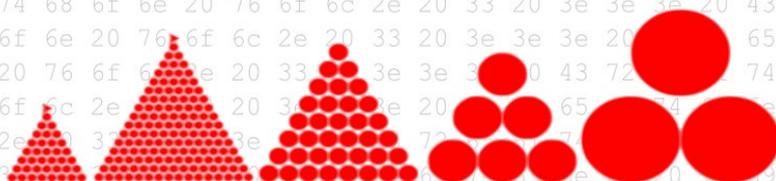
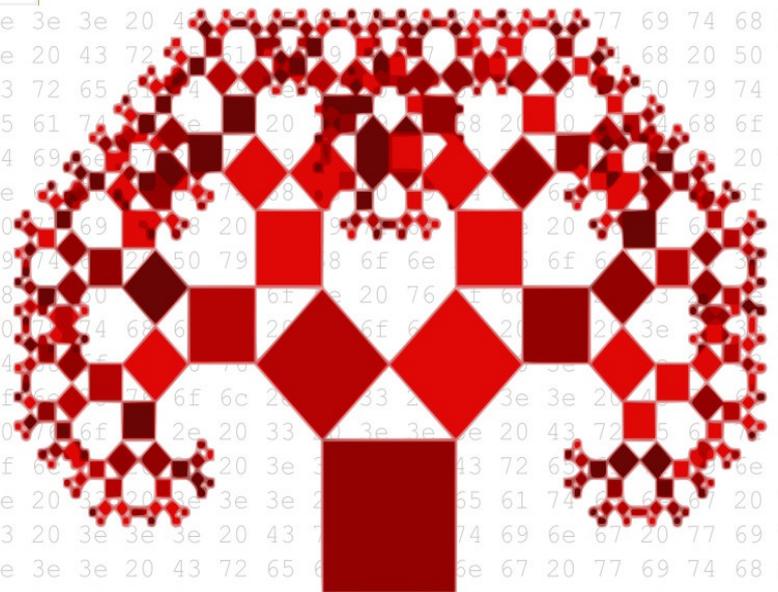
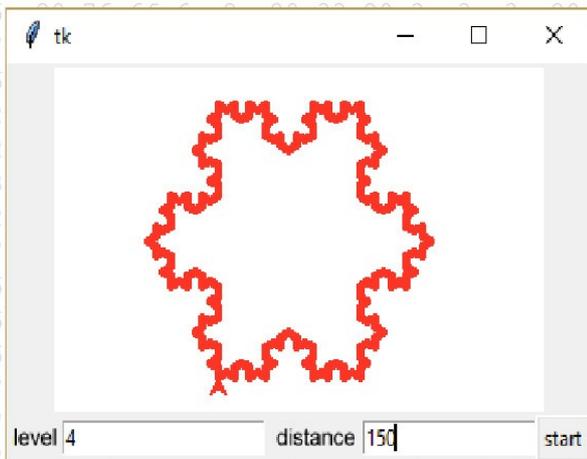
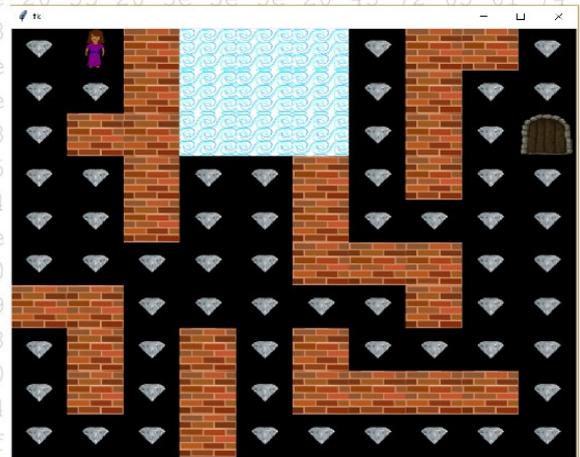
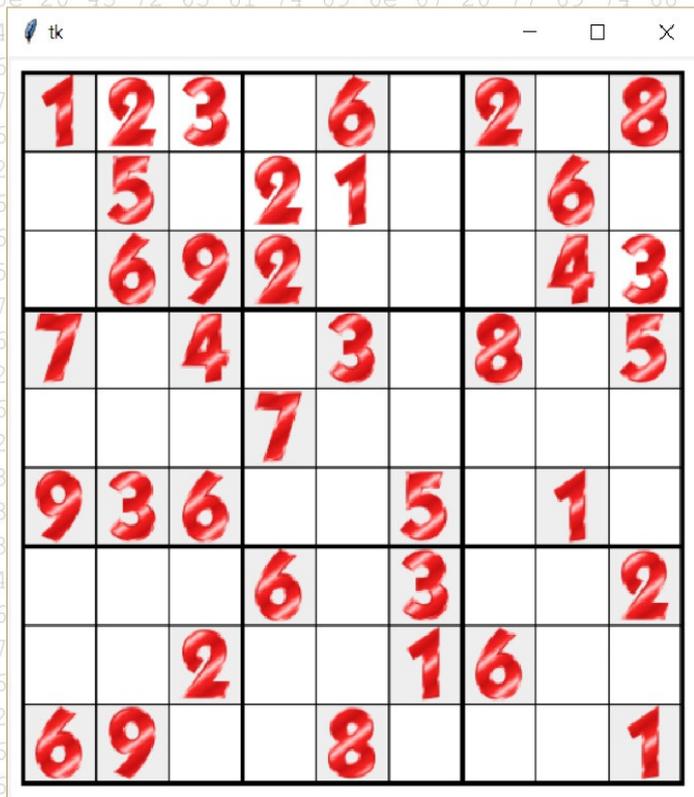


Creating with Python

Peter Kučera, Jaroslav Výboštok

Vol. **3**



Creating with Python vol. 3

Authors © Mgr. Peter Kučera, Mgr. Jaroslav Výboštok

Design © Mgr. Peter Kučera

Translation: Jana Ondíčová, Bruno Petrus, Joshua Ruggiero

First published, 2019

Version number: 20200212

Publisher: Peter Kučera

Copyright © 2019 by Peter Kučera, Jaroslav Výboštok

eBook preview

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

For permission requests, write to the publisher at peter.kucera@gmail.com or via www.creatingwithpython.com.

Ordering Information:

Quantity sales. Special discounts are available on quantity purchases by corporations, associations, schools, and others. For details, contact the publisher at peter.kucera@gmail.com or via www.creatingwithpython.com.

Visit the author's website at www.creatingwithpython.com

ISBN 978-80-570-1574-1 (pdf)

ISBN 978-80-570-1575-8 (epub)

ISBN 978-80-570-1576-5 (mobi)

Contents

[Contents](#)

[Introduction](#)

[1 List of Lists \(Two-Dimensional Array\)](#)

[1.1 Tic-Tac-Toe](#)

[1.2 The Diamond Labyrinth](#)

[2 Raster Graphics](#)

[3 Turtle Graphics](#)

[3.1 Drawing with the Turtle Module](#)

[Commands \(Methods\) Used in the Turtle Module:](#)

[Using Turtle Graphics in Tkinter](#)

[3.2 Working with Multiple Turtles](#)

[4 Recursion and fractals](#)

[4.1 Recursion](#)

[4.2 Right recursion and fractals](#)

[4.3 Recursion and functions with a return value, recursion without graphics](#)

[5 Classes and objects](#)

[5.1 Creation of a class and its uses](#)

[Class Traffic Sign](#)

[Class Flag](#)

[5.2 Creating a module with a class](#)

[5.3 Class composition](#)

[5.4 Class inheritance](#)

[Class Picture, Dragging pictures](#)

[5.5 Usage of objects inside programs](#)

[A snake](#)

[Playing cards](#)

[6 Graphs](#)

[6.1 Creating a graph](#)

[6.2 Graph representation](#)

[Adjacency matrix](#)

[Adjacency list - using dictionaries](#)

[6.3 Graph traversal](#)

[Depth-first search](#)

[Breadth-first search](#)

[Finding the shortest path](#)

[Finding the shortest path in a weighted graph \(Dijkstra's algorithm\)](#)

[7 Additional Python features](#)

[7.1 Ternary operator](#)

[7.2 Easier list creation](#)

[7.3 Default parameter values](#)

[7.4 Exceptions](#)

[List of commands](#)

[Bibliography](#)

Introduction

The textbook *Creating with Python* was published in blue; its second volume came out in green. However, the third component, red, was still missing to complete the RGB color spectrum. Therefore, this third volume which you are currently reading is red. It addresses topics that exceed the target requirements for the Slovak state-administered school-leaving exam, or Maturita. The textbook is suitable not only for students who like programming and would like to deal with it in greater depth in the future, but also for those who want to be equipped with a powerful tool for solving the problems of today's digital age.

We carefully formulated the content of this work using our many years of experience in teaching programming in high school.

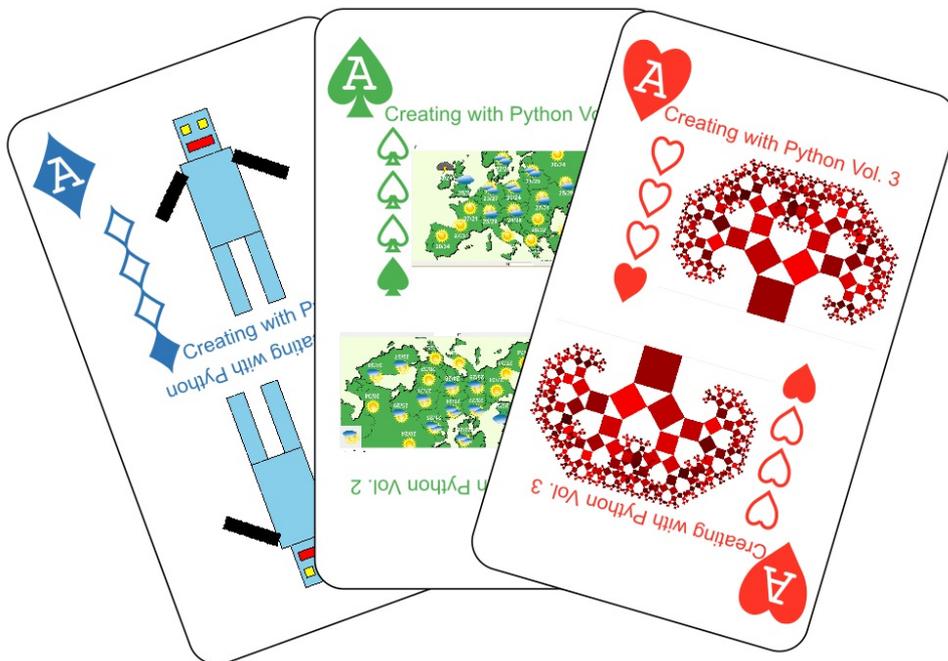
You will find many practical examples and solutions in this textbook as well as exercises to practice and questions that will encourage you to think, discover context, discuss in a group, experiment, search for bugs and intuitively seek out optimal solutions.

You will learn to create and use more complex structures, program algorithms using recursion – a powerful but elegant tool – and test your boundaries when using and creating graph algorithms. We believe you will also be interested in object programming.

If you have just found this third volume by chance and lack the basics, don't be afraid of returning recursively the previous level. You don't have to worry about infinite cycling, *Creating with Python Vol. 1* also solves trivial cases. :)

When programming with this textbook, we wish you fun, many pleasant moments, success, and yet also a few bug reports, which may bother you a little, but will teach you even more...

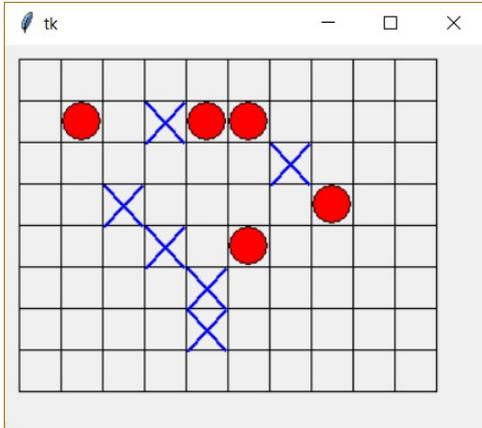
The authors



1 List of Lists (Two-Dimensional Array)

1.1 Tic-Tac-Toe

The following program creates a grid as preparation for a Tic-Tac-Toe game. Then, by clicking on the left mouse button either a circle or a cross is created at the position of the mouse click based on whose turn it is.



```
#p1-01.py
import tkinter
canvas = tkinter.Canvas(width=400, height=400)
canvas.pack()
number_of_squares_x, number_of_squares_y = 10, 8
sz = 30 #square size
grid_x, grid_y = 10, 10 # grid's topleft corner - x,y coordinates
mark = 1

def draw_grid(nx, ny, sz, gx, gy):
    for column in range(nx):
        for row in range(ny):
            canvas.create_rectangle(column*sz + gx, row*sz + gy,
                                   (column+1)*sz + gx, (row+1)*sz + gy)

def draw(column, row, mark):
    if mark == 1:
        canvas.create_oval(column*sz + grid_x + 1, row*sz+grid_y+1,
                           (column+1)*sz+grid_x - 2, (row+1)*sz+grid_y-2,
                           fill = 'red')

    if mark == 2:
        canvas.create_line(column*sz + grid_x + 1, row*sz+grid_y+1,
                           (column+1)*sz+grid_x - 1, (row+1)*sz+grid_y+1,
                           fill = 'blue', width = 2)
        canvas.create_line(column*sz + grid_x + 1, (row+1)*sz+grid_y+1,
                           (column+1)*sz+grid_x - 1, row*sz + grid_y+1,
                           fill = 'blue', width = 2)

def where(x, y):
    column = (x - grid_x) // sz
    row = (y - grid_y) // sz
    return column, row

def select(coords):
    global mark
    mark = 3 - mark
    column, row = where(coords.x, coords.y)
```

```
draw(column, row, mark)

draw_grid(number_of_squares_x, number_of_squares_y, sz, grid_x, grid_y)
canvas.bind('<Button-1>', select)
```

Questions:

1. Which object will appear on the screen on the very first click?
2. Which command in the program is there to make sure that a different object will be created on the next click?
3. What other ways are there to ensure the change of the object being created (`mark = 3 - mark`)?
4. Why is it impossible to create a cross or a circle over two squares of the board by clicking on the border between them?
5. What index does the first column (or the first row) of the playing board have?
6. What happens when we click multiple times on the same square?
7. What happens when we click outside of the playing board (the grid)?
8. How can we increase the number of squares on the playing board, the size of the individual squares or move the entire grid?
9. How do we figure out how many objects were placed on the board by a player during a game? We do not wish to considerably alter the original (already existing) program, therefore we could solve this simply by using a button and creating a function that will provide us with this information.

So far the crosses and circles created in our program have not been stored in any data structure. We could get information regarding the already existing objects using the properties of said graphical objects in the canvas (we have worked with these properties in Chapter 9 of Creating in Python 2). Suitably picked labels for the objects (tags) would certainly help. However, it would be far too complicated and without greater benefits.

Therefore, we will be using a different method in which the already created crosses and circles are held in a list. Every square on the board (even an empty one) is assigned one number in the list. For example: an empty square is represented by a 0, a circle by the number 1 and a cross by the number 2. For a playing board with 10 columns and 8 rows (a 10 x 8 grid), an 80-item list would be sufficient. How would we find out what is placed in the third column, second row? Every time we would need to calculate the index of that particular item in the list. For that, we can use a formula such as `(row-1) * number_of_squares_x + column - 1`.

```
>>> row = 2
>>> column = 3
>>> (row-1) * number_of_squares_x + column - 1
12
>>> row = 1
>>> column = 1
>>> (row-1) * number_of_squares_x + column - 1
0
>>> row = 8
>>> column = 10
>>> (row-1) * number_of_squares_x + column - 1
79
>>> row = 3
>>> column = 1
>>> (row-1) * number_of_squares_x + column - 1
20
```

To make the operations with the indexes of the squares easier, we can store the data differently. We'll make one list for every row of the board - in this case 8 lists. By creating one single list consisting of these 8 lists, we'll make a list of lists. An item of this list is a list consisting of all numbers assigned to a particular row. One row is a list of 10 numbers.

```
# p1-01a.py
board = []
for i in range(8):
    board.append([0]*10)
print(board)
```

```
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

We want to change an empty square in the second row, third column in the list `grid` to a cross. First of all, we specify the index of the row in question (which is 1, since the index 0 is assigned to the first row) and then the index of the column within the said row (which is 2, since the first column is assigned index 0 and the second column index 1). In this way we can change the value assigned to `board[1][2]` from 0 to number 2.

```
>>> board[1][2] = 2
>>> board
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 2, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

As the next step, we'll generalize this to work for the size of our playing board, which is stored in variables, and add it to our program before the definition of the function `draw_grid`:

```
#p1-01a-2.py
board = []
for i in range(number_of_squares_y):
    board.append([0]*number_of_squares_x)
print(board)
```

Now we can add a feature that checks whether a square is empty to the function `select()`. Only when a square is empty are we able to change its value in the list of lists, draw an object (a circle or a cross) and change the type of filling.

```
def select(coords): #p1-01b.py
    global mark
    column, row = where(coords.x, coords.y)
    if board[row][column] == 0:
        mark = 3 - mark
        board[row][column] = mark
        draw(column, row, mark)
```

Questions:

10. Why does the program report an error when we click outside of the grid?
11. How can we avoid this error?

Once the created objects are stored in the list of lists, we are able to save an unfinished game in form of a text file. To do that, we'll add the button `Save` to our program. The first line of the text file holds information regarding the size of our playing board (in the format total number of columns and total number of rows).

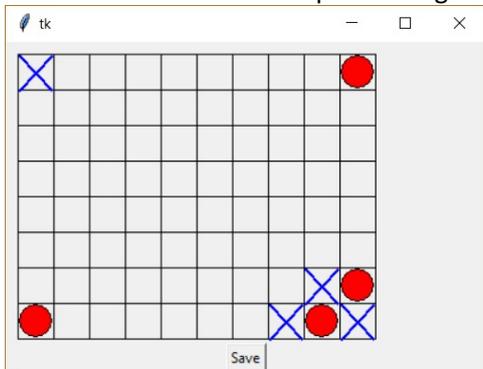
```
def save(): #p1-01c.py
    file = open('tictactoe.txt', 'w')
    file.write(str(number_of_squares_x)+' '+str(number_of_squares_y)+'\n')
    for row in board:
        newline = ''
        for item in row:
```

```

        newline = newline + str(item) + ' '
        newline = newline[:-1]+'\\n'
        file.write(newline)
    file.close()
button1 = tkinter.Button(text='Save', command=save)
button1.pack()

```

The saved text file for this particular game looks like this:



```

10 8
2 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 2 1
1 0 0 0 0 0 0 2 1 2

```

Now we'll add a **Load** button. This button stores the data loaded from the text file into a list and creates an accordingly sized grid and the objects. By reading a line and using the `split()` method, we can form a list consisting of strings representing the numbers in said line. Now we need to transform every item of the list from a string into an integer using the `int()` function. To do that we create the function `int_list()`, which transforms all items of the list into integers and gives us a list consisting of integers as the output.

```

def int_list(mylist):          #p1-01d.py
    output = []
    for item in mylist:
        output.append(int(item))
    return output

```

```

def load():                    #p1-01e.py
    global number_of_squares_x, number_of_squares_y, board
    file = open('tictactoe.txt', 'r')
    info = file.readline()
    info = info.strip()
    size = info.split()
    number_of_squares_x, number_of_squares_y = int_list(size)
    board = []
    for row in file:
        row = row.strip()
        row_list = row.split()
        row_list = int_list(row_list)
        board.append(row_list)
    canvas.delete('all')
    draw_grid(number_of_squares_x, number_of_squares_y, sz, grid_x, grid_y)
    draw_board()

```

```

button2 = tkinter.Button(text='Load', command=load)
button2.pack()

```

The `draw_grid()` function that is called after the text file is read goes over all items in the list `grid` and gradually creates the individual objects by calling the `draw()` function.

```
def draw_board():          #p1-01f.py
    for row in range(number_of_squares_y):
        for col in range(number_of_squares_x):
            draw(col, row, board[row][col])
```

So far, our program looks like this:

```
# p1-01g.py
import tkinter
canvas = tkinter.Canvas(width=400, height=400)
canvas.pack()
number_of_squares_x, number_of_squares_y = 10, 8
sz = 30 # square size
grid_x, grid_y = 10, 10      # grid's topleft corner - x,y coordinates
mark = 1
board = []
for i in range(number_of_squares_y):
    board.append([0]*number_of_squares_x)

def draw_grid(nx, ny, sz, gx, gy):
    for column in range(nx):
        for row in range(ny):
            canvas.create_rectangle(column*sz + gx, row*sz + gy,
                                    (column+1)*sz + gx, (row+1)*sz + gy)

def draw(column, row, mark):
    if mark == 1:
        canvas.create_oval(column*sz + grid_x + 1, row*sz+grid_y+1,
                            (column+1)*sz+grid_x - 2, (row+1)*sz+grid_y-2,
                            fill = 'red')
    if mark == 2:
        canvas.create_line(column*sz + grid_x + 1, row*sz+grid_y+1,
                            (column+1)*sz+grid_x - 1, (row+1)*sz+grid_y+1,
                            fill = 'blue', width = 2)
        canvas.create_line(column*sz + grid_x + 1, (row+1)*sz+grid_y+1,
                            (column+1)*sz+grid_x - 1, row*sz + grid_y+1,
                            fill = 'blue', width = 2)

def where(x, y):
    column = (x - grid_x) // sz
    row = (y - grid_y) // sz
    return column, row

def select(coords):
    global mark
    column, row = where(coords.x, coords.y)
    if board[row][column] == 0:
        mark = 3 - mark
        board[row][column] = mark
        draw(column, row, mark)

def save():
    file = open('tictactoe.txt', 'w')
    file.write(str(number_of_squares_x)+' '+str(number_of_squares_y)+'\n')
    for row in board:
        newline = ''
        for item in row:
            newline = newline + str(item) + ' '
        newline = newline[:-1]+'\\n'
        file.write(newline)
```

```

file.close()

def int_list(mylist):
    output = []
    for item in mylist:
        output.append(int(item))
    return output

def draw_board():
    for row in range(number_of_squares_y):
        for col in range(number_of_squares_x):
            draw(col, row, board[row][col])

def load():
    global number_of_squares_x, number_of_squares_y, board
    file = open('tictactoe.txt', 'r')
    info = file.readline()
    info = info.strip()
    size = info.split()
    number_of_squares_x, number_of_squares_y = int_list(size)
    board = []
    for row in file:
        row = row.strip()
        row_list = row.split()
        row_list = int_list(row_list)
        board.append(row_list)
    canvas.delete('all')
    draw_grid(number_of_squares_x, number_of_squares_y, sz, grid_x, grid_y)
    draw_board()

button1 = tkinter.Button(text='Save', command=save)
button1.pack()
button2 = tkinter.Button(text='Load', command=load)
button2.pack()

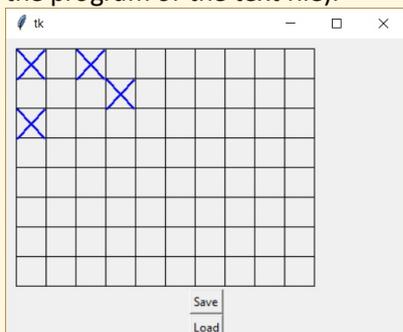
draw_grid(number_of_squares_x, number_of_squares_y, sz, grid_x, grid_y)

canvas.bind('<Button-1>', select)

```

Questions:

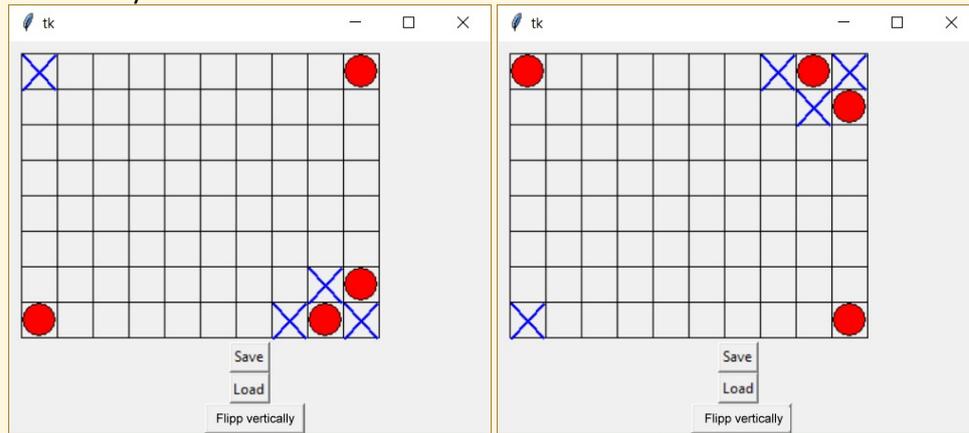
12. Which object will be drawn on the playing board on the next click after we have loaded any saved game (created by this program)? How can we ensure it will always be the right object (based on whose turn it is)?
13. How would we need to modify our program if we left out the spaces between the numbers in all the lines (except for the first one) in the text file?
14. Someone managed to create this playing board using our program (they have not made any changes to the program or the text file):



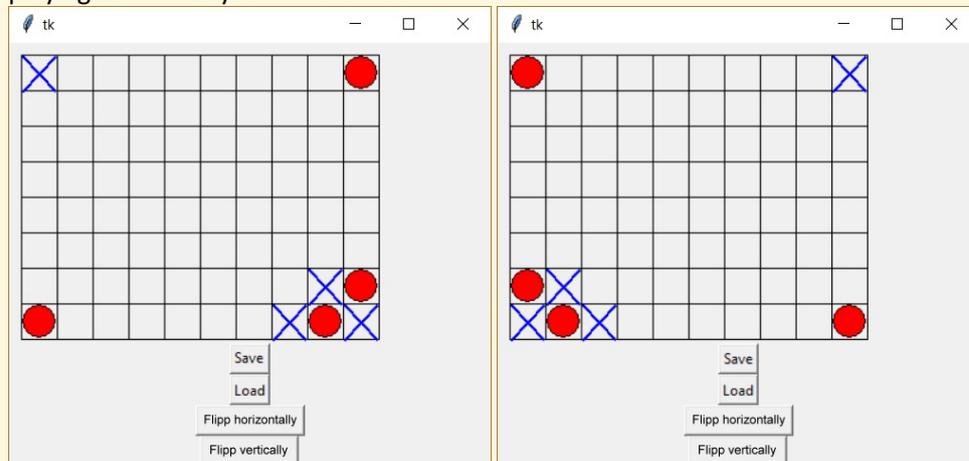
Try to figure out how this could have happened.

Exercises:

- 1 Add a button called `swap` to the program which will swap crosses for circles and vice versa (not only on the screen, but also in the list `grid`).
- 2 Extend the program with a `scale` widget ("slider") which we can use to adjust the size of the squares. Everything on our canvas will be redrawn immediately after an adjustment is made.
- 3 Define a `quantities()` function that will keep track of the number of crosses and circles already placed on the board.
- 4 Add a button called `flip vertically` to the program that will create a mirror image of the playing board like you can see below:



- 5 Add a button called `flip horizontally` to the program that will create a mirror image of the playing board like you can see below:



- 6 Create a program that will check whether there is a record of a real game saved in the text file or if someone just opened notepad and made up random values. For example, check if the proportions of the grid correspond to the number of lines and if the number of circles is the same as the number of crosses or if there is an extra circle.

Questions:

15. How would you change our program so that it would work like a real Tic-tac-toe game, recognizing the end of the game whenever at least five identical symbols are in a row (horizontally or vertically)?
16. What do the following `A()` and `B()` functions do? Why is the command `global` included in function `A()` but not included in function `B()`?

```

def A():
    global grid
    grid = grid[::-1]

def B():
    for r in range(len(grid)):
        grid[r] = grid[r][::-1]

```

17. How would we construct functions `A()` and `B()` (from the previous question) if we were either not familiar with or not allowed to use slices?

Let's go back to how we created the list of lists. In program `p1-01a.py` we used a `for` loop for this purpose.

```

# p1-01a.py
board = []
for i in range(8):
    board.append([0]*10)
print(board)

```

Is there an easier way to create a list of lists? Let's take a look at this solution:

```

>>> board = [[0] * 3] * 5
>>> board
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> board[0][1] = 2
>>> board
[[0, 2, 0], [0, 2, 0], [0, 2, 0], [0, 2, 0], [0, 2, 0]]
>>>

```

When we try this in the command line, at first it might seem that using this method will lead to a correctly created list of lists, too. However, as soon as we change the value of an item in one of the lists, the value of this item changes in all of the lists of the list. Why is that? The notion `[0] * 3` created one list consisting of three items - three zeros. Then a five-item list was created using the notion `[[0] * 3] * 5`. The problem though lies in the fact that this list holds five times the same reference to the place in the memory where the original three-item list is stored. All five items therefore refer to **the exactly same list consisting of three zeros**. Every time we change the value of an item in whichever of the five lists, we are modifying the same three-item list. The best way to understand this is to actually see it by creating a visualization at www.pythontutor.com

Write code in Python 3.6

```

1 board = [[0] * 3] * 5
2 print(board)
3 board[0][1] = 2
4 print(board)

```

Print output (drag lower right corner to resize)

```

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
[[0, 2, 0], [0, 2, 0], [0, 2, 0], [0, 2, 0], [0, 2, 0]]

```

Frames: Global frame, board

Objects: list (0, 1, 2) containing 0, 2, 0; list (0, 1, 2, 3, 4) containing references to the first list object.

Done running (4 steps)

List of commands

List of Lists (Two-Dimensional Array)

```
myList = []
for i in range(5):
    myList.append([0] * 10)
myList[4][9] = value
print(myList[3][5])
print(myList)

for row in range(5):
    for column in range(10):
        print(myList[row][column])

for row in myList:
    for item in row:
        print(item)
```

Raster Graphics

```
img = tkinter.PhotoImage(file='file.png')
canvas['width'] = img.width()
canvas['height'] = img.height()
canvas.create_image(0, 0, image=img)

color = img.get(x, y)
print(color[0], color[1], color[2]) #RGB
img.put('color', (x, y))
canvas.update()
```

Turtle Graphics

```
import turtle

t = turtle.Turtle()
t.forward(distance)
t.right(angle)
t.left(angle)
t.penup()
t.pendown()
t.pensize(width)
t.pencolor('color')
t.backward(distance)
t.dot(diameter)
t.setposition(x, y)
t.setheading(angle)
t.towards(x, y)
print(t.position())
print(t.heading())

ts = turtle.TurtleScreen(canvas)
t = turtle.RawTurtle(ts)
canvas.config(scrollregion=(0, 0, 800, 400))
ts.tracer(0)
ts.delay(0)
ts.update()
t.speed(0)
t.reset()
t.home()
t.clear()
t.fillcolor('color')
t.begin_fill()
t.end_fill()
turtle.mainloop()
```

Recursion and fractals

```
def myFunction(parameters):
    if condition:
        ...commands...
        myFunction(parameters)

def mySum(n):
    if n < 1:
        return 0
    return n + mySum(n - 1)
```

```
def myFunction(parameters):
    if condition:
        trivial case
    else:
        ...commands...
        myFunction(parameters)
        ...commands...
        myFunction(parameters)
        ...commands...
```

Classes and objects

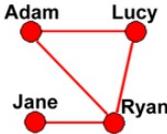
```
class Name_of_class:
    def __init__(self, parameters):
        self.attribute = parameter1
        ...commands...
    def method(self, parameters):
        ...commands...

instance = Name_of_class(parameters)
instance.method(parameters)
```

```
class myClass(Parent):
    def __init__(self, parameters):
        Parent.__init__(self, parameters)
        self.attribute = parameter1
        ...commands...
    def method(self, parameters):
        ...commands...
```

Graphs

```
vertices = ['Adam', 'Lucy', 'Jane', 'Ryan']
graph = [[0, 1, 0, 1],
         [1, 0, 0, 1],
         [0, 0, 0, 1],
         [1, 1, 1, 0]]
```



```
graph1 = {'Adam':{'Ryan':1, 'Lucy':1},
          'Lucy':{'Adam':1, 'Ryan':1},
          'Jane':{'Ryan':1},
          'Ryan':{'Adam':1, 'Jane':1, 'Lucy':1}}
```

Additional Python features

```
variable = value1 if condition else value2
myList = [expression for item in myList]
myList = [expression for item in myList if condition]
```

```
def myFunction(parameter1, parameter2=default_value):
    ...commands...

myFunction(value1)
myFunction(value1, value2)
myFunction(value1, parameter2=value2)
```

```
try:
    ...commands...
except:
    commands for exception
```