Mandatory Assignment 3

Basic Python programming (15 points)

University of Oslo - IN3110/IN4110

Fall 2019

Your solutions to this mandatory assignment should be placed in the directory `assignment3` in your Github repository.

Your delivery should contain a file called `README.md` containing information on how to run your scripts. In particular, it is important that you document how to run your tests.

In addition, your code should be well commented and documented. All functions should have docstrings explaining what the function does, how, and an explanation of the parameters and return value (including types). We recommend you use a well-established docstring style such as the Google style docstrings[1]. However, you are free to choose your own docstring style.

3.1 `wc` (3 points)

Make a Python implementation of the standard utility `wc` which counts the words of a file. When called with a file name as command line argument, print the single line `a b c fn` where `a` is the number of lines in the file, `b` the number of words, `c` the number of characters, and `fn` the filename.

Further, extend your script so that it can be called as `wc *` to print a nice list of word counts for all files in the current directory, or `wc *.py` to print a nice list of word counts for all python scripts in the current directory.

Exactly what constitues a word is not very important. A simple approach where word are separated by space is enough.

Name of file: `wc.py`

3.2 Unit tests for complex numbers (3 points)

In the rest of this assignment, you are going to work with *test driven development*. Before you write your code, you write tests that can confirm that your code works as expected. With this, we achieve (at least) two things

[1] `https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html`

1. We are forced to think about what our code actually should do, before we start coding. Planning ahead is generally a good idea, and test driven development forces us to do exactly that.

2. It is easier to check that changes we make to our code doesn't break anything that worked before.

For instance, if you want to write an addition function `plus(a, b)`, you would expect that 2 and 2 becomes 4. This can be formalized as a unit test as follows:

```
def test_two_plus_two():
    assert plus(2, 2) == 4
```

A test should by convention have a name starting with `test_`, and raise an `AssertionError` if the test fails (this is what the `assert` statement does). The test should always test the same thing, i.e. generating something random in the test is usually a bad idea. If you do this, you might end up with tests that *sometimes* pass, which makes debugging difficult.

In the next problem you are going to implement complex numbers in python (they actually already exist, but we will create our own implementation). A complex number, z, can be written on the form $z = a + bi$, where the real part a and the imaginary part b are real numbers that can be represented on the computer as floating point numbers. i is the imaginary unit with the property that $i^2 = -1$. Addition and subtraction of two complex numbers, $z_1 = a_1 + b_1 i$ and $z_2 = a_2 + b_2 i$, can be performed separately for the real and imaginary parts,

$$z_1 - z_2 = (a_1 - a_2) + (b_1 - b_2)i$$

For multiplication we must remember that $i^2 = -1$,

$$z_1 z_2 = (a_1 + b_1 i)(a_2 + b_2 i) = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1)i$$

The last two operations you need to know for this assignment are

- Conjugate: $\overline{z} = a - bi$
- Modulus: $|z| = \sqrt{a^2 + b^2}$. (This is the "length" of the number when we consider it as a point in the complex plane)

For our implementation, the goal is to define a class `Complex` that can be used as follows:

```
z = Complex(1, 2) # the complex number 1+2i
print(z)          # Prints a nice string representation of z (i.e 1+2i)
w = Complex(0, 1) # the complex number i
print(z + w)      # should be 1 + 3i
```

Take a look at the stub `complex.py` to see which methods should be available. Before you start filling out the stub (which is the next exercise), write some unit tests. Implement the following tests:

- Three or more tests verifying that adding two complex numbers returns what it's supposed to
- Three or more tests that subtracting two complex numbers returns what it's supposed to
- Three or more tests verifying that multiplication of two complex numbers and of a constant with a complex number both work.
- Three or more tests checking that the conjugate and modulus method works.
- Three or more tests showing that the `__eq__` method works.

It is of course close to impossible to catch everything that might go wrong with your code. However, this does not mean that you can go for the easiest tests. Try to cover different scenarios for the three (or more) tests of each functionality, such as when the real or imaginary part is zero and the other is non-zero.

It is recommended to work mostly with integer values, to avoid rounding errors that can make comparisons more difficult.

We recommend implementing your tests with pytest[2]. The tests should live in a separate file, so you must import the Complex class to properly run your tests.

Name of file: `test_complex.py`

3.3 Implement complex numbers (4 points)

Implement all the methods that you have developed tests for in the previous task. That is, implement the methods in the first part of `complex.py`.

Note: You should not use the built-in complex numbers in Python to do your calculations.

Name of file: `complex.py`

3.4 Make your implementation work with Python's complex numbers (5 points)

Change your implementation such that we also can combine our complex numbers with Python's complex numbers, and real numbers (ints and floats). e.g., such that `Complex(2,3) + (2+2j) == Complex(4,5)`, or `4*Complex(3,4) - 2 == Complex(10,2)`. You will also have to implement `__radd__`, `__rsub__` and `__rmul__`. Implement these in terms of `__add__`, `__mul__` etc. This way, you don't have to do the same work twice.

Before starting, you should create new tests to check that this works.

[2] `http://doc.pytest.org/en/latest/getting-started.html`

3.4.1 About `__rmul__` etc.

When you write e.g. `a * b` in Python, this is executed as `a.__mul__(b)`. In our case, `a` might be an integer, and `b` might be an instance of our `Complex` class. This will give us trouble, because `int`'s `__mul__` method does not know how to work with instances of `Complex`. What python will do instead, is trying to execute `b.__rmul__(a)` instead. It does not execute `b.__mul__(a)`, because it could be that `a * b != b * a`.

Name of file: `complex.py`, `test_complex.py`