Assignment #10: Recursion

Master in Informatics and Computing Engineering
Programming Fundamentals
Instance: 2018/2019

***Goals**: write programs using a recursion*

***Pre-requirements (prior knowledge)**: See bibliography of Lecture #17 an Lecture #18*

***Rules**: you may work with colleagues, however, each student must write and submit in Moodle his or her this assignment separately. Be sure to indicate with whom you have worked. We may run tools to detect plagiarism (e.g. duplicated code submitted)*

***Deadline**: 8:00 Monday of the week after (10/12/2018)*

***Submission**: to submit, first pack your files in a folder RE10, then compress it with zip to a file with name `2018xxxxx.zip` (your_code.zip) and last (before the deadline) go to the Moodle activity (**you have only 2 attempts**)*

1. Flatten

Given a nested list `alist` (i.e., a list which may contain lists which themselves may contain other lists, and so on), write a recursive Python function `flatten(alist)` that returns a single list with each of the non-list elements, in order of occurrence. This process is known as list flattening.

Save the program in the file `flatten.py`

For example:

- `flatten(['Hello', [2, [[], False]], [True]])` returns the list: `['Hello', 2, False, True]`
- `flatten([[]])` returns the list: `[]`

2. Calculator

Write a function `calculator(expr)` that given an expression `expr` computes its value. The expression `expr` is a nested tuple composed of (tuple/integer, operator, tuple/integer) or an integer. The valid operators are: addition (+), subtraction (-) and multiplication (*).

Save the program in the file `calculator.py`

For example:

- `calculator((1, '+', 2))` return the integer `3`
- `calculator(((1, '+', 2), '*', 3))` return the integer `9`

3. File Finder

Let `dirs` be a recursively-defined nested list, representing a directory tree containing an arbitrary number of directories, sub-directories and files.

Consider the following example:

```
dirs = ["home",
          ["Documents",
             [ "FPRO", "lists.txt", "recursion.pdf", "functions" ],
             [ "Python", "hello_world.py", "readme.md" ],
          ],
          ["Downloads",
             [ "Movies",
                [ "TV Series", "BreakingBad.mp4", "TheBigBangTheory.avi" ],
                "1.avi", "22", "001.mp4"
             ],
          ],
          "tmp.txt", "page.html"
       ]
```

In the above example, each directory is defined by a list, and the first element of the list is the name of that directory (home, Documents, FPRO, etc). The rest of the elements of a list contain either strings, which are filenames inside that directory, or lists, which contain a sub-directory.

Write a Python function `file_finder(dirs, file_name)` that returns the full path of a file `file_name` (given as a string), or `None` if it is not in the directory tree `dirs`. The full path of a file includes the slash-separated names of all the directories that contain it. Therefore, the full path of "BreakingBad.mp4" is "home/Downloads/Movies/TV Series/BreakingBad.mp4".

Save the program in the file `file_finder.py`

For example:

- `file_finder(dirs, 'Documents')` returns: `None` (because Documents is a sub-directory not a file)
- `file_finder(dirs, 'recursion.pdf')` returns the string: `"home/Documents/FPRO/recursion.pdf"`

4. Crazy letter soup

Write a function **`soup(matrix, word)`** that, given a **`matrix`** of letters, returns the first location of the **`word`** in the matrix.

For example, let's say we have the following matrix and are trying to find the word **`"PORTO"`**.

	1	**2**	**3**	**4**	**5**	**6**
A	X	A	B	N	T	O
B	Y	T	N	R	I	T
C	U	P	O	M	D	S
D	I	O	H	U	O	O
E	R	T	E	L	Q	X
F	I	W	J	K	P	Z

Then the function returns **`"C2"`** because the word starts in line=**C**, column=**2**.

Notice that the words can be in any direction: northwest, north, northeast, east, southeast, south, southwest or west.

All letters are given in upper-case and the function returns the first occurrence of the word using lexicographical order (i.e. if the word can be found in "A4" and "B2", then it returns "A4"). You may want to use **`chr()`** and **`ord()`** to have the line as a letter.

Save the program in the file **`soup.py`**

For example:

- ```
  mx = (('X', 'A', 'B', 'N', 'T', 'O'),
        ('Y', 'T', 'N', 'R', 'I', 'T'),
        ('U', 'P', 'O', 'M', 'D', 'S'),
        ('I', 'O', 'H', 'U', 'O', 'O'),
        ('R', 'T', 'E', 'L', 'Q', 'X'),
        ('I', 'W', 'J', 'K', 'P', 'Z'))
  ```
- `soup(mx, 'PORTO'))` returns the string **`"C2"`** (as illustrated above)
- `soup(mx, 'HOTEL')` returns the string **`"D3"`**
- `soup(mx, 'RIO')` returns the string **`"B4"`**
  ```

5. Budget version 2.0

In RE09, the budget was not enough to pay for entire wishlist and the proposed algorithm was to remove the cheapest items first until *cost* ≤ *budget*. This, however, wasn't the optimal solution. Imagine that my budget is 1000 and I want to buy 2 laptops (each costs 2000) and 3 jeans (50). The previous algorithm would decide not to buy anything, not even the jeans, because it would end up removing everything from the wishlist.

We may use another approach: the knapsack problem (*problema da mochila*) is a famous problem from computer science. We have a fixed budget and we want to buy as much as we can to fit the budget. In this version of the knapsack problem, there is a benefit proportional to the cost as long as the product is in the wishlist.

Write a Python function **`budgeting2(budget, products, wishlist)`** that receives an integer indicating the budget, a **`products`** dictionary showing the price of each item, and a **`wishlist`** dictionary indicating how much quantity we want of each product. Your goal is to spend as much of your budget as you can, without going over the budget.

Save the program in a file called **`budgeting2.py`**

For example:

- **`budgeting2`**`(1000, {'ps4': 350, 'smartphone': 400, 'jacket': 40, 'pc': 600, 'glasses': 75}, {'ps4': 1, 'smartphone': 1, 'pc': 1})` returns the dictionary: `{‘pc’: 1, 'smartphone': 1}`
- **`budgeting2`**`(1500, {'xbox': 250, 'smartphone': 500, 'jeans': 50, 'pc': 600, 'glasses': 100}, {'glasses': 3, 'jeans': 2, 'pc': 1, 'xbox': 1})` returns the dictionary: `{'glasses': 3, 'jeans': 2, 'pc': 1, 'xbox': 1}`
- **`budgeting2`**`(1000, {'laptop': 2000, 'jeans': 50}, {'laptop': 2, 'jeans': 3})` returns the dictionary: `{'jeans': 3}`

The end.

FPRO, 2018/19