Introduction in Python — Part 2

Data types II

Exercise 1 In German the most often used letters are E, N, I, S, R, A, T, D, H, U. Write a function to test if a given string only consist of these characters.

1.1 Is there a way to make the function independent of lower/upper case?

1.2 Is there a way to get the characters which are not part of the most often used letters?

Exercise 2 Write a program which counts the appearances of all characters in a text file.

- To optimize the memory usage read the file line by line.
- Print all appearing characters and their count.

Exercise 3 (Morse code) Write a function which will translate a given string into Morse code. The translation should ignore upper/lower case.

A	· –	J	· – – –	S	· · ·
B	– · · ·	K	– · –	T	–
C	– · – ·	L	· – · ·	U	· · –
D	– · ·	M	– –	V	· · · –
E	·	N	– ·	W	· – –
F	· · – ·	O	– – –	X	– · · –
G	– – ·	P	· – – ·	Y	– · – –
H	· · · ·	Q	– – · –	Z	– – · ·
I	· ·	R	· – ·		

Object-oriented programming

Exercise 4 (Point class)

4.1 Implement the class *Point* as described in the lecture. If a *Point* object is printed (using `str` or `print` function) the output should be readable. Two *Point* objects should be equal if their x and y member field are the same.

4.2 Add the methods to add (+) and multiply ($*$) two objects of class *Point*:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \end{pmatrix}, \qquad \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} * \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = (x_1 * x_2) + (y_1 * y_2)$$

The following should work:

```
>>> print(Point(1, 2) + Point(4, 2))
(5, 4)
>>> print(Point(1, 2) * Point(4, 2))
8
```

4.3 Implement the norm of a point using the already defined scalar product:

$$\|p\| = \sqrt{p * p}$$

Exercise 5 (Banking account)

5.1 Implement a class *Account* with the attributes *account number*, *balance* and *account holder*.

It is not allowed to overdraw the account. Try to ensure that accessing the balance will not result in a negative balance.

5.2 All accounts have the same *interest rate*. Implement a method in class *Account* to get the actual interest rate and another method to get the interest for the balance of the actual *Account* object.

Modules and Packages

Exercise 6 (Password generator) Write a program to create a password. The password length should be 8 characters.
Hint: Take a look at `string.ascii_letters` and `string.digits`!

Exercise 7 Write a program which will get a *directory name* as command line argument. All files in this directory which ends with `htm` should be renamed, so that the new ending is `html`.

Exercise 8 (CSV) The following text file is given (create it):

```
"Mr. Spock","Vulcan"
"Freddie ""Nightmare"" Krueger, Jr.","Elm Street 42,
Springfield"
```

(The line break in the second row is intentionally!). The two data sets contain two fields (*name* and *address*). Read in all data and print them out on screen.

Hint: What makes such data sets difficult to parse? (Try it on your own.)

Exercise 9 (Create your own module) Write a script which uses your *Point Class* as a module. Modify your *Point Class* the way that no top level statement will be executed on an *import*. Therefore implement the feature mentioned in the slides to test modules if they run as an ordinary script.

Exercise 10 (XML-RPC) There is a XML-RPC-Server running with the address `http://vsm1.zam.kfa−juelich.de`:8000. Try to connect to the server by writing a XML-RPC-Client and use the methods `listMethods` and methodHelp to find out, which methods the server is offering and try to use them.

Bonus exercises

Exercise 11 (UNO/Mau-Mau) Implement the game *UNO* without any special rules for certain cards. Try this approach:

11.1 (Basics) Proposal for the class structure:

- Class `Card` with the attributes `color` and `value` and a method `compatible(self, other)`, which checks, if two cards can be put on each other.
- Class `DeckUNO` for a shuffled card deck which is based on the class **list**:

```
class DeckUNO(list):
    VALUES = [”0”, ”1”, ”2”, ”3”, ”4”, ”5”, ”6”, ”7”, ”8”, ”9”]
    COLORS = [”red”, ”yellow”, ”green”, ”blue”]

    def __init__(self):
        cards = []

        for color in self.COLORS:
            for value in self.VALUES:
                cards.append(Card(color, value))

        random.shuffle(cards)

        list.__init__(self, cards)
```

- Class `Player` where the init method will get a *DeckUNO* object. Six cards will be withdrawn from the deck and saved as the players hand cards. Additional attributes (player name) can be defined. In addition the player needs a method to play a card. This method can vary for different types of players (human and computer). Therefore the specific player classes can inherit from a general *Player* class.

The program should run until a player has won or there is no card left in the card deck (*Talon*). Therefore exception can be used which inherit from the general `Exception` class:

```
class WinException(Exception):
    pass

class TalonEmptyException(Exception):
    pass
```

11.2 (Special cards) Implement the special UNO cards *Skip*, *Draw Two*, *Reverse*, *Wild* and *Wild Draw Four.*

11.3 (For experts) The sorted card deck can be easily created by using list comprehension or the `itertools` module.

Exercise 12 (For experts) It was explained in the lecture how classes can be used to create arbitrary data structures. Improve this principle and write a class `Bunch`, whose `init` method gets arbitrary keyword parameters and saved these as attributes:

```
point = Bunch(x=2, y=3)
print(point.x, point.y)

person = Bunch(forename="Homer", surename="Simpson", phone=123456)
print(person.forename, person.surename, person.phone)
```

Not enough? More exercises:

http://www.pythonchallenge.com