Exercise 4 – Python, Part II

1. Functions (20P)

In a file **`functions.py`**:

a. (5P) Define a function that accepts two strings as input, concatenates them and then prints out the result.
b. (5P) Define a function that accepts an integer number as input and prints out whether the number is even or odd.
c. (10P) Define a function to compute a division by zero and use try/except to catch the exceptions.

2. Classes and their methods (50P)

a. (10P) In a file **`persons.py`**: Define a class **`Person`** and its two child classes: **`Male`** and **`Female`**. All classes have a method **`get_gender()`** which should return their respective gender.
b. (40P) In cryptography, a Caesar cipher is a very simple encryption technique in which each letter in the plain text is replaced by a letter some fixed number of positions down the alphabet. For example, with a shift of **3**, **A** would be replaced by **D**, **B** would become **E**, and so on. The method is named after Julius Caesar, who used it to communicate with his generals. ROT-13 ("rotate by 13 places") is a widely used example of a Caesar cipher where the shift is 13.
In a file **`rot13.py`**: Implement an Class to encode and decode messages with ROT-13, and use it to decipher the message:

Vagebqhpvat FQAf

Note: In Python, the key for ROT-13 may be represented by means of the following dictionary:

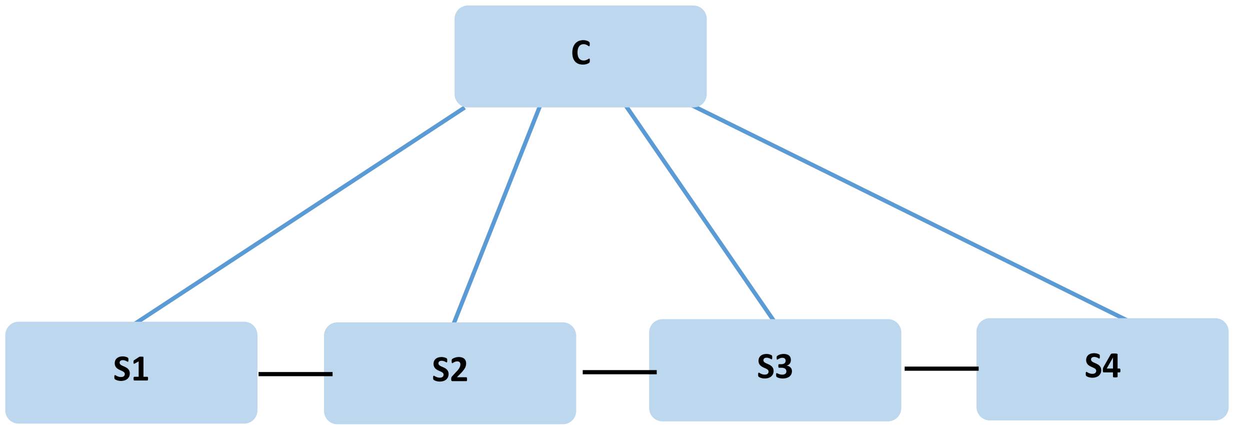
key = {'a':'n', 'b':'o', 'c':'p', 'd':'q', 'e':'r', 'f':'s', 'g':'t', 'h':'u', 'i':'v', 'j':'w', 'k':'x', 'l':'y', 'm':'z', 'n':'a', 'o':'b', 'p':'c', 'q':'d', 'r':'e', 's':'f', 't':'g', 'u':'h', 'v':'i', 'w':'j', 'x':'k', 'y':'l', 'z':'m', 'A':'N', 'B':'O', 'C':'P', 'D':'Q', 'E':'R', 'F':'S', 'G':'T', 'H':'U', 'I':'V', 'J':'W', 'K':'X', 'L':'Y', 'M':'Z', 'N':'A', 'O':'B', 'P':'C', 'Q':'D', 'R':'E', 'S':'F', 'T':'G', 'U':'H', 'V':'I', 'W':'J', 'X':'K', 'Y':'L', 'Z':'M'}

Generalize the coder so that it can provide methods to encrypt and decrypt the chifre defined by the following binary dictionary.

{'A': '11111', 'B': '11110', 'C': '11101', 'D': '11100', 'E': '110111', 'F': '11010',
'G': '11001', 'H': '11000', 'I': '10111', 'J': '10110', 'K': '10101', 'L': '10100',
'M': '10011', 'N': '10010', 'O': '10001', 'P': '10000', 'Q': '01111', 'R': '01110', 'S':
'01101', 'T': '01100', 'U': '01011', 'V': '01010', 'W': '01001', 'X': '01000',
'Y': '00111', 'Z': '00110',' ': '00000',"11111": "A", '11110': 'B', '11101': 'C',
'11100': 'D','11011': 'E', '11010': 'F', '11001': 'G', '11000': 'H', '10111': 'I',
'10110': 'J', '10101': 'K', '10100': 'L','10011': 'M', '10010': 'N', '10001': 'O',
'10000': 'P','01111': 'Q', '01110': 'R', '01101': 'S', '01100': 'T','01011': 'U',
'01010': 'V', '01001': 'W', '01000': 'X', '00111': 'Y', '00110': 'Z', '00000':' '}

3. A basic Software-defined Network simulator (100P)

(100P) Now that you know all the basics of Python, let's get back to the course content! In this exercise, you will build your very own (VERY basic) SDN simulator. At this point in the lecture you have learned that SDNs basically consist of two elements: switches and controllers. Suppose you have the following topology:



We call this topology, in which the four switches (S1-S4) are connected in a single line, a **linear** topology. The switches are orchestrated by a single controller C. Your task is to rebuild this topology and simulate the exchange of packets over that topology in a Python simulator. Consider the lifetime of the very first packet in the network that will be send from, for instance, S1 to S4:

The packet will be created at S1, and since it is the first packet, S1 will not have any flow tables installed. Instead, it will have to ask the controller C for a new rule. C has to install a new rule in S1. The rule should be: "Forward to S2". S2 has to ask C for a new rule as well. Note that in this case, the rule should be "If received from S1, forward to S3". Ultimately, the packet will arrive at S4, which should know that it is the destination of the packet.

Try to figure out how to build such a simulator in Python yourself first. After that, test your system with two flows:

- one flow of five packets originating at S1 and with S4 as destination
- a second flow consisting of three packets from S4 to S1
- you do not need to acknowledge the packets in this basic simulator

If you need help, there are some hints on the next page of this exercise.

4. Hints for the SDN Simulator (OP)

1. The entire simulator can be written in around 100 lines of code
2. You do NOT need to build a graphical interface. Rather, build a text simulator.
3. You do not need to import any additional python modules
4. The basic classes you will need are:

    - **Switch** – A switch will process a packet according to its **`flow_table`**:

    ```
    INPUT: Packet (including source switch)
    IF the switch is the destination of the packet:
        DO receive packet
    ELSE:
        IF the switch has a flow match entry in its flow_table:
            DO forward according to flow match
        ELSE:
            DO Forward to controller
    ```

    - **Controller** – The controller will setup the switches' **`flow_table`**s. Upon a switch request for an unknown flow, the controller takes the following actions:

    ```
    INPUT: Packet (including source switch)
    DO find appropriate next_hop
    DO update flow_table of switch
    ```

    - **Packet** – the object that is passed between switches and the controller

5. Instantiate these objects for the elements of your topology and use methods and attributes to connect them with each other.
6. Your controller can be very simple (it simply needs to let packets being forwarded from 'left to right' and 'right to left'). You can make use of the fact that each switch only has two neighbours at maximum.
7. The controller needs to decide the forwarding action for each switch. Each switch has a **`flow_table`** that should be modified by the controller. In your simulator, the modification can be done by the switch, but the controller has to make the decision.
8. At the beginning, the flow tables are empty at each switch.
9. Your code should also work if we extend the linear topology by any number of switches.