinary user can execute the program from a console, i.e. run the functions to_roman() and from_roman().

3) The author of “Dive into Python 3” have missed to implement a couple of obvious test cases. Find, implement and report at least one important missing test case in the code. Remember that adding code so the module which can be run by the user should not contain any “unit test like” code for bad user input. This since we want to make our module portable. Hints:
1. Can to_roman() handle string input?
2. Can from_roman() handle lowercase chars?

4) Use the Python Coverage tool: https://coverage.readthedocs.io and report the effectiveness of your unit tests. However note that achieving 100% coverage does not mean that there is no bugs left in the code: https://martinfowler.com/bliki/TestCoverage.html

Högskolan Dalarna
Röda vägen 3
781 88 BORLÄNGE
Telefon: 023-77 80 00
Telefax: 023-77 80 50
URL: http://www.du.se/

1.2 Unit testing in Python on your own

Report:

1. Hand in the full source code of your solution, which can be tested
2. and also run as a normal script.
3. Also report your test coverage output for statements and branches etc.

Task:

1)

With the help of and knowledge from the previous task perform unit testing in Python with the files: leapyear.py and test_leapyear.py. Try to follow the TDD (Test Driven Development) principle if possible.

Implement the to_leap_year function in leapyear.py.

```
def to_leap_year(year):
```

When you are finished with the unittest implementation it will output **at least** something like this when running your tests via the console. You have to write like this to get the verbose and nice console output:

`python -m unittest –v`

or

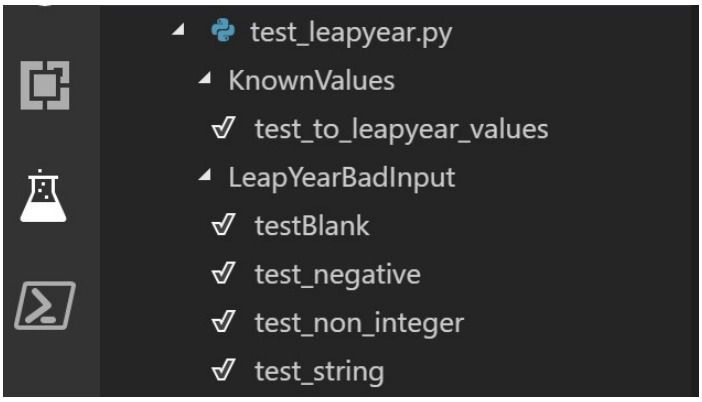
`python .\test.py -v` with empty __main__ args as: `unittest.main()`

```
C:\myhome\azure-devops\gmI23n\leapyear> python .\test_leapyear.py -v
test_to_leapyear_values (__main__.KnownValues)
to_leap_year should give known result with known input …
2000 is a leap year
1904 is a leap year
2400 is a leap year
2020 is a leap year
2010 is not a leap year
1900 is not a leap year
1800 is not a leap year
2002 is not a leap year
ok
testBlank (__main__.LeapYearBadInput)
to_leap_year should fail with blank string ... ok
test_negative (__main__.LeapYearBadInput)
to_leap_year should fail with negative input ... ok
test_non_integer (__main__.LeapYearBadInput)
to_leap_year should fail with non-integer input ... ok
test_string (__main__.LeapYearBadInput)
to_leap_year should fail with string input ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.005s

OK
```


Högskolan Dalarna
Röda vägen 3
781 88 BORLÄNGE

Telefon: 023-77 80 00
Telefax: 023-77 80 50
URL: http://www.du.se/

Using the VS Code Test Explorer functionality it should look **at least** something like this:



2)
When your tests are done add some code so a user can use the program from a command line prompt, i.e. run the function `to_leap_year()` via the `leapyear.py file`.
Remember that adding code so the module can be run by the user should not contain any "unit test like" code for bad user input since we want to make our module portable.

3)
Then use the Coverage tool and report the effectiveness of your unit tests.

1.3 Testing and refactoring with JavaScript

Report:

1. Hand in the source code of your solution which have all the test cases implemented
2. and code which is refactored.

Task:
1)
Implement the missing test cases using the assert function in the supplied code files (prime_calc1s.js and prime-assert1.html).
The assert function is explained in the article "Quick and Easy JavaScript Testing with Assert" which is attached and also found here: https://code.tutsplus.com/tutorials/quick-and-easy-javascript-testing-with-assert--net-14050

Note that this task does not use any Unit Test framework, we only use our own assert function/method. If you want to use a JavaScript unit testing framework in this task it is OK with me. You can begin reading "An Overview of JavaScript Testing in 2021": https://medium.com/welldone-software/an-overview-of-javascript-testing-7ce7298b9870

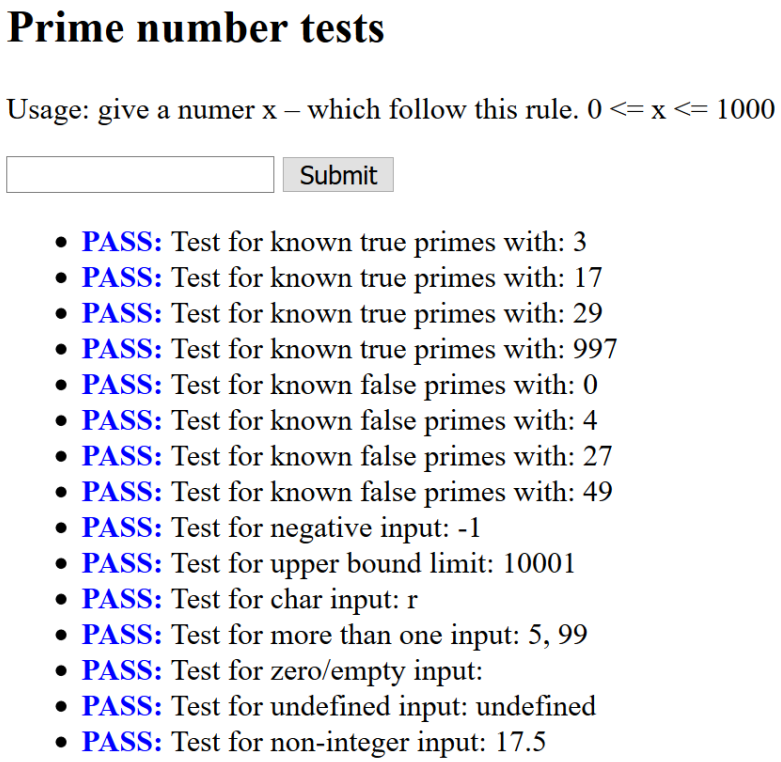
2) Then refactor the prime calculation so it executes more efficient.

1.3.1 Detailed description

Test Cases
First you must implement all the test cases so you know that your refactoring will not change the output from the prime calculation algorithm.

Högskolan Dalarna
Röda vägen 3
781 88 BORLÄNGE
Telefon: 023-77 80 00
Telefax: 023-77 80 50
URL: http://www.du.se/

Implement **at least** the 9 test cases (test for known true/false primes counts as two test cases) found in the checkTest() function so you will have an output like below when pressing the Submit button.



Examine the “function check(num)” so you understand how the “function assert(outcome, description)” works and can be called.

Refactor
When finished,

- refactor the code so it only calculates the prime array once for repeated prime checks during a session.
- Also only do the search of prime candidates up to the square root of max for the factors since we only need to test factors up to this limit.
  Explanation: https://stackoverflow.com/questions/5811151/why-do-we-check-up-to-the-square-root-of-a-prime-number-to-determine-if-it-is-pr
- Remove any duplicated code using DRY (Don't Repeat Yourself) principle if present.
- Lastly verify that all your tests still works after you have refactored the code. This is the great benefit with unit tests. After large changes of the code validating the code is only a muse click away!

1.4 Lab feedback

a) Were the labs relevant and appropriate and what about length etc?
b) What corrections and/or improvements do you suggest for these labs?

Högskolan Dalarna
Röda vägen 3
781 88 BORLÄNGE
Telefon: 023-77 80 00
Telefax: 023-77 80 50
URL: http://www.du.se/