Simulation – A bigger programming project

Read: *Chapter 9 from textbook*

Lab 8: The tic-tac-toe project you'll build below is **needed** for your lab problems. Therefore, it is essential that you implement it following the process described below *and with lots of comments*.
Keep it in your directory, in order to have it available in your lab.

In this unit, we will deal with the development of a bigger programming project that will incorporate what you have learned so far. The project falls in the scope of a (maybe the most) basic application of computers, namely **simulation.** Using simulation, we try to solve real-world problems by modelling real-world processes to provide otherwise unobtainable information. A basic component of simulation is *randomness*. Therefore we will begin by producing (pseudo-)random numbers in Python, and then we will move to building our project. Along the way, there is a parallel (similar) project that you should be building, following the example of the book.

Randomness

- Python has the library `random` with functions used to produce pseudorandom numbers[1]. We are going to use two functions, `randrange()` and `random()`. Let's import them:

```
>>> from random import random, randrange
```

The `randrange(a,b,c)` will produce an integer multiple of c that is between a and b, not including b (a,b,c are integers). Try a few times the same call to `randrange`, say `randrange(3,95,11)`, and note that the result "looks" random.

The `random()` function produces a random floating point number between 0 and 1. Note that the probability of random() returning a number between 0 and 0.65 is exactly 65%. We will take advantage of this in our projects.

Simulation project

For this project, we will follow what has been called the "top-down" approach.

- **Informal statement:** Racquetball is played between two players using a racquet to hit a ball in a four-walled court. One player starts the game by putting the ball in motion (serving). Players try to alternate hitting the ball to keep it in play, in what is referred to as a rally. The rally ends when one player fails to hit a legal shot. The player who misses the shot loses the rally. If the loser is the player who served, service passes to the other player. If the server wins the rally, a point is awarded. Players can only score points during their own service. The first player to reach 15 points wins the game. In our simulation, the ability-level of the players will be represented by the probability that the player wins the rally when he or she serves. Write a program that asks the user for the service probability for both players, simulates multiple games with these probabilities, and prints out the results.

Practice project: The tic-tac-toe game we are going to create is the well-known game: There is a 3x3 board (matrix), initially empty. There are two players, the X-player and the O-player, that take turns each making one 'move' at a time. A 'move' is the placement of an 'X' or an 'O' in a currently empty position of the board by the X-player or the O-player, respectively (depending on whose turn it is to play). The X-player always starts first. The winner is the first player that will fill a horizontal or a vertical or a main diagonal line with his/her symbol (i.e., three consecutive 'X's or 'O's). Note that the game may end without a winner (actually kids stop playing the game once they realize that, after a little thought, draw is always the end-result). In this project, there are two possibilities for the game: either both players are human (2-player mode) or one player is human and the other player is the computer. To simplify things, let's always assume that the X-player is human. To help the player(s), after every move, the computer outputs the current state of the board in the following format:

---

1 The numbers are *pseudo*random and not truly random, because they are produced by a deterministic (instead of a random) process, starting from a *seed.* Every time someone provides the same seed to this process, it will produce exactly the same sequence of pseudorandom numbers.

```
-------
|X|O| |
-------
| |X|O|
-------
|X| |O|
-------
```

Obviously, this is a state of the game right after the O-player has made a move. Before the game starts, the computer asks whether it's in an 1-player or a 2-player mode, and it also asks for an initial state in the form of a list. For example, if the initial state is the one above, the user should enter the following:

```
> Enter initial state:
X,O, , ,X,O,X, ,O
```

In the 2-player mode, the game is quite straight-forward: the computer asks for a move by the player whose turn has come, the player enters the coordinates of an empty position, and the computer checks whether the move is legitimate; if it is, then it outputs the new state of the game. For example, if the state of the game is the one above, the interaction should look like the following:

```
> X-player's next move:
(0,2)
> New state:
-------
|X|O|X|
-------
| |X|O|
-------
|X| |O|
-------
```

Note that the coordinates of the upper-left corner is (0,0) and those of the lower-right (2,2). If the new move is a winning one, the computer announces it, otherwise the computer continues, until the game ends (either with a winning move or a draw).

In the 1-player mode, things become more complicated (and, therefore, more interesting). Now the computer plays the role of the O-player, and must decide its next move. The algorithms used for deciding the machine moves is what is called 'AI' (from 'Artificial Intelligence') in the game industry. The final project of this course will involve the development of sophisticated AI algorithms for tic-tac-toe, but for now we will settle for probably the stupidest possible one: the computer just picks an empty position *at random*. (Don't forget that this must be a *legitimate* move).

- **Detailed specification:** You can find this in the bottom half of p. 269.

Practice problem: Write a detailed specification for the practice project.

- **Top-Level design:** Here is a first attempt in breaking down the solution in distinct tasks:

```
Print an introduction
Get the inputs: probA, probB, n
Simulate n games of racquetball using probA and probB
Print a report on the wins for playerA and playerB
```

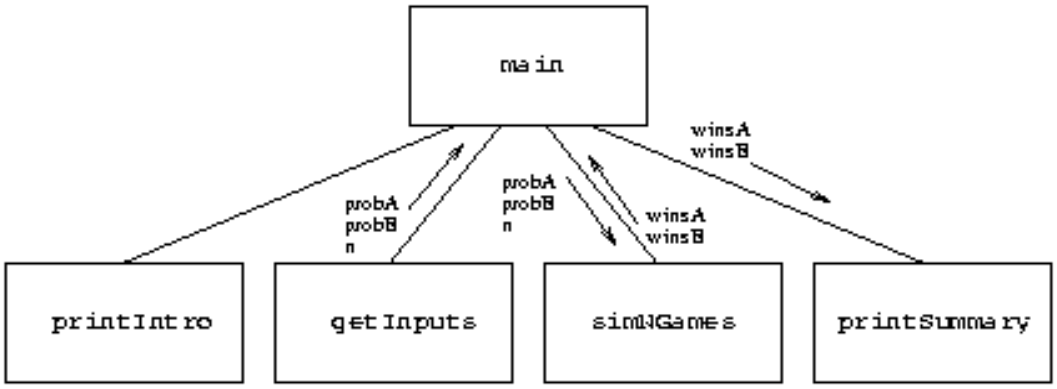
Practice problem: Do the same for your practice project.

After developing this basic break-down (p. 273-274), a picture of the algorithm starts emerging:

```
def main():
printIntro()
probA, probB, n = getInputs()
winsA, winsB = simNGames(n, probA, probB)   printSummary(winsA,
winsB)
```

Practice problem: Do the same for the practice project.

- **Second-Level design:** The different tasks can be further elaborated by using a *structure (or module hierarchy) chart*.



Practice problem: Do the same for your practice project..

At this point we can start mapping subtasks to functions. The `printIntro` and `getInputs` tasks can be implemented as follows:

```
def printIntro():
# Prints an introduction to the program
print("This program simulates a game of racquetball between two")
print('players called "A" and "B".  The abilities of each player is')
print("indicated by a probability (a number between 0 and 1) that")
print("the player wins the point when serving. Player A always")
print("has the first serve.\n")

def getInputs():
# RETURNS probA, probB, number of games to simulate
a = eval(input("What is the prob. player A wins a serve? "))
b = eval(input("What is the prob. player B wins a serve? "))
n = eval(input("How many games to simulate? "))
return a, b, n
```

Practice problem: Check your practice project design so far, to see whether any particular module can be mapped to a function.

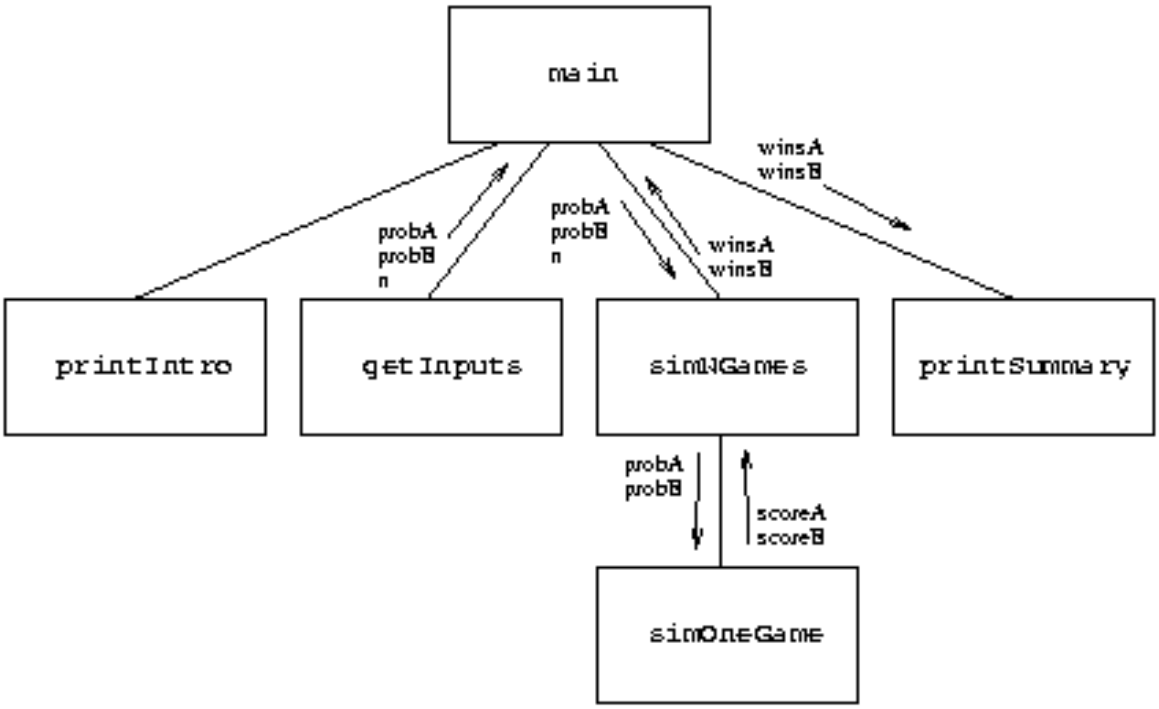
For the simNGames module, the process of developing ever more detailed versions is described in Section 9.3.4. We already have the signature (or interface) of the function from the chart:

```
def simNGames(n, probA, probB):
# Simulates n games of racquetball between players A and B
# RETURNS number of wins for A, number of wins for B
```

The final result is the following:

```
def simNGames(n, probA, probB):
# Simulates n games of racquetball between players A and B
# RETURNS number of wins for A, number of wins for B
winsA = winsB = 0
for i in range(n):
scoreA, scoreB = simOneGame(probA, probB)
if scoreA > scoreB:
winsA = winsA + 1
else:
winsB = winsB + 1
return winsA, winsB
```

The updated chart is the following:

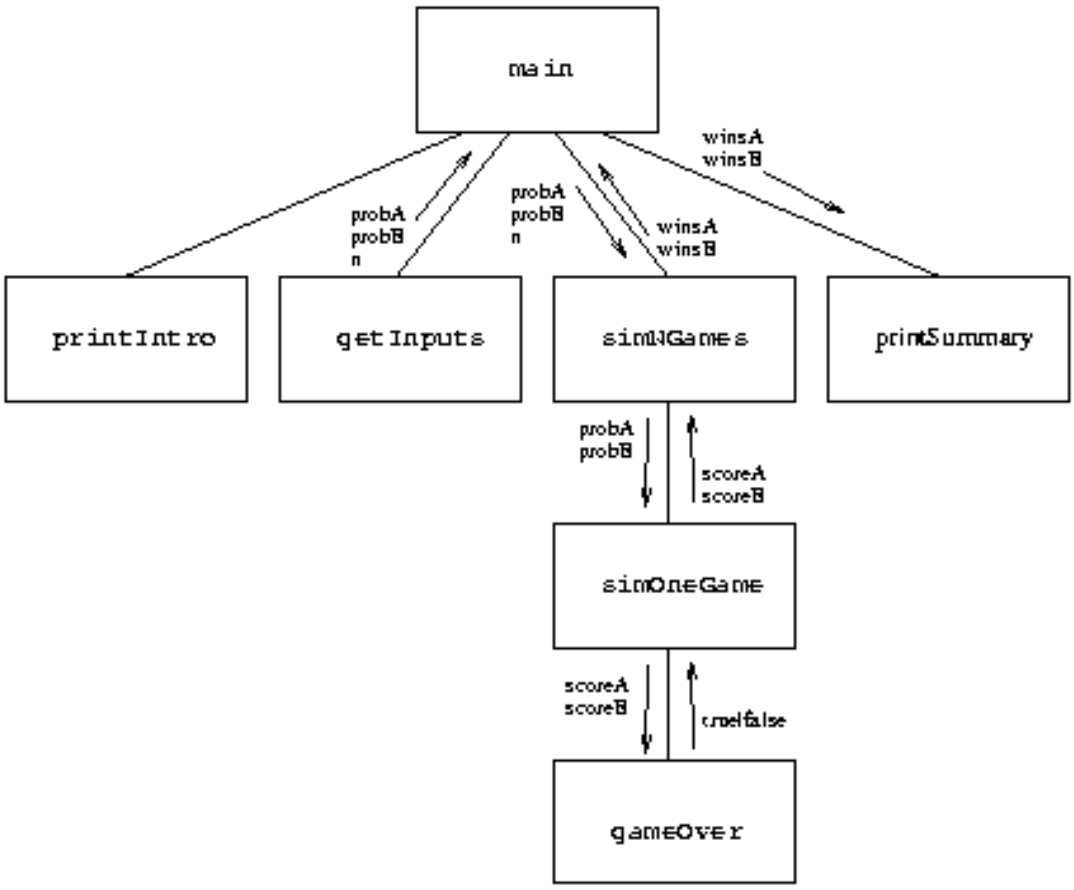


Practice problem: Do the same for the second-level design of your practice project.

- **Third-level design:** Note that now your chart has a third level. We can proceed in designing it. The design process is described in section 9.3.5. A crude description of what SimOneGame module should do is the following:

```
Initialize scores to 0
Set serving to "A"
Loop while game is not over:
Simulate one serve of whichever player is serving
update the status of the game
Return scores
```

After many rounds of refinement, the final function is described in p. 281, while the updated chart is the following:



Practice problem: Design the third level of the practice project.

- The process continues in the same vein with the fourth-level design etc. until the final product is rball.py in p. 282-283. Notice the last line of the program: this is how a typical Python program is run (instead of just calling main()); for now we are not interested in explaining this technical detail.

Practice problem: Finish the implementation of the practice project.

- **Testing:** After any implementation, the most important stage is the testing stage. One way to do your testing is to test each of your modules (functions) separately, usually in the reverse order of designing them (so, we would first test gameOver). In fact, you can perform this testing once a function implementation is complete, i.e., when the function and all its sub-levels have been implemented, without waiting for the whole implementation to finish; this gives you the ability to test smaller chunks of code, and, therefore, your testing can be thorough. See Section 9.4.

Practice problem: Test the correctness of your practice project implementation. *(Hint: Since you are going to modify your practice project in the lab, I hope you did remember to write plenty of comments...)*

Practice problem: Add to your design a function `ValidState(S,turn)` that takes a state `S` in the form of a list like `[X,O, , ,X,O,X, ,O]` for the first example above, and `turn` is either X or O, depending on whose turn it is to play. It returns 1 if it's a winning state for the X-player, -1 if it's a winning state for the O-player, 0 if it's a draw state, 2 if it's a legitimate state (i.e., none of the previous cases, and equal number of Xs and Os if `turn=X` or one less O if `turn=O`), and 3 otherwise (`S` is not a legitimate state).