LAB 04

1. Tutorial: docstrings

Unlike conventional comments that are generally used to help the developer understand 'how' a segment of the code works, docstrings describe 'what' a segment of the code does. Docstrings can document modules, classes, functions and methods. Docstrings are generally meant for users of the code. Docstrings are identified by triple quotes.

Here is a tutorial on how to write and organise docstrings:
https://www.geeksforgeeks.org/python-docstrings/

From now onwards, consider using docstrings in your code including the solutions to the following exercises.

2. Moving robots

We'll start by building a class *Robot1* for our first version of Robots.
We will think of a Robot as holding some data about itself and as also having certain behaviours. The data is captured by instance variables and the behaviours are captured by instance methods. In our first example, the only data held by a Robot is a battery charge.

Robots will initially have two behaviours: they can get their batteries recharged (by a certain amount), and they can move a certain distance, based on the charge level of their batteries.

Figure 1: source http://www.telovation.com/photos/balancing-robot.jpg

Here is how we might want to call some Robot methods, based on what we have so far.

```
r = Robot1(); # start off with a well-charged battery
r.move(11);               # move around and use up the charge

r.batteryReCharge(2.5);   # get a new charge
r.batteryReCharge(0.5);   # add a bit more
r.move(5);                # move some more
```

Here is a skeleton of the *Robot1* class.

```
class Robot1:

    def __init__(self):
        self.batteryCharge = 5.0

    def move(self, distance):
        pass

    def batteryReCharge(self, charge):
            pass
```

Note that we've made the arbitrary decision that when a Robot rolls off the assembly line, its battery has a charge level of 5.0.

The next thing is to write the bodies of the instance methods. Implement **batteryReCharge()** so that the value of the argument is *added* to the existing *value of batteryCharge*.

Next, implement the method *move()* so that it satisfies the following conditions:

- A Robot can only move when it has a battery charge greater than or equal to 0.5.
- For every unit of movement, the battery charge goes down by 0.5 units.

To make the exercise more interesting, get **batteryReCharge()** to print out the result of charging the battery. In addition, get **move()** to print out something to the terminal for every unit of movement that is executed, and to report if a flat battery is detected. Here's one way of doing it (based on the code used earlier). The numbers in square brackets are meant to represent units of movement. However, you can do this in whatever way you like.

```
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] Out of power!
Battery charge is: 2.5
Battery charge is: 3.0
[1] [2] [3] [4] [5]
```

What have we learned from this very simple example of OOP? The main thing to take away is that data and behaviour can be closely interlinked. In this example, the effect of the method **move()** for Robot *r* depends on the state of *r* when **move()** is called; the number of units moved is dependent on the initial battery charge level. At the same time, behaviour changes the state of the object, since calling **move()** also depletes the battery charge level.

3. Talking robots

In our next robot example, we're going to get the critters to talk. So let's think about the class Robot2. Objects in this class will hold data about possible sayings, and have a method for speaking them; or more precisely, printing them out to the console! As before, we'll start by considering what our code might look like. Here's an example:

```
1. r1 = Robot2()
2. u1 = ["Exterminate, Exterminate!", "I obey!",
3.         "You cannot escape.", "Robots do not feel fear.",
4.             "The Robots must survive!" ]
5. r1.setSayings(u1)
6. print("Robot r1 says: ")
7. r1.speak()
8.
9. print("Robot r2 says: ")
10.     r2 = Robot2()
11.     u2 = [ "I obey!" ]
12.     r2.setSayings(u2);
13.     r2.speak()
```

So here we have created two Robots, r1 and r2, and assigned each of them their own list of sayings. While r1 has a good repertoire, r2 is obsessively obedient. So different Robots can hold different data, in the shape of a list of strings.

Given this code, see if you can figure out how to implement **Robot2**, taking **Robot1** as a starting point (see lecture on 'inheritance'). You may find it convenient to assign your *sayings* the empty list as an initial value.

Define **speak()** so that each time it is called, it picks one of the available sayings at random.

Hint: Check-out the official python documentation available here: https://docs.python.org/3/library/random.html

4. Credit card

In this question, you are going to build a class representing a credit card. The exercise will give you more practice in manipulating strings. It also requires you to use the python *datetime* module in order to get the current year and month, which in turn will tell you whether a card has expired or not.

In a more realistic implementation, we would do a lot more checking of the validity of things like dates and credit card numbers! However, we won't bother with such details right now.

The **CreditCard** data type will contain the following data:

- an *expiryMonth* and an *expiryYear*, both of type *int*; these will determine the card's expiry date.
- a *firstName* and a *lastName*, both of type string; these will determine the name of the card's account holder.
- and finally, a *ccNumber*, the string that holds the card's number.

You should implement all these as private instance variables.

The class should look like this:

CreditCard(expiryMonth, expiryYear, firstName, lastName, ccNumber)

Here's an example of initializing a new instance of this class:

```
cc1 = CreditCard(10, 2014, "Bob", "Jones", "1234567890123456");
```

Notice that we are passing the *expiryYear* argument as a four-digit number, and that *ccNumber* is a 16-character string.

formatExpiryDate(): Return the expiry date (month and year) in the format 'MM/YY'. For example, calling **formatExpiryDate()** on *cc1* should return the string "10/14". The month does not need to be zero-padded.

formatFullName(): Return the full name of the account holder in the format *'firstName lastName'*. For example, calling **formatFullName()** on *cc1* should return the string "Bob Jones".

formatCCNumber(): Return the credit card number as a four-block string, where blocks are separated by a single whitespace. For example, calling *formatCCNumber()* on *CC1* should return the string "1234 5678 9012 3456".

isValid(): Returns *true* if the expiry date of the credit card is later than the current value provided by the datetime utility. For example, calling **isValid()** on *cc1* would return *true* if evaluated at date November 2018 or any prior date.

__str__(): is a built-in python method. It is used to return a nicely formatted string containing name of the class and its memory address in hexadecimal. Override it so it returns a multi-line *string* which contains information such as the credit card number, the account holder, the expiration date and whether the card is valid or not. Here's an example of the expected output:

```
Number: 1234 5678 9012 3456 Expiry date: 10/14 Account
holder: Bob Jones Is valid: true
```

Write a **CreditCard** class which meets the conditions above.

In addition, we recommend that you write a code which creates some **CreditCard** objects (one valid and one invalid) and calls the methods of these objects.

5. DNA[1]

Our genetic makeup, and that of plants and animals, is stored in DNA. DNA is composed of two DNA-Strands, which twist together to form a double helix.

Each strand of DNA is a chain of smaller bases connected in a line. There are four bases:

[1] Adapted to python from http://www.inf.ed.ac.uk/teaching/courses/inf1/op/2018/labs/lab6q4.html

- adenine (abbreviated A)
- thymine (T)
- cytosine (C)
- guanine (G)

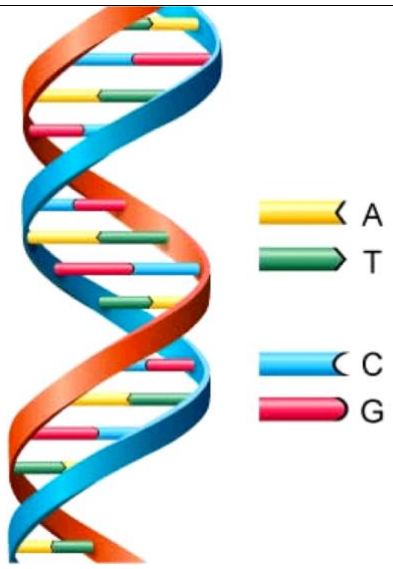

Figure 2: Source http://www.astrochem.org/sci_img/dna.jpg

The bases have pairs: A binds to T and C binds to G. The two strands 'match up', so that the bases at each point bind to their corresponding pairs. So for example, if one strand was

A-C-G-G-T-C

then the other strand would be:

T-G-C-C-A-G

Since we have the pairs A:T, C:G, G:C, G:C, T:A and C:G. This may sound like a redundant way of storing information, but it means that if we pull apart the strands of the double helix and throw one side away, we are still able to reconstruct it. Furthermore, if we want to duplicate our double helix, we can split it into the two strands, reconstruct each side, and then end up with two copies of the original double helix.

In this exercise, we will create a class that represents a strand of DNA.

Here how a **DNAStrand** instance is created:

DNAStrand(dna).

isValidDNA(): Returns true if the DNA is valid, i.e, only contains the letters, A,T,C,G (in uppercase) and at least one of these characters.

complementWC(): Returns the Watson Crick complement, which is a string representing the complementary DNA strand (i.e., the other strand in the double helix). So swap all T's with A's, all A's with T's, all C's with G's and all G's with C's.

palindromeWC(): Returns the Watson Crick Palindrome, which is the reversed sequence of the complement.

containsSequence(seq): Returns true if the DNA contains the subsequence seq.

__str__(): Override it so it returns the underlying DNA sequence string.

Write a **DNAStrand** class which meets the above requirements.

In addition, create a code which contains the following static method.

```
1. def summarise(dna):
2.     print("Original DNA Sequence: " , dna)
3.
4.     if dna.isValidDNA():
5.         print("Is valid")
6.         print("Complement: " , dna.complementWC())
7.         print("WC Palindrome: " , dna.palindromeWC())
8.     else:
9.         print("Not Valid DNA")
```

Create one or more DNAStrand objects and test them with the **summarise()** method.

End.