PE3: PE of 21/12/2018

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

Here you have a possible solution for each question of the Practical on computer Evaluation.

1. Treasure

Write a Python function **`treasure(clues)`** that receives a dictionary of **`clues`** where each key is a location and each value is a clue of what is the next location to go. Start at (0,0) and return the tuple of the final location you end up with. Note that clues may be used only once.

For example, if **`clues={(0,0): (1,0), (2,1): (3,5), (1,0): (2,1)}`** then you should first go to (0,0) then (1,0) then (2,1) then (3,5) and then finish because there's no clue in that location. In this case, the function returns (3,5).

Save the program in the file **`treasure.py`** inside the folder **`PE3`**.

For example:

- **`treasure({(0,0): (1,0), (1,0): (2,0), (2,0): (3,0)})`** returns the tuple **`(3,0)`**
- **`treasure({(0,0): (1,0), (2,1): (3,5), (1,0): (2,1)})`** returns the tuple **`(3,5)`**
- **`treasure({(0,0): (5,6), (7,8): (6,7), (5,6): (6,7), (6,7): (7,8)})`** returns the tuple **`(6,7)`**

Solution:

```
def treasure(clues):
    pos = (0, 0)                        # initial location
    while pos in clues:
        oldpos, pos = pos, clues[pos]  # location ("pos") moves to next clue
        del clues[oldpos]               # delete old location
    return pos
```

2. Polynomials

Suppose we have a polynomial **a** represented as a list of coefficients, a[0], a[1], ..., a[n-1], where a[i] is the coefficient of xi; that is:

$$f(x) = a_0x^0 + a_1x^1 + ... + a_nx^n$$

Write a function **`evaluate(a, x)`** using list comprehensions (or *map* and *reduce*) that evaluates the value of the polynomial for a given integer **`x`**.

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

For example:

- **`evaluate([1, 2, 4], 5)`** returns the integer **`111`**
- **`evaluate([1, 2, 4], 10)`** returns the integer **`421`**
- **`evaluate([1, 2, 4, 6, 8], 2)`** returns the integer **`197`**

Solution:

```
def evaluate(a, x):
    xi = [x**e for e in range(0, len(a))]           # [x^0, x^1, x^2, ...
    axi = [a[i] * xi[i] for i in range(0, len(a))]  # [a[0]*x^0, a[1]*x^1, ...
    return sum(axi)

# another one-line solution
def evaluate(a, x):
    return sum(ai*x**e for e, ai in enumerate(a))
```

3. Recursive dot product

Write a function `recursive_dot(l1, l2)` that computes the inner dot product using the two lists provided — the two lists follow the same structure. For example, if `l1=[1, [2, 3]]` and `l2=[4, [5, 6]]`, then the result will be 1*4+(2*5+3*6).

Save the program in the file `recursive_dot.py` inside the folder `PE3`.

You might want to use `type(x)` to find the type of `x`.

For example:

- `recursive_dot([1, [2, 3]], [4, [5, 6]])` returns the integer `32`
- `recursive_dot([[5, 3, 1], [2, 4]], [[4, 2, 0], [1, 3]])` returns the integer `40`
- `recursive_dot([2], [1])` returns the integer `2`

Solution:

```
# recursive solution
def recursive_dot(l1, l2):
    if type(l1) == int:
        return l1*l2
    if len(l1) == 0:
        return 0
    return recursive_dot(l1[0], l2[0]) + recursive_dot(l1[1:], l2[1:])

# another solution with for
def recursive_dot2(l1, l2):
    if type(l1) == int:  # base condition
        return l1*l2
    sum = 0
    for i, j in zip(l1, l2):
        sum += recursive_dot2(i, j)
    return sum
```

4. Interleave Lists

Write a Python function **`interleave(alist1, alist2)`** that, given two lists with the same structure but not necessarily the same length, **`alist1`** and **`alist2`**, which may contain other lists, returns a list with the interleaved elements of both **`alist1`** and **`alist2`**. ~~The result list has the length of the smaller of the two lists.~~

Save the program in the file **`interleave.py`** inside the folder **`PE3`**.

For example:

- **`interleave`**`([1, [4,2]], [3, [7,4]])` returns the list `[1,3,4,7,2,4]`
- **`interleave`**`(['a','b','c'], [1,2,3,4,5])` returns the list `['a',1,'b',2,'c',3]`
- **`interleave`**`([], [1,2])` returns the list `[]`

Solution:

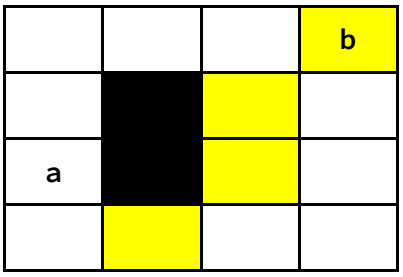
```
# recursive solution
def interleave(l1, l2):
    if type(l1) != list:
        return [l1, l2]
    if len(l1) == 0 or len(l2) == 0:
        return []
    return interleave(l1[0], l2[0]) + interleave(l1[1:], l2[1:])

# another solution with for
def interleave2(l1, l2):
    if type(l1) != list: # base condition
        return [l1, l2]
    res = []
    for i, j in zip(l1, l2):
        res += interleave2(i, j)
    return res
```

5. Minimum path

Write a function **`min_path(matrix, a, b, visited=[])`** that discovers the minimum path between **`a`** and **`b`** inside the **`matrix`** maze without going through **`visited`** twice. Positions **`a`** and **`b`** are tuples (line, column), **`matrix`** is a matrix of booleans with **`False`** indicating no obstacle and **`True`** indicating an obstacle and **`visited`** is a list of visited tuples. Valid movements include all 8 adjacent tiles.

For the maze of the following figure, a minimum path between **`a`** and **`b`**, in yellow, is **4**:



Save the program in the file **`min_path.py`** inside the folder **`PE3`**.

For example:

- ```
  mx = [
    [False]*4,
    [False, True, False, False],
    [False, True, False, False],
    [False]*4
  ]
  ```

- `min_path(mx, (2, 0), (0, 3))` returns the integer `4`
- `min_path(mx, (3, 1), (0, 1))` returns the integer `3`
- `min_path(mx, (0, 0), (3, 3))` returns the integer `4`

Solution:
  ```

```
IMPOSSIBLE = 999999

def min_path(matrix, a, b, visited=[]):
    if a == b:                              # final position
        return 0
    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 matrix[a[0]][a[1]]:                  # an obstacle
        return IMPOSSIBLE
    if a in visited:                        # already visited
        return IMPOSSIBLE
    dists = [
        1 + min_path(matrix, (a[0]+1, a[1]), b, visited+[a]),
        1 + min_path(matrix, (a[0]+1, a[1]-1), b, visited+[a]),
        1 + min_path(matrix, (a[0]+1, a[1]+1), b, visited+[a]),
        1 + min_path(matrix, (a[0]-1, a[1]), b, visited+[a]),
        1 + min_path(matrix, (a[0]-1, a[1]-1), b, visited+[a]),
        1 + min_path(matrix, (a[0]-1, a[1]+1), b, visited+[a]),
        1 + min_path(matrix, (a[0], a[1]-1), b, visited+[a]),
        1 + min_path(matrix, (a[0], a[1]+1), b, visited+[a]),
    ]
    return min(dists)

# another solution
def min_path2(matrix, a, b, visited=[]):
    if a == b:                              # final position
        return 0
    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 matrix[a[0]][a[1]]:                  # an obstacle
        return IMPOSSIBLE
    if a in visited:                        # already visited
        return IMPOSSIBLE

    # find a min path
    mindist = IMPOSSIBLE
    possible = [(0,1), (1,0), (1,1), (-1,-1), (-1,0), (-1,1), (0,-1), (1,-1)]
    for p in possible:
        l, c = a[0]+p[0], a[1]+p[1]
        # try the direction
        d = 1 + min_path2(matrix, (l, c), b, visited+[a])
        # see if it's the best so far
        if d < mindist:
            mindist = d
    return mindist
```

The end.

FPRO, 2018/19