PE02: 02/12/2020

Master in Informatics and Computing Engineering
Programming Fundamentals
Instance: 2020/2021

An example of solutions for the 5 questions in this Practical on computer evaluation.

1. Check Armstrong number

Write a Python function `is_armstrong(n)` that checks if a number `n`, with three digits, is an Armstrong number or not. In an Armstrong number of three digits, the sum of the cubes of each digit is equal to the number itself.

Solution:

```
def is_armstrong(n):
    sum = 0
    # find the sum of the cube of each digit using the accumulator sum
    aux = n
    while aux > 0:
        digit = aux % 10
        sum += digit ** 3
        aux //= 10
    return sum == n
```

2. Reverse sub-tuples

Write a Python function `reverse_subtuples(alist)` that receives a list `alist`, which contains tuples, and returns a list with all its tuples reversed.

For example:

- `alist=[(1, 2, 3), (4, 5, 6)]` => `[(3, 2, 1), (6, 5, 4)]`

Solution:

```
def reverse_subtuples(alist):
    result = []
    # reverse each tuple of alist using an accumulator
    for t in alist:
        rt = ()
        for i in t:
            rt = (i, ) + rt
        result.append(rt)
    return result

# alternate solution
def reverse_subtuples(alist):
    result = []
    for t in alist:
        result.append(tuple(reversed(t)))
        # result.append(t[::-1])
    return result
```

3. Which are in?

Write a Python function `which_are_in(l1, l2)` that, given two lists of strings **l1** and **l2** returns a sorted list in lexicographical order of the strings of **l1** which are substrings of strings of **l2**.

For example:

- l1 = ["arp", "live", "strong"], l2 = ["lively", "alive", "harp", "sharp", "armstrong"] => ["arp", "live", "strong"]
- l1 = ["tarp", "mice", "bull"], l2 = ["lively", "alive", "harp", "sharp", "armstrong"] => []

Solution:

```python
def which_are_in(l1, l2):
    res = []
    # for all combinations of list 1 and list 2
    for si in l1:
        for sj in l2:
            # check if si is a sub-string of sj, not yet in the result
            if si in sj and si not in res:
                res.append(si)
    return sorted(res)
```

4. Moving Average

Write a Python function `moving_average(alist, n)` that, given a list `alist` of integers and an odd integer **n** (representing the neighborhood span) returns a list with the averages of the neighborhood for each number, rounded to two decimal places. Note that each neighborhood span has exactly **n** elements.

If `alist` has length < 3 or if **n** < 3, then the result is the empty list.

For example:

- alist = [1, 2, 3, 4, 5], n = 3 => `[(1+3)/2, (2+4)/2, (3+5)/2]` => [2.0, 3.0, 4.0]

Solution:

```
def moving_average(alist, n):
    # solve the restriction first
    if len(alist) < 3 or n < 3:
        return []
    avgs = []
    # go through all indexes i with a neighborhood span (with n elements)
    for i in range(0+n//2, len(alist)-n//2):
        # find the left and right neighbors of index i
        left = alist[i-n//2:i]
        right = alist[i+1:i+n//2+1]
        neighbors = left + right
        # append the current average round to 2
        avgs.append(round(sum(neighbors)/len(neighbors), 2))
    return avgs
```

5. Genealogy by group and order

The genealogy of somebody's family may be written in two forms:

- **Form 1**: `[("Ana", "sibling"), ("Carlos", "parent"), ("Diana", "parent")]`
- **Form 2**: `[(["Ana"], "sibling"), (["Carlos", "Diana"], "parent")]`

Write a Python function called `genealogy(l1)` that receives a genealogy list `l1` written in **Form 1** and returns a list written in **Form 2**.

The returned list must be ordered by relationship using the following rule: *sibling* < *parent* < *cousin* < *grandparent*, and each sub-list must be ordered by *name*.

Solution:

```
def f_order(elem):
    """ function to compare items """
    name, relation = elem
    order = ("sibling", "parent", "cousin", "grandparent").index(relation)
    return order, name

def genealogy(l1):
    result = []
    # sort l1 using key
    l1 = sorted(l1, key=f_order)
    # traverse the sorted list of tuples
    for t in l1:
      if result == [] or t[1] != result[-1][1]:
        # the first tuple or a new relationship
        result.append(([t[0]], t[1]))
      else:
        # append to an existing relationship
        result[-1][0].append(t[0])
    return result
```

The end.