Advanced Programming in Python: Exercise 1

DAT 515, Chalmers University of Technology

Aarne Ranta, email aarne@chalmers.se

This is the first exercise session of the course. It consists of old exam questions from introduction courses.

You should try your hand with these questions before going to the exercise class - Lecture 1 should have brought you to the level where you can solve them!

The exercises are neither compulsory nor a part of grading - they are offered just to help your learning.

Enjoy programming in Python!

Question 1: Pandemic Rules

Norway has the following rules for persons entering the country from Sweden:

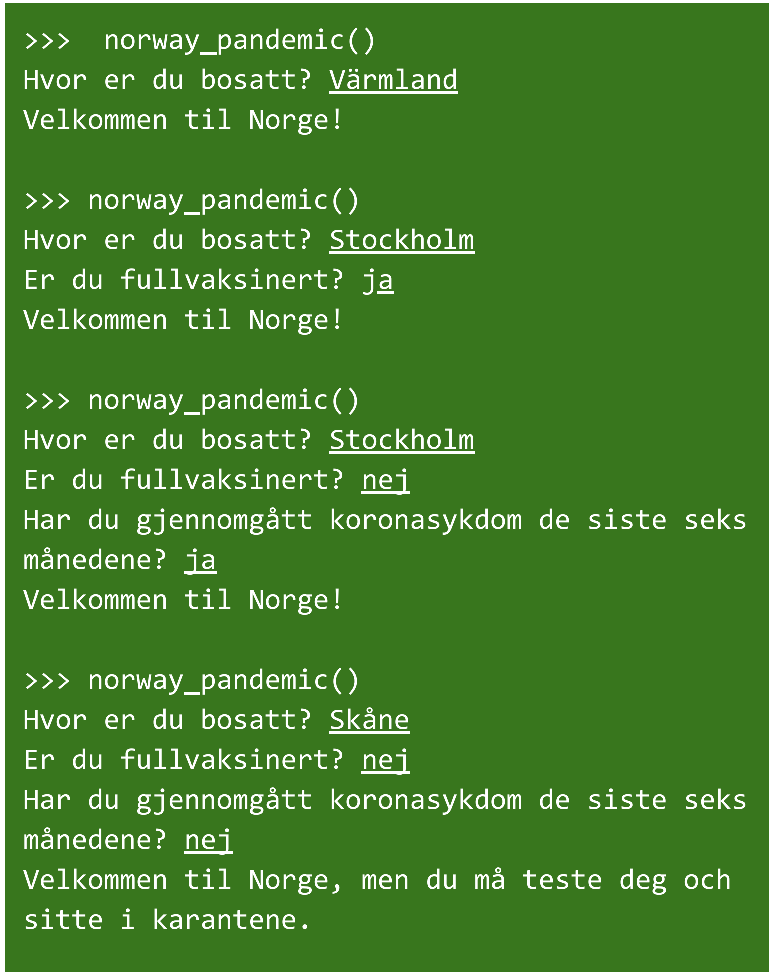
- you are welcome to Norway if you come from a "green" region, you can enter freely
- you are also welcome if you are fully vaccinated or have had Covid-19 during the last 6 months
- otherwise, you are still welcome but must get tested and stay in quarantine

(These rules are a simplified version from the official rules in https://www.udi.no/om-koronasituasjonen/innreise-og-opphold-jeg-er-utlandet/bosatt-i-eu-eos-schengen-eller-storbritannia/#link-24945 ; you don't need to read these :-)

Your task is to write a function norway_pandemic(), which implements these rules in a dialogue. It asks a series of three questions (in Norwegian of course!): region, vaccine, and earlier Covid. It expects answers as follows:

- the green regions are ones that start with a 'V' (Västra Götaland, Värmland, etc - another simplification)
- the yes/no questions interpret the answer 'ja' as positive, all others as negative
- **important: the program terminates as soon as it can be sure about the outcome, and does not pose any other questions then**

The screenshot on the right shows four different ways the dialogue can proceed in Python3. **Copy the system's questions and answers from the dialogue.** The user's answers are underlined.



```
>>>  norway_pandemic()
Hvor er du bosatt? Värmland
Velkommen til Norge!

>>> norway_pandemic()
Hvor er du bosatt? Stockholm
Er du fullvaksinert? ja
Velkommen til Norge!

>>> norway_pandemic()
Hvor er du bosatt? Stockholm
Er du fullvaksinert? nej
Har du gjennomgått koronasykdom de siste seks
månedene? ja
Velkommen til Norge!

>>> norway_pandemic()
Hvor er du bosatt? Skåne
Er du fullvaksinert? nej
Har du gjennomgått koronasykdom de siste seks
månedene? nej
Velkommen til Norge, men du må teste deg och
sitte i karantene.
```

Question 2: Ordering drinks

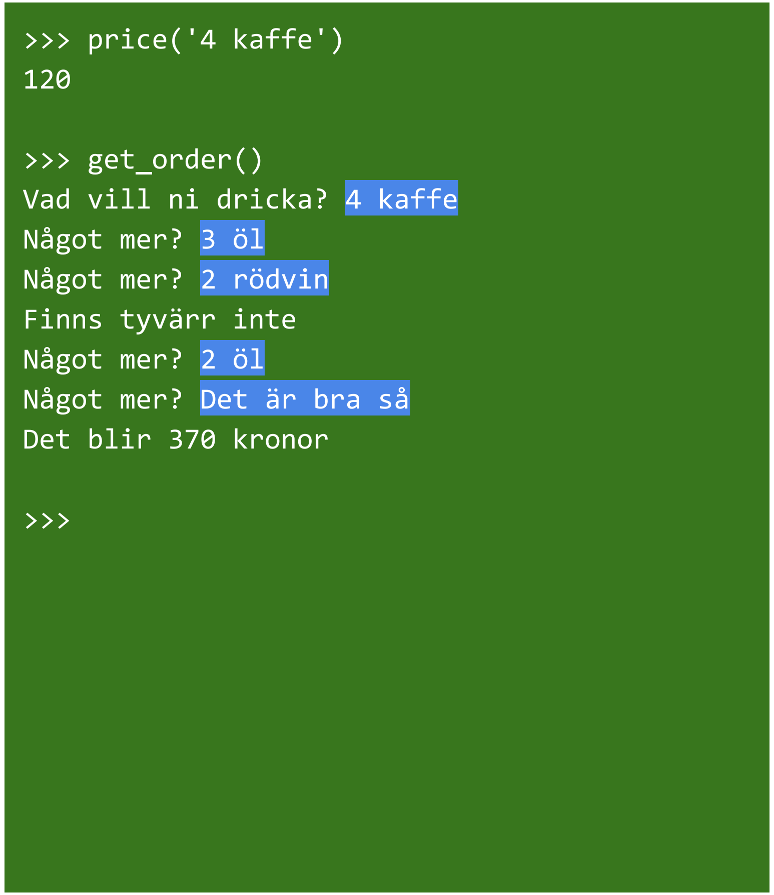
Write a function **price()**, which takes a string as argument and returns an integer. The string consists of a number and a name of a drink. The returned integer gives the price (in Swedish crowns) of that number of drinks, using the following price list:

- kaffe 30
- öl 50
- kola 25

Also write a function **get_order()**, which implements a dialogue between a bartender and a customer. The bartender starts by asking what the customer wants: *Vad vill ni dricka?* The customer answers by a number (in digits) and a drink name (in the exact form to which **price()** applies). If the drink is not on the list, the bartender apologizes that they don't have that drink: *Finns tyvärr inte*. After each input from the customer, the bartender asks if they want anything else: *Något mer?* This can be repeated any number of times, until the customer answers that it is enough: *Det är bra så.* The bartender then concludes the dialogue by stating the total price *X*: *Det blir X kronor.*

The screen dump shows examples of how these two functions behave. Please make sure that you get the same behaviour with your functions!

Notice: you don't need to take care of other kinds of input, which might cause errors. For instance, "ingenting" is an answer that need not be recognized. This if for simplicity: a real application should of course be more robust.



```
>>> price('4 kaffe')
120

>>> get_order()
Vad vill ni dricka? 4 kaffe
Något mer? 3 öl
Något mer? 2 rödvin
Finns tyvärr inte
Något mer? 2 öl
Något mer? Det är bra så
Det blir 370 kronor

>>>
```

Question 3. Scrambled text

Accroidng to a rseaecrehr at Cmarbdige Uinevsrtiy , it deons't mtaetr in waht odrer the lteetrs in a wrod are , the olny ipmroatnt tihng is taht the frist and lsat lteetr be at the rgiht palce .
(adapted from https://www.dictionary.com/e/typoglycemia/)

Your task is to implement a word scrambling algorithm that works as follows: for every word in a text, keep the first and the last letter as they are, but **alter** the order of the letters in between. The screen dump on the right shows how this works with several examples.

Altering here means that you swap the first letter with the second, the third with the fourth, and so on. If there are an odd number of letters, the last letter remains in its original place.

Your solution should consist of three functions:

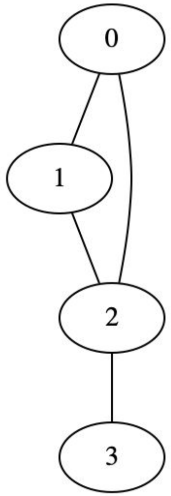
- **alter(s),** which swaps the letters of string s as described
- **scramble(s),** which keeps the first and last letters of string s and applies alter() to the letters between them
- **scrambles(s)**, which splits the string s into words and applies scramble() to each of them (you can assume that punctuation marks are separated by spaces, as in our example)

Bonus question: how do you unscramble a scrambled text, i.e. bring it back to the original? Do you need a new function for this?

```
>>> alter("ab")
'ba'
>>> alter("abcdef")
'badcfe'
>>> alter("abcde")
'badce'
>>> alter("a")
'a'
>>> scramble("Raby")
'Rbay'
>>> scramble("Rab")
'Rab'
>>> scramble("Rabely")
'Rbaley'
>>> scrambles("According to a researcher at Cambridge
University , it doesn't matter in what order the
letters in a word are , the only important thing is
that the first and last letter be at the right place
.")
"Accroidng to a rseaecrehr at Cmarbdige Uinevsrtiy ,
it deons't mtaetr in waht odrer the lteetrs in a wrod
are , the olny ipmroatnt tihng is taht the frist and
lsat lteetr be at the rgiht palce ."
```

Question 4. Graph representations

A **graph** is a mathematical object consisting of a set of **nodes** where some nodes are connected by **edges**. The picture on the right shows a graph with nodes 0, 1, 2, 3 and edges (0,1), (0,2), (1,2), (2,3).



In Python, graphs can be represented in several ways. Here are two:

- **list of edges**: pairs of nodes that are connected by an edge, e.g [(0,1), (0,2), (1,2), (2,3)] for the graph in the picture
- **adjacency list**: a dictionary that to each node assigns the list of edges that it is connected ("adjacent") to. For the graph in the picture, it is {0: [1, 2], 1: [0, 2], 2: [1, 0, 3], 3: [2]}

Your task is to define two functions:

- **edges2adjacency(edges)**, which converts a list of edges to an adjacency list. Make sure that an edge (*a*,*b*) adds to the dictionary entry of both *a* and *b*.
- **adjacency2edges(adj)**, which converts an adjacency list to a list of edges. Make sure that the list has no duplicates: if (*a*,*b*) is in it, then (*b*,*a*) should not be. The order of the resulting list does not matter: the important thing is that it contains every edge, exactly ones.

Make sure that your functions behave like the examples in the screen dump! But you should test with other examples as well.

Note to those familiar with graphs: this question deals with the special case of ***undirected graphs*** *without nodes that have no edges.*

```
>>> edges = [(0,1), (1,2), (2,0), (2,3)]

>>> adj = edges2adjacency(edges)

>>> adj
{0: [1, 2], 1: [0, 2], 2: [1, 0, 3], 3: [2]}

>>> adjacency2edges(adj)
[(0, 1), (0, 2), (1, 2), (2, 3)]
```