

## CSI31 Lectures 21-22

**Topics:** *Chapter 9. Simulation and Design*

9.1 Simulating Racquetball

9.2 PseudoRandom numbers

9.3 Top-Down Design

9.4 Bottom-Up Implementation

**HW # 15** (last homework assignment):

p. 291 programming exercise 4

(only algorithm, but a thorough one)

## 9.2 Pseudo random numbers

there is nothing random about computers, they are instruction-following machines. So the numbers that we get with the help of computers are called pseudo-random numbers.

Here is the idea of the pseudo-random number generator:

- we start with some *seed* value.

- this seed value is fed into a function to produce a «random» number

- next time the random number is needed, the current value is fed back into the function to produce a new number.

The function has to be carefully chosen.

Python provides a library module that contains a number of useful functions for generating pseudorandom numbers. The functions in this module derive an initial seed value from the date and time when the module is loaded, so each time the program is run we get a different seed value.

We two functions of interest to us are `randrange` and `random`.

`randrange(a, b)` – generates pseudorandom integer from the interval [a,b]

`randrange(a, b, c)` – same, but with step c

`random` – generates pseudorandom floating point numbers between 0 and 1

Type in the following in the interactive window (and see what happens):

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

```
>>> randrange(1,10)
```

```
>>> randrange(1,10)
```

```
>>> randrange(1,10)
```

```
>>> randrange(1,10)
```

```
>>> randrange(1,10)
```

```
>>> randrange(1,10)
```

```
>>> randrange(1,10)
```

```
>>>randrange(5,55,6)
```

```
>>> from random import random
```

```
>>> random()
```

```
>>> random()
```

```
>>> random()
```

## 9.1 Simulating Racquetball

Right now we have almost all tools to solve interesting problems (i.e. the ones that are difficult or impossible to solve without the ability to write and implement computer algorithms).

**Simulation** – is one of techniques for solving real-world problems

**Examples** of problems that are solved with computer simulation:  
weather prediction, aircraft design, video games, etc.

Let's develop/write a program that will be a simple simulation of the **racquetball** game.

Racquetball is a sport played between two players using racquets to strike a ball in a four-walled court.

To start the game, one of the players puts the ball into play (called *servicing*). The players then alternate hitting the ball to keep it in play. This is a *rally*. The rally ends when one of the players fails to hit a legal shot. The player who misses the shot loses the rally. If the loser is the player who served, then 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.

Why do we want to simulate this play? What is of interest for us?

- the correspondence between ability-level of the players and the number of wins.

Is it true that if one of the players plays *slightly better* than the other, he/she should win *slightly more* games?

Convention: in our simulation the ability-level of players will be represented by the probability that the player wins the rally when he or she serves.

$P_A$  – the probability that the player A wins on his/her serve

$P_B$  – the probability that the player B wins on his/her serve

## Specification:

### Input:

- the service probabilities of two players (for Player A and for Player B)
- the number of games to be simulated

### Output:

- the number of games simulated
- the number of wins for Player A
  - (and what percent of the total number of games is it)
- the number of wins for Player B
  - (and what percent of the total number of games is it)

### Assumption:

- in each game player A serves first

## 9.3 Top-Down Design

### Idea:

- start with the general problem and try to express a solution in terms of smaller problems (i.e. break the original problem into sub-problems);
- then the same technique is applied to each smaller sub-problem
- eventually the problems get so small that they are trivial to solve

### basic algorithm:

print an introduction

get Pa and Pb

get n (the number of games)

simulate n games using Pa and Pb

print the report on the wins for Player A and Player B

## 9.3 Top-Down Design

### Idea:

- start with the general problem and try to express a solution in terms of smaller problems (i.e. break the original problem into sub-problems);
- then the same technique is applied to each smaller sub-problem
- eventually the problems get so small that they are trivial to solve

### basic algorithm:

print an introduction

get Pa and Pb

get n (the number of games)

simulate n games using Pa and Pb

print the report on the wins for Player A and Player B

## *Top-level Design*

### **basic algorithm:**

print an introduction

get Pa and Pb

get n (the number of games)

simulate n games using Pa and Pb

print the report on the wins for Player A and Player B

### **Main function:**

```
def main():
```

```
    Intro() # print an introduction
```

```
    Pa, Pb, n = getInput() # getting input from the user
```

```
    # simulating n games, returning the result of simulation
```

```
    WinsA, WinsB = simNgames(n, Pa, Pb)
```

```
    report(n, WinsA, WinsB) # reporting back to the user
```

### **What we did so far:**

outlined the sub-parts of the main function and the relationships between them (what is returned by each function, what is supplied as an argument to each function, which function goes after which)

## *Second-Level Design*

print an introduction

```
def Intro():  
    print 'Hello, this is a simulation of the racquetball game'  
    print 'This is a game for two players: Player A and Player B'
```

get Pa and Pb

get n (the number of games)

```
def getInput():  
    ProbA = input('Please, input the probability that the  
player A wins on his/her serve (i.e. Ability-level):'  
  
    ProbB = input('Please, input the probability that the  
player B wins on his/her serve (i.e. Ability-level):'  
  
    n = input('please, input the number of games you'd like to  
simulate:')  
  
    return ProbA, ProbB, n
```

print the report on the wins for Player A and Player B

```
def report(n, wA, wB):  
    print '%d games are simulated' % n  
  
    PercentA = wA*100.0/n  
    print 'Wins for Player A: %d (%0.1f)' %(wA, PercentA)  
  
    PercentB = wB*100.0/n  
    print 'Wins for Player B: %d (%0.1f)' %(wB, PercentB)  
  
    print 'Have a good day!'
```

simulate n games using Pa and Pb

```
def simNgames(n, Pa, Pb):
```

```
    ....
```

## Main function:

```
def main():  
    Intro() # print an introduction  
  
    Pa, Pb, n = getInput() # getting input from the user  
  
    # simulating n games, returning the result of simulation  
    WinsA, WinsB = simNgames(n,Pa,Pb)  
  
    report(n,WinsA,WinsB) # reporting back to the user
```

What we did so far:

outlined the sub-parts of the main function and the relationships between them (what is returned by each function, what is supplied as an argument to each function, which function goes after which);  
we also implemented the simple sub-parts of the program

! At each level of design we need to determine important characteristics of something and ignore other details (called **abstraction**)

## Designing simNgames

**algorithm:**

```
WinsA=0
WinsB=0
for i in range(n):      # loop n times
    simulate a game
    if PlayerA wins
        WinsA=WinsA+1
    else
        WinsB=WinsB+1
return WinsA,WinsB
```

*Third-Level Design*



Let's make «**simulate one game**» as a sub-part of **simNgames** part.

Recall our convention:

in our simulation the ability-level of players will be represented by the probability that the player wins the rally when he or she serves.

$P_A$  – the probability that the player A wins on his/her serve

$P_B$  – the probability that the player B wins on his/her serve

**comments:** for one rally (player A serves)

if  $P_A$  is the probability that the player A wins on his/her serve, then  $1-P_A$  is the probability that the player A loses on his/her serve +

**assumption:** in each game player A serves first

Players keep doing rallies until somebody collects 15 points: this suggests an **infinite loop**

! need to keep the track of players' scores (**scoreA**, **scoreB**)

How to deal with probabilities? Where to apply them?

**Algorithm of simOneGame:**

```
set serving to Player A
```

```
scoreA=0
```

```
scoreB=0
```

```
while scoreA != 15 or and scoreB != 15
```

```
    simulate one serving
```

```
    update the status of the game
```

```
return who wins
```

Here we need to decide **how**  
to signal back to our  
simNgames  
who won the game

Let's return **1** if the Player A wins,  
and **0** otherwise

Let's talk about probabilities.

### algorithm:

```
set serving to Player A
scoreA=0
scoreB=0
while scoreA != 15 or and scoreB != 15
    simulate one serving
    update the status of the game
return who wins
```

### implementation:

```
def simOneGame(Pa,Pb):
    serving='A'
    scoreA=0
    scoreB=0
    while scoreA!=15 or and scoreB!=15:
        if serving == 'A': # A is serving
            if random() < Pa: # A wins the serve
                scoreA=scoreA+1
            else: # A loses the serve
                serving='B'
        else: # B is serving
            if random() < Pb: # B wins the serve
                scoreB=scoreB+1
            else: # B loses the serve
                serving='A'
    if scoreA==15:
        return 1
    elif scoreB == 15:
        return 0
    else:
        print 'Something went wrong'
        print 'None of the players collected 15 points, but the rally is over'
```

## 9.4 Bottom-Up Implementation

start with sub-sub-....-sub parts

Dont's forget testing!