DIT376 Python for Data Science
Assignment 3

There are two problems in this assignment.
To submit:

- A Jupyter notebook (.ipynb) with clearly marked solutions for: Problem 1 and Problem 2. The notebook must show examples that you used to run the codes, and the outputs that you get after running the codes.
- An html file (a result of ”Save and Export Notebook As” html of your Jupyter notebook) after you run all the codes.

Do not forget to add your name and your group partners's name at the very top of your notebook.
Note: If you add any details or make any assumptions, please clearly describe these in your submission.

1 Dunder and static methods

The real numbers $\mathbb{R}$ can be extended to a bigger class numbers in a number of different ways. The ones that can be represented on a 2D-plane are the complex numbers, dual numbers and split complex numbers. In this assignment you will implement the complex numbers in Python.

Complex numbers are used frequently in signal processing, physics, computer graphics, computer vision and wireless networks because they are a very efficient method to describe 2 dimensional rotations and sinusoidal signals. In machine learning, complex numbers are used in complex-valued neural network (i.e., a neural network that has at least one complex-valued layer). Such networks have been found to outperform real-valued neural networks on image classification tasks (see `https://pratt.duke.edu/about/news/imaginary-numbers-machine-learning`).

1.1 Properties of complex numbers

Several properties of the complex numbers are discussed here. In this assignment, you will need to implement these properties in Python code.

A complex number z, can be represented as:

$$z = a + ib \tag{1}$$

Here, a and b are both real numbers and i is the imaginary unit equal to $\sqrt{-1}$. The part a is called the real part of the complex number and b is called the imaginary part, in symbols:

$$Re(z) = a \qquad Im(z) = b \tag{2}$$

Just like with real numbers, we can apply the following operations with complex numbers:

- addition

$$z + w = (a + ib) + (c + id) = (a + c) + i(b + d) \tag{3}$$

- subtraction

$$z - w = (a + ib) - (c + id) = (a - c) + i(b - d) \tag{4}$$

- multiplication

$$z \cdot w = (a + ib) \cdot (c + id) = ac + iad + ibd + i^2bd = (ac - bd) + i(ad + bc) \tag{5}$$

  where we have made use of the fact that $i^2 = -1$

- division. Division is slightly more complicated and involves the conjugate of a complex number. The conjugate $\bar{z}$ is given by flipping the sign of the imaginary part, in symbols:

$$z = a + ib \qquad \bar{z} = a - ib \tag{6}$$

  so the conjugate of $3+4i$ is $3-4i$ and the conjugate of $1-2i$ is $1+2i$. Division is given by:

$$\frac{z}{w} = \frac{z\bar{w}}{w\bar{w}} = \frac{(a+ib)(c-id)}{(c+id)(c-id))} = \frac{ac+bd+i(bc-ad)}{c^2+d^2} \tag{7}$$

- norm. The norm of a complex number is given by:

$$\|z\| = \sqrt{z.\bar{z}} = \sqrt{(a+ib)(a-ib)} = \sqrt{a^2+b^2} \tag{8}$$

1.2 Your tasks

1. Define a class called Complex. This is not to be confused with Python class complex. The class Complex stores the real part (as real) and the imaginary part (as imag) of complex numbers. Make the attributes real and imag as private attributes. Set both real and imaginary parts by default to zero.

2. Define a constructor for the class Complex.

3. Use the @property decorator to write methods for getting and setting the real part and the imaginary part separately. Overwrite the following "dunders" (double underscores) methods to implement the properties of the complex numbers:

   - __str__ for printing the complex numbers
   - __add__ for addition
   - __sub__ for subtraction
   - __mul__ for multiplication
   - __truediv__ for division
   - __eq__ for equality
   - __ne__ for not equal

4. Write two instance methods called conjugate and norm that return the conjugate and norm of a complex number separately.

5. Use your code to find the value of the following statements:

```
z1 = Complex(1, 2)
z2 = Complex(5, -2)
z3 = Complex(-2, 3)
print(z1 + z2)
print(z3 - z1)
print(z1 * z2)
print(z1 * z2 / z3)
print(z1 == z1)
print(z2 != z3)
print(z1.conjugate())
print(Complex.norm(z1*z2*z3))
```

The output should be:

```
6
-3+1i
9+8i
0.46153846153846156-3.3076923076923075i
True
True
1-2i
43.41658669218482
```

6. Import matplotlib.pyplot, and use the scatter function to plot the following points in 2D space: $2+1i$, $-1+3i$, $4-1i$, where the x-axis is the real part of the complex number and the y-axis is the imaginary part of the complex number. Now multiply each of these points by i using your code and plot both sets of points on the same plot, colour code which points are which. Finally multiply the original points by $-i$ and plot those points again with a different colour. Describe what happened to the points when multiplying by i and $-i$.

7. Explain the benefit of built-in *dunder* methods.

8. Explain why one might want to use static methods.

2 Multiple inheritance and method resolution order

This task is about encoding and decoding messages through a cipher with a key. A cipher is a process or algorithm that encodes a piece of data into another piece of data. This alternative form is usually chosen because it has some benefit, such as secrecy or compression. The cipher should also be able to be used to decrypt a message back into its original form.

2.1 Your tasks

1. Define a class Cipher(). All ciphers have a *sender* and a *receiver* as their basic attributes; these will be strings that are the names of the people sending and

receiving. The class should have a constructor and methods for getting and setting these attributes.

2. Create three ciphers: an alphabet cipher, an XOR cipher, and a hybrid cipher. All three of these are classes that will inherit from the Cipher class.

   The alphabet cipher inherits from the Cipher class and has the following additional attribute, a key, which is a number between 0 and 25. The class must have two methods *encode* and *decode*. The encode will take in a message string that will be in all capital letters and return an encoded string. Each letter has a number associated with it, $A = 0, B = 1$, and so on. The encoded string will add the key number to each letter and return a new string. If the number becomes larger than 25 you should loop back around to the beginning. As an example, if the key is 3, the string ABZ will be encoded to DEC because $A(0)B(1)Z(25)$, plus the key 3 will be $D(3)E(4)C(2)$. The decode method should take in an encoded string and return the original decoded string. You can ignore the space characters between words.

   The XOR cipher will also take in a key which will be a capital letter $A-Z$, the key should be only a single character long (you should check for this). Once again we will have an encode function that takes in a message in all capital letters and returns an encoded string, once again you should ignore the space character. The decode method will reverse this process.

   The XOR cipher works by using the XOR binary operation for each character in the message with the key. As an example, we consider the string "TEST" in ascii encoding (in which each character is encoded to a single byte) (see Table 1).

Table 1: Ascii representation of the word "TEST"

T	E	S	T
01010100	01000101	01010011	01010100

   We choose the key P which has a binary representation 01010000. The XOR operation takes in two binary numbers and returns 1 if only one of them is 1 and 0 if both are 0 or both are 1. The truth table is given in Table 2. For the XOR cipher, each character is XORed with the key, as an example see Table 3,

Table 2: XOR operation

	1	0
1	0	1
0	1	0

Table 3: Encoded message "TEST" with key "P"

	T	E	S	T
Message	01010100	01000101	01010011	01010100
Key	01010000	01010000	01010000	01010000
Encoded	00000100	00010101	00000011	00000100

You can decode the string by just XORing it again with the same key. Two XORs will return you where you started. Can you think of a clever way of implementing the decode method that does not involve you copying code?

To implement this, note that $ord()$ gives you the integer value of a character in Python, $ord("H") = 72$ and you can XOR two integers together using the operator (ˆ) in Python (note this is the "to the power of" symbol or caret). Finally you can obtain a character from an integer by using the $chr()$ function.

The Hybrid cipher should inherit from both the alphabet cipher and the XOR cipher and should have attributes for both types of keys. You should use the *super* keyword here to populate the attributes of the classes you inherit from. The hybrid cipher first encodes by means of the alphabet cipher and then encoded the encoded string again by means of the XOR cipher. Decoding, works similarly, first the XOR decoder and then the alphabet decoder. The encode and decode functions of the hybrid cipher should make use of the encode and decode functions from the class that it inherits from. You should not repeat code and rewrite the encoding and decoding functions again.

Here are some examples of statements to the corresponding output to check that your code works well.

```
basicCipher = Cipher("Alice", "Bob")
print(basicCipher.sender)
print(basicCipher.receiver)
```

The output should be:

```
Alice
Bob
```

```
alphaCipher = AlphabetCipher("Alice", "Bob", 2)
encodedString = alphaCipher.encode("PYTHON IS A PROGRAMMING LANGUAGE")
print(encodedString)
decodedString = alphaCipher.decode(encodedString)
print(decodedString)
```

The output should be:

```
RAVJQP KU C RTQITCOOKPI NCPIWCIG
PYTHON IS A PROGRAMMING LANGUAGE
```

```
xorcipher = XORCipher("Alice", "Bob", "A")
encodedString = xorcipher.encode("THIS IS A SECRET MESSAGE")
print([char for char in encodedString])
decodedString = xorcipher.decode(encodedString)
print(decodedString)
```

The output should be:

```
['\x15', '\t', '\x08', '\x12', ' ', '\x08', '\x12', ' ', '\x00',
' ', '\x12', '\x04', '\x02', '\x13', '\x04', '\x15', ' ', '\x0c',
'\x04', '\x12', '\x12', '\x00', '\x06', '\x04']
THIS IS A SECRET MESSAGE
```

```
hybrid = HybridCipher("Alice", "Bob", 12, "F")
encodedString = hybrid.encode("THIS IS ANOTHER SECRET MESSAGE")
print([char for char in encodedString])
decodedString = hybrid.decode(encodedString)
print(decodedString)
```

The output should be:

```
['\x07', '\x15', '\x14', '\x04', ' ', '\x14', '\x04', ' ', '\x0c',
'\x1b', '\x00', '\x07', '\x15' , '\x10', '\x05', ' ', '\x04', '\x10',
'\x0e', '\x05', '\x10', '\x07', ' ', '\x18', '\x10', '\x04', '\x04',
'\x0c', '\x12', '\x10']
THIS IS ANOTHER SECRET MESSAGE
```