Mandatory Assignment 3

Basic Python programming (15 points)

University of Oslo - INF3331/INF43331

Fall 2018

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

Your delivery should contain a Readme file containing information on how to run you 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 (includeing types)

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. 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` (this is what the `assert` statement does) if the test fails. 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 that is beside the point). You should be able to do things like the following:

```
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:

- A test verifying that adding two complex numbers of your choice returns what it's supposed to
- A test verifying that subtracting two complex of your choice returns what it's supposed to
- Tests checking that the conjugate and modulus method works.
- A test 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. For instance `Complex(0,0)*Complex(0,0)` might give you correct result, while `Complex(2,3)*Complex(4,1)` might fail, because multiplication with 0 is much easier to get right.

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

Name of file: `test_complex.py`

3.3 Implement complex numbers (4 points)

Implement all the methods that are in the first part of `complex.py`. Here are some useful formulas. Assume that $z = a + bi$

- 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 addition/subtracition and multiplication, complex numbers follow normal, algebraic rules. Just remember that $i^2 = -1$. If you are not comfortable working with complex numbers, formulas can easily be found online.

You should not use the built-in complex numbers in Python to do your calculations. (i.e. convert to Pythons complex numbers, do the calculation, and then convert back).

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.

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. We don't execute `b.__mul__(a)`, becuase we might have a case where `a * b != b * a`.

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

Tips

- It is recommended that you use a test framework. Remember to document this in your Readme file

- Try to implement the methods in `Complex` in terms of each other. For instance, there is a close relationship between addition and subtraction, and this will mean that the reversed operations shouldn't require too much work.