Università della Svizzera italiana

Faculty of Informatics

Programming Fundamentals I
Fall 2014
Topic 8: Robust programming

Prof. Mehdi Jazayeri

Assisted by: Anton Fedosov, Yuriy Tymchuk, Rui Xin

Week 8: 10 - 16 November, Due: 17 November

Instructions

Please read these instructions carefully.

Each week in the first ten weeks of the semester, you are supposed to master a topic and demonstrate your mastery to the teaching staff. To gain mastery, you should attend the lecture, study the topic, and do many exercises on the week's topic. The more exercises you do, the more confidence you will gain in your ability and your understanding. We will post some questions on iCorsi that you can use to test yourself. Also, the textbook has exercises that you can use to test yourself. Make sure you can answer ALL the exercises in the book.

To show your mastery of the topic, you will typically be given a program to study and then modify it to enhance its features. We will also give you a list of topics you could explore on your own if you want to extend your knowledge.

Finally, in the Monday atelier session, we will ask you to write a program and explain how it works.

For any questions, feel free to post on the iCorsi questions forum for the course or ask during the atelier session.

Week 8

Topic: This week you will work with exceptions and other features to write robust programs that deal with unexpected situations.

Introductory questions:

1. Use an assert statement to state your assumption that a given variable x will never be negative. What happens if x becomes negative during program execution?
2. Write a function that accepts two parameters and returns -1 if both parameters are not lists. It does something (we don't care what) if they are lists. It should use *type()* to check its parameter types.
3. Write a function that takes two parameters and uses *isintance()* to check that its parameters are both integers. What happens if the parameters are not integers?
4. Write a function that tries to read a file and uses exceptions to print a message if the file does not exist.
5. Explain why it does not make sense to put a *try...except* clause around a function definition. Also explain why it may make sense to put it around a function call.

Program to modify:

The code snippet below has a number of mistakes that prevent it from running. We have implemented a general exception handler to start catching all possible exceptions. Your task is to ensure smooth execution of the program, by creating a number of specific exception handlers (*try...except* statements). You should implement at least four different concrete exceptions (e.g. file not found, arithmetic errors, etc) that replace the general exception handler. We encourage you to start by observing the output of the program and gradually add new exception handlers.

```
import sys

try:
    import file
    y = int(input("Choose a denominator: "))
    f = open('file.txt')
    i = int(f.readline().strip())
    result = i / y
    print(result)
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Hint: Complete list of concrete exceptions is available at the following address: `https://docs.python.org/3/library/exceptions.html#concrete-exceptions`

Program to write: Reversi

In this section we ask you to implement a simple **Reversi** game that can interact with keyboard inputs.

Reversi[1] (also called *Othello*) is a game for 2 players on an 8×8 board. Players take turns by placing discs on the board. When a player places a disc on the board (this is called a 'move'), this causes the opponent's discs that are located in the same straight line and are 'captured' by the current player's discs, to be flipped to the current player's color. At the end, the player with more discs in his color wins. You can view `http://www.mathsisfun.com/games/reversi.html` to have a try.

In this section, your task is to implement a Reversi game played by two users' inputs. Your

[1] http://en.wikipedia.org/wiki/Reversi

code should achieve the following goals:

1. Print the board. We give you the basic arrangement of the game in a file *board.txt*. Read it and print the boaard in a readable format.
2. Simulate the game with keyboard inputs. Write a function *play()* to simulate the game by letting two users input ard boa position (e.g, (1, 2), (0, 3)...) in turns. (If you like, print the board after every move.) Pay attention that there may exist a time when the current player has no legal move available, in this case you should skip the user's round.
3. Output the result. When there is no available move for either player, the game ends. At that point, you should print the winner of the game.
4. Handle exceptions. In this program, you should consider exceptions. We provide some possible exceptions that you **may** focus on:
   - FileNotFoundError. Raised when file does not exist.
   - IndexError. Raised when the content of the file is smaller than a 8×8 board, and when the input position is beyond the range of the board.
   - TypeError. Raised when the input position value is of wrong type.
   - EOFError. Raised when the input hits an end-of-file condition (EOF) without reading any data.

An example of the output is as bellows:

```
>> play()
  0   1   2   3   4   5   6   7
- - - - - - - - - - - - - - - - -
0|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
1|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
2|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
3|   |   |   | * | o |   |   |   |
- - - - - - - - - - - - - - - - -
4|   |   |   | o | * |   |   |   |
- - - - - - - - - - - - - - - - -
5|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
6|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
7|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
Player o's turn, input the position, such as 0, 1: # enter 3, 2
  0   1   2   3   4   5   6   7
- - - - - - - - - - - - - - - - -
0|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
1|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
2|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
3|   |   | o | o | o |   |   |   |
```

```
- - - - - - - - - - - - - - - - -
4|   |   |   | o | * |   |   |   |
- - - - - - - - - - - - - - - - -
5|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
6|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
7|   |   |   |   |   |   |   |   |
- - - - - - - - - - - - - - - - -
Player *'s turn, input the position, such as 0, 1:
.....
.....
.....
# At the end
  0   1   2   3   4   5   6   7
- - - - - - - - - - - - - - - - -
0| o | * | * | * | * | * | * | o |
- - - - - - - - - - - - - - - - -
1| * | * | o | * | * | * | o | o |
- - - - - - - - - - - - - - - - -
2| * | * | * | * | * | o | o | o |
- - - - - - - - - - - - - - - - -
3| o | * | * | * | * | * | o | o |
- - - - - - - - - - - - - - - - -
4| * | * | * | o | o | o | o | o |
- - - - - - - - - - - - - - - - -
5| * | * | * | o | * | * | o | o |
- - - - - - - - - - - - - - - - -
6| * | o | * | * | * | o | o | o |
- - - - - - - - - - - - - - - - -
7| * | * | * | * | o | o | o | o |
- - - - - - - - - - - - - - - - -
Player o has no available moves. Skipped.
Player * has no available moves. Skipped.
* wins
```

Here we provide a sample of the main function *play()* and some of the functions' description. You are not forced to use exactly this sample to form your solution, but we encourage you to organize your code in a clear way.

```
"""
    reversi.py
    This is a simple Reversi game without AI.
    Started by calling play().
"""
sym = {0:'o', 1:'*'} #represent player symbols

def play():
    """
```

```
    This is a sample of the main function.
    """
    board = read('board.txt')
    show(board)

    turn = 0
    skip = 0
    # When skip is larger than 2 (two turns with no moves
       possible), end the game.
    while skip < 2:
        avail_moves = a_mvs(board, turn)
        if len(avail_moves) > 0:
            pos = get_input(avail_moves, turn)
            board = place(board, pos, avail_moves, turn)
            skip = 0
            show(board)
        else:
            print('Player ' + sym[turn]
                  + ' has no available moves. Skipped.')
            skip += 1
        turn = (turn + 1) % 2
    winner(board)


def read(filename):
    """
    Read from file ${filename}. If it doesn't exist or in
     wrong format, use a default board.
    """


def show(board):
    """
    Print the board in certain format.
    """


def a_mvs(board, turn):
    """
    Return the avalable moves on the board of the current player
       .
    """


def get_input(avail_moves, turn):
    """
    Get input position from users. Pay attention to possible
       exceptions.
```

```
    """


def place(board, pos, avail_moves, turn):
    """
    Perform the placing and flipping of discs.
    """


def winner(board):
    """
    Count discs and print the winner.
    """
```

Attention:

- You may need to modify the code of the sample fucntions, and add more necessary functions to complete the program of your individual solution.
- Your program is not strictly required to act as the example, but must perform similarly.
- You may not need to consider all listed exceptions, depending on your implementation. But you should try your best to keep your program robust.
- You're **not allowed to** use topics that havn't been taught yet.
- Compared to previous assignments, this one is a little bit more complex to build. Try to divide the task into several reachable steps/functions, such as checking one straight line started from a given square → checking a single square → checking all sqaures.
- After successfully completing this program, you will be ready for doing a project.

Study further: TDD your code

Assertion can be used in smart way to set up a systematic way to test your code. Imagine that you are writing the answers to the questions of an assignment. Let's assume that your file with solutions is named *assignment6.py*. And one of the assignments asks you to compare if two lists are equal.

You could start with the folowing code in a file called *assignment6_tests.py*:

```
import assignment6

# first question
assert assignment6.lists_equal([1, 2, 3], [1, 2, 3])
assert not assignment6.lists_equal([1, 2, 3], [1, 2])
```

When you run this file for the first time, you risk to get an error like:

```
ImportError: No module named assignment6
```

This means that you don't have the file *assignment6.py* in your directory, and you should create it. Then you can encounter an error stating:

```
AttributeError: 'module' object has no attribute 'lists_equal'
```

This means that you have not defined that *lists_equal* function. After you define it, you may get the warnings about the assertions like:

```
assert solutions.lists_equal([1, 2, 3], [1, 2, 3])
AssertionError
```

Which means that your *lists_equal* function is not working as expected.
When your test script runs without any assertion errors, you can add more complicated tests:

```
assert solutions.lists_equal([1, [2, 3]], [1, [2, 3]])
assert not solutions.lists_equal([1, [2, 3], 4], [1, [2, 2], 4])
```

After your test work without assertion errors, you can continue to the next question. You can reuse this file everytime you make a change to your programs. Having tests available for your code makes it easier to make sure that the changes that you make don't alter the expected functionality or what was already working. This kind of testing is called *regression testing* and Writing test upfront is called TDD (Test Driven Development). Test driven develoment allows you to describe the goal the you want to achieve with your program before you start to implement the program. The tests serve as a specification of the program you want to write.