PE05: PE of 08/01/2020 (solutions)

Master in Informatics and Computing Engineering
Programming Fundamentals
Instance: 2019/2020

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

1. Clean phrase

Write a Python function `clean_phrase(astr)` that receives a string of lowercase words separated by whitespace as `astr` and returns the words after removing all duplicate words and sorting them in lexicographical order.

```
def clean_phrase(astr):
    words = astr.split(" ")
    words_uniq = list(set(words))
    words_sort = sorted(words_uniq)
    phrase = " ".join(words_sort)
    return phrase

# another more compact solution
def clean_phrase(astr):
    return " ".join(sorted(list(set(astr.split(" ")))))
```

2. Clean list

Write a Python function `clean_list(alist, adict)` that receives a list of integers as `alist` and a dictionary where the keys are strings and the values are lists of integers as `adict`. The function returns a set with the dictionary's keys which have in the corresponding value any of the integers of `alist`.

For example:

- If `alist` = `[1, 2, 3]` and `adict` = `{'a': [1, 2], 'b': [4, 5]}` then `clean_list(alist, adict)` returns the set `{'a'}`.

```
def clean_list(alist, adict):
    rset = set()
    for k, lv in adict.items():
        for v in lv:
            if v in alist:
                rset.add(k)
    return rset

# another more pythonic solution
def clean_list(alist, adict):
    return {k for k, lv in adict.items() if any(v in alist for v in lv)}
```

3. Zip product

Write a Python function `prod_zip(functions, arguments)` that, given a list of `functions` and a list of `arguments` of the same size, returns the product of each function applied to each argument of the same index.

For example:

-
  ```
  prod_zip([lambda x: x*2,
             lambda x: x**2,
             lambda x: -x],
            [-5, 10, 3])
  ```
  returns `3000` because `-5*2 * 10**2 * (-3)` = `3000`

You **cannot** use cycles, only map/filter/reduce or list comprehensions.

```
import functools

def prod_zip(functions, arguments):
    parcels = [f(a) for f, a in zip(functions, arguments)]
    return functools.reduce(lambda a, b: a*b, parcels, 1)
```

4. Aliquot generator

Write a generator function `aliquot(n)` that iteratively *yields* the next term in the Aliquot sequence, starting in `n`.

Aliquot is a sequence of integers in which each term is the sum of the proper divisors of the previous term. Proper divisors are divisors of a number excluding the number itself.

For example, if `n` is the number 33:

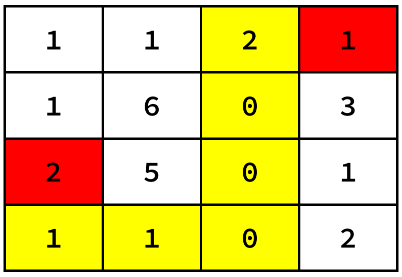
- the first term is 33
- 33 can be properly divided by 1, 3, and 11, therefore the next term is 1+3+11=15
- 15 can be properly divided by 1, 3 and 5, therefore the next term is 1+3+5=9
- 9 can be properly divided by 1 and 3, therefore the next term is 1+3=4
- 4 can be properly divided by 1 and 2, therefore the next term is 3
- 3 can be properly divided by 1, therefore the next term is 1
- 1 has no proper divisors, therefore the final term is 0.

```
def aliquot(n):
    yield n
    while n > 0:
        n = sum(i for i in range(1, n) if n % i == 0)
        yield n
```

5. Minimum cost path

Write a function `min_cost_path(matrix, a, b, visited=[])` that discovers the lowest cost path between the positions **a** and **b** inside the maze `matrix`. You can use the `visited` list of positions to avoid walking through the same path twice. Positions **a** and **b** are tuples (row, column) and `matrix` is a cost matrix of non-negative integers indicating the cost of that tile in the maze. Valid movements include all 4 adjacent tiles (up, down, left, right).

For example, for the maze of the following figure, a minimum cost path between the two points in red is the yellow path with `cost=2+1+1+0+0+0+2+1=7`.



```python
IMPOSSIBLE = 999999999

def min_cost_path(matrix, a, b, visited=[]):
    if a[0] < 0 or a[0] >= len(matrix):     # outside matrix lines
        return IMPOSSIBLE
    if a[1] < 0 or a[1] >= len(matrix[0]):  # outside matrix columns
        return IMPOSSIBLE
    if a in visited:                        # already visited
        return IMPOSSIBLE

    c = matrix[a[0]][a[1]]
    if a == b:                              # final position
        return c

    dists = [
        c + min_cost_path(matrix, (a[0]+1, a[1]), b, visited+[a]),
        c + min_cost_path(matrix, (a[0]-1, a[1]), b, visited+[a]),
        c + min_cost_path(matrix, (a[0], a[1]+1), b, visited+[a]),
        c + min_cost_path(matrix, (a[0], a[1]-1), b, visited+[a]),
    ]
    return min(dists)
```

The end.

FPRO, 2019/20

==

xx. Closest pair of points [fjcm] [?]

Write a Python function that given 2 list of points with x and respective y coordinates, produce a minimal distance between a pair of 2 points

For example,

- ...

```
def closest_split_pair(p_x, p_y, delta, best_pair):
    ln_x = len(p_x)  # store length - quicker
    mx_x = p_x[ln_x // 2][0]  # select midpoint on x-sorted array
    # Create a subarray of points not further than delta from
    # midpoint on x-sorted array
    s_y = [x for x in p_y if mx_x - delta <= x[0] <= mx_x + delta]
    best = delta  # assign best value to delta
    ln_y = len(s_y)  # store length of subarray for quickness
    for i in range(ln_y - 1):
        for j in range(i+1, min(i + 7, ln_y)):
            p, q = s_y[i], s_y[j]
            dst = dist(p, q)
            if dst < best:
                best_pair = p, q
                best = dst
    return best_pair[0], best_pair[1], best
```

x. Agenda [rpc] [easy] [?]

Write a Python function `agenda(alist, adict)` that given a list of names in `alist` returns the respective phone numbers by searching for them in the dictionary in `adict`.

For example:

- [['Andre', 'Maria', 'Eduardo'], {'Andre': 93182713, 'Eduardo': 9182782762, 'Maria': 2219372781} => [93182713, 2219372781, 9182782762]

Tests:

1. inputs: [x] - output: "y"'

Solution:

```
def agenda(alist, adict):
    return [adict[name] for name in alist]
```

x. Iterate lists by reverse order [rpc]

Write a Python program that has **three** lists of equal size referenced by variables `names` (strings), phone `numbers` (integers) and `emails` (strings). The program prints each name/number/email in each line, in reverse order.

For example:

- if `names = ["anabart", "bernandomarie", "jocarlos"]` and `numbers =`

[938421028, 916381961, 939090082], emails=["ana.serra@gmail.com", "b1999@hotmail.com", "up201945321@fe.up.pt"], the output is "ana 938421028 ana.serra@gmail.com\nbernardo 916381961 b1999@hotmail.com\ncarlos 939090082 up201945321@fe.up.pt"

Tests:

2. inputs: [x] - output: "y"'

Solution:

```
names = ["anabart", "bernardomarie", "carlosjo"]
numbers = [938421028, 916381961, 939090082]
emails=["ana.serra@gmail.com", "b1999@hotmail.com", "up201945321@fe.up.pt"]

for i in range(len(names)-1, -1, -1):
    print(names[i], numbers[i], emails[i])
```

x. Process commands [rpc]

Define a list of integers called **commands** and a list of floats called **parameters**. For example:

```
commands = [1, 3, 2, -4, 2]
parameters = [10, -2, 0.05]
```

Start with a **result = 0**. Each element in **commands** represent a different operation that you must perform to the **result** using the **parameters**:

command	operation	description
1	sum	sum each parameter to the result
2	subtraction	subtract each parameter from the result
3	multiplication	multiply each parameter to the result
otherwise	ignore	

Print the result using **round** with two decimal places.

Tests:

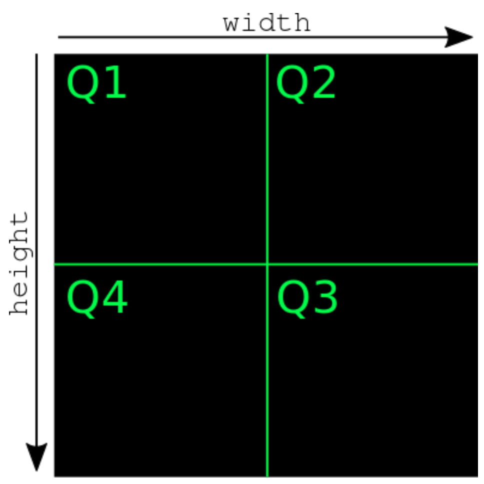
3. inputs: [x] - output: "y"'

Solution:

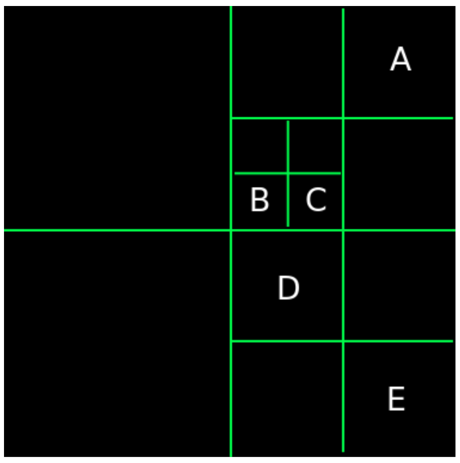
```
result = 0
for c in commands:
    for p in parameters:
        if c == 1: result += p
        if c == 2: result -= p
        if c == 3: result *= p
print(round(result, 2))
```

5. Search quadrants [rpc] [hard?] [PE05?]

In a MMORPG game, for efficiency reasons, the map is divided into quadrants: Q1, Q2, Q3 and Q4. The map has a certain *width* and *height*, and the quadrants divide the map evenly.



Each quadrant is itself divided into a quadrant, *ad infinitum*, until a player can be found ('A', 'B', ...):



Write a Python function `search_quadrants(quadrant, size, rectangle)` that receives the following arguments:

1. a `quadrant` which is represented by a tuple `(q1, q2, q3, q4)`, where each quadrant is a function that, when called, produces either (i) a sub-quadrant or (ii) a string with the player's name or (iii) `None` if there are no objects. The function will generate an error if you ask for quadrants outside the search space.
2. a `size` tuple containing `(width, height)` of the quadrant in real units.
3. a `rectangle` tuple containing the rectangular search space `(x1, y1, x2, y2)` in real units.

The function should return a set containing all players' names whose sub-quadrants intersect the search space.

Solution:

```
def search_quadrants(quadrant, size, rectangle):

    # Base cases
    if quadrant is None:
        return set()
    if type(quadrant) == str:
        return {quadrant}

    # Recursive search
    q1, q2, q3, q4 = quadrant    # Functions that give the quadrant
    width, height = size         # Quadrant size
    x1, y1, x2, y2 = rectangle   # Search space
    res = set()
    # Q1
    if x1 < width/2 and y1 < height/2:
        res |= search_quadrants(q1(), (width/2, height/2), rectangle)
    # Q2
    if x2 > width/2 and y1 < height/2:
        res |= search_quadrants(q2(), (width/2, height/2), (x1-width/2, y1,
x2-width/2, y2))
    # Q3
    if x2 > width/2 and y2 > height/2:
        res |= search_quadrants(q3(), (width/2, height/2), (x1-width/2,
y1-height/2, x2-width/2, y2-height/2))
    # Q4
    if x1 < width/2 and y2 > height/2:
        res |= search_quadrants(q4(), (width/2, height/2), (x1, y1-height/2,
x2, y2-height/2))

    return res
```