DELFT UNIVERSITY OF TECHNOLOGY
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

TW1090 Introduction to Programming
Examination
January 22, 2019, 09:00 – 12:00 h.

Preparation

First of all, read the instruction on the separate "Instructions page". Do not start with the assignments before you have changed the file names in the way explained on the "Instructions page".

In your implementations only use the modules / libraries which are already imported by means of import statements, so do not add any import statements, unless mentioned explicitly in the assignment. Do not change the main program, if already present. So, only add implementations of the functions asked for. Test the functions using the main program. Also fill in your name and student number in the comments section at the top of the program (see the contents of the .py files). Open the .py file you want to edit by double clicking on the file name or icon. You will enter the IDLE environment. Be sure that only one .py file is open at the same time. Save your .py files regularly.

Norm: Ass 1: 7 points. Ass 2: 4 points. Ass 3: 7 points. **Score: total points + 2** **Grade: score / 2**

Assignment 1 **(a: 1 point; b: 4 points; c: 2 points)**

Your goal is to implement functions `throw(n)`, `play(n, m)` and `print_table(c, y, s)`. A small test program is available in the file **`A1.py`** . Implement the functions in **`A1.py`**. During the game the user will only push the Enter key (to throw with the dice). So you do **not** need to check for incorrect input.

a) Implement a function `throw(n)` that simulates a throw with `n` dice and returns the total number of pips (dots, in Dutch: ogen).

b) Implement a function `Play(n, m)` that plays a game one time. The rules of the play are as follows:
The game score starts at 0. First the computer throws with `n` dice and the total number of pips is stored in a list `compu`. Then the user is asked to push the Enter key. The user throws with n dice and the total number of pips is stored in a list `your`. The difference of the two throws is calculated and added to the game score. The difference is positive if your throw has a higher number of pips than the computer throw and negative otherwise. The (intermediate) game scores are stored in a list `score`. If the game score is larger or equal `m` the user has won the game and the text "YOU WON" is printed. If the game score is smaller or equal `-m` the computer has won and the text "YOU LOST" is printed. Finally the lists `compu`, `your` and `score` are printed. Below part c) an example game play is shown (with 2 dice and `m` = 5).

c) Implement a function `print_table(c, y, s)` that has the lists with the computer throws, the user throws and the intermediate scores as arguments and prints a table with the results (game statistics). The table must be exactly in the same format as in the example game play below.

Example game play:

```
compu throw: 7
Push enter to throw:
your throw: 3
SCORE: -4

compu throw: 4
Push enter to throw:
your throw: 8
SCORE: 0

compu throw: 7
Push enter to throw:
your throw: 9
SCORE: 2

compu throw: 7
Push enter to throw:
your throw: 7
SCORE: 2

compu throw: 5
Push enter to throw:
your throw: 9
SCORE: 6

YOU WON
compu: [0, 7, 4, 7, 7, 5]
your : [0, 3, 8, 9, 7, 9]
score: [0, -4, 0, 2, 2, 6]


GAME STATISTICS:
 compu throw  your throw       score
           0           0           0
           7           3          -4
           4           8           0
           7           9           2
           7           7           2
           5           9           6
>>>
```

Assignment 2 **(a: 2 points; b: 2 points)**

Your goal is to implement a function `t_depth(t)` and `t_is_balanced(t)` in the file **`A2.py`**.

We define a data structure for binary trees in the following way:

- A binary tree is a graph with nodes and arcs. From every node there are exactly 2 outgoing arcs.
- In every internal node we store a natural number.
- In the leaves of the tree nothing is stored.
- Our internal representation of a binary tree is a nested dictionary (**not** the representation for directed graphs form the lecture slides).
- Every dictionary (except for the empty dictionaries in the leaves) has 3 keys called 'L' (for left subtree), 'R' (for right subtree) and 'V' (for value, i.e. the natural number stored in the node).

Examples:

```
tree0 = {}
```

an empty tree (a single leave).

```
tree1 = {'L': {}, 'R': {}, 'V': 3}
```

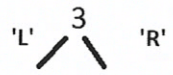


figure: tree1

```
tree2 = {'L': {'L': {},
               'R': {},
               'V': 2},
         'R': {'L': {},
               'R': {'L': {},
                     'R': {},
                     'V': 6},
               'V': 1},
         'V': 5}
```

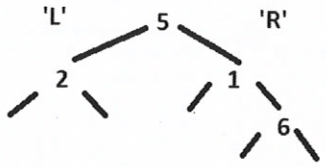


figure: tree2

```
tree = {'L': {'L': {},
              'R': {'L': {'L': {},
                          'R': {'L': {'L': {},
                                      'R': {'L': {},
                                            'R': {},
                                            'V': 9},
                                      'V': 4},
                                'R': {},
                                'V': 13},
                          'V': 2},
                    'R': {},
                    'V': 8},
              'V': 3},
        'R': {'L': {},
              'R': {'L': {},
                    'R': {},
                    'V': 6},
              'V': 1},
        'V': 5}
```

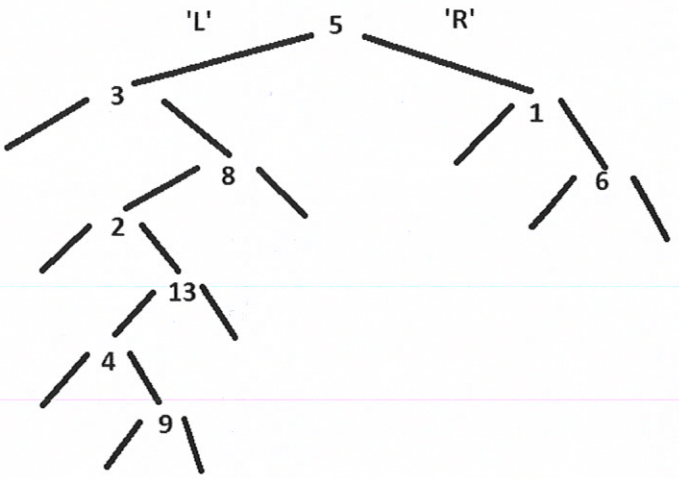


figure: tree

a) We define the depth of a tree to be the number of nodes (containing a number, so not the leaves) from the root to the lowest node containing a number. In the last example the path to the lowest node is 5, 3, 8, 2, 13, 4, 9, so this tree has depth 7. The depths of all trees above are:

```
tree0       depth 0
tree1       depth 1
tree2       depth 3
tree        depth 7
tree['L']   depth 6
```

Implement a recursive function `t_depth(t)` that returns the depth of tree `t`.

b) We define a tree to be balanced if for every (sub)tree the absolute difference of the depth of the 'L' subtree and 'R' subtree is 0 or 1.

```
tree0       balanced?: True
tree1       balanced?: True
tree2       balanced?: True
tree        balanced?: False
```

```
tree['L']   balanced?: False
```

Implement a recursive function t_is_balanced(t) that returns a boolean reporting whether t is balanced or not.

Expected test program output:

```
DEPTH:
0
1
3
7
6
IS BALANCED?:
True
True
True
False
False
>>>
```

Assignment 3 **(a: 2 points, b: 1 point; c: 1 point; d: 3 points)**

Your goal is to implement a class `Shape` in the file **`A3.py`** with the following attributes and methods:

```
Class Shape:

    Attributes:
    x:    a list with x coordinates
    y:    a list with y coordinates
    n:    the number of sub intervals in which [0,2π] is divided

    Constructor:
    Creates an instance of the Shape class in such a way that Shape(n) creates the
    lists of x-coordinates and y-coordinates of points on an n-sided regular polygon
    (Dutch: regelmatige n-hoek) and stores the value of n.

    Methods:
    Scale:      Scales the shape for which the method is called (so all x-coordinates
    are multiplied with sx and all y-coordinates are multiplied with sy).

    Translate:  Translates the shape for which the method is called (so to all x-
    coordinates tx is added and to all y-coordinates ty is added).

    Plot:       Draws the shape for which the method is called in a matplotlib
    window. The matplotlib window must already be open and plt.plot() to show the
    graphical output in the window must not be included in the method (because it
    must be possible to draw several shapes in the same window).
```

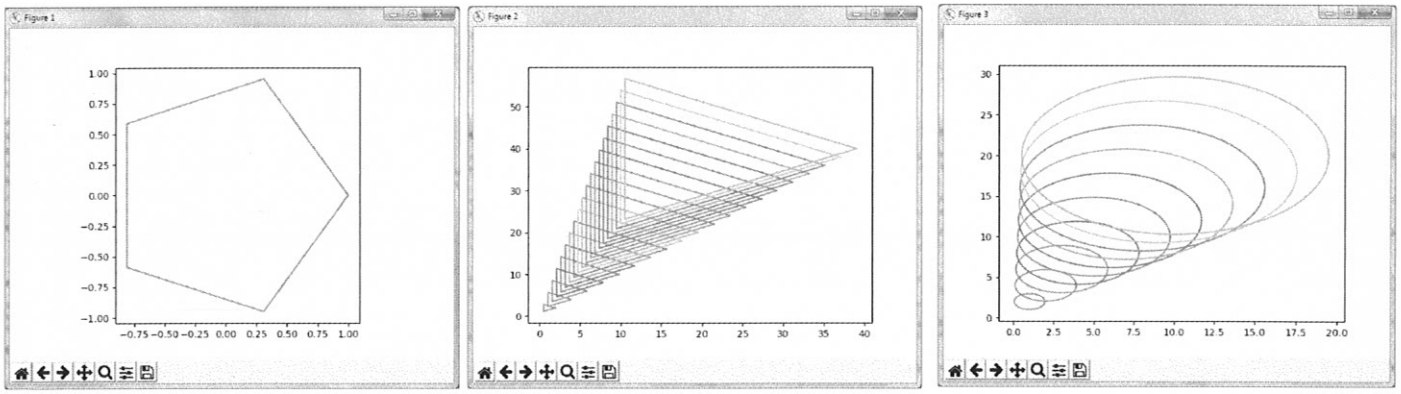
a) Start your implementation of the class `Shape` and implement the constructor.
b) Implement the method `Scale` with arguments `sx` and `sy`, that changes the object for which it is called in the way described above.
c) Implement the method `Translate` with arguments `tx` and `ty`, that changes the object for which it is called in the way described above.

d) **Outside the class** implement a function `draw_shapes` with arguments `shape, sx, sy, tx, ty` and `m` that draws `m` shapes in a matplotlib window as follows:

- The first polygon that is drawn is `shape` itself.
- The next one is `shape`, scaled with scale factors `sx, sy` and then translated over the vector `<tx, ty>`.
- The next shape is created from the original shape by first scaling with `2*sx, 2*sy` and then translating the result over the vector `<2*tx, 2*ty>`.
- This process continues until `m` shapes have been drawn.

Below example output of the test program is given after your implementation has been finished, including the correct images from part d).

```
x =  [ 1.          0.30901699 -0.80901699 -0.80901699  0.30901699  1.        ]
y =  [  0.00000000e+00   9.51056516e-01   5.87785252e-01  -5.87785252e-01
  -9.51056516e-01  -2.44929360e-16]
scaled:
x =  [ 2.          0.61803399 -1.61803399 -1.61803399  0.61803399  2.        ]
y =  [  0.00000000e+00   2.85316955e+00   1.76335576e+00  -1.76335576e+00
  -2.85316955e+00  -7.34788079e-16]
translated:
x =  [ 3.5         2.11803399 -0.11803399 -0.11803399  2.11803399  3.5       ]
y =  [ 0.5         3.35316955  2.26335576 -1.26335576 -2.35316955  0.5       ]
```



Finishing your session / handing in your work

Read the last part of the "Instructions page" and perform all actions to finish and hand in your work.

YOUR WORK WILL BE GRADED ONLY USING THE .PY FILES WITH YOUR NAME AND STUDENT NUMBER.
IF YOUR .PY FILES ARE NOT STORED ON DRIVE(P:) THEN YOU HAVE NOT HANDED IN YOUR WORK.

End of examination