M30299 – Programming
Lecture 14 – Design and Simulation

Matthew Poole & Nadim Bakhshov
moodle.port.ac.uk

School of Computing
University of Portsmouth

2020/21

Introduction to lecture

- We've now covered almost all the main programming language concepts (data types, functions, control structures).
- However, most of the exercises and examples we've studied have been written using one or two (fairly short) function definitions.
- We'll now start writing complete, but still fairly short, **programs**.
- In these two lectures we'll see how to **design** programs using a technique known as **top-down design**.
- The idea of top-down design is to express a solution to a large problem in terms of smaller sub-problems.
- Each sub-problem is then solved using the same technique; at some point, sub-problems become small enough to solve easily.

A simulation problem

- We'll introduce top-down design using a **simulation** problem.
- That is, we'll write a program to simulate a "real-world" system, to try to understand and/or predict its behaviour.
- A tennis player has noticed that he wins about 40% of points, yet he wins much fewer than 40% of his games.
- We'll write a program that estimates the proportion of **games** he will win by simulating several tennis games.
- To do this, we'll need to use a **probabilistic** approach—the outcome of each point will be based on a **probability**.
- Such simulations are often called **Monte Carlo** simulations.

Generating random numbers

- Probabilistic simulation problems like this require the generation of **random numbers**.
- We can generate random numbers, evenly distributed between 0 and 1, using the `random` function of the `random` module:

```
>>> from random import random
>>> random()
0.95310838187740532
>>> random()
0.26136656748442944
>>> random()
0.48660655656290897
```

Simulating coin flips

- As a simple example of a simulation problem, suppose we need to simulate a single flip of a coin.
- We know that the probability of obtaining “heads” is 0.5, and so we can write:

```
>>> if random() < 0.5:
        print("Heads")
    else:
        print("Tails")
Tails
```

- If we had a **biased** coin where the probability of heads was 0.7, we could simply change the value in the above code.

Top-down design

- Let's begin to design a solution to the tennis simulation problem.
- Before we begin, let's pin down what the inputs & outputs of the program should be:
  - inputs: the "point win probability" (the proportion of points the player wins), and the number of games to simulate.
  - outputs: the number & proportion of games won by the player.
- The process of top-down design begins by writing down a algorithm giving the main steps that will provide a solution.
- The following seems reasonable:

```
get user inputs (prWinPt and numGames)
simulate numGames games with prWinPt
display number of wins and proportion of wins
```

First-level design

- For now, we ignore the complexity of each of these three steps, and just assume that there exist functions that do them for us.
- e.g., we suppose that a function `getInputs()` exists that asks the user to enter the point win probability and the number of games.
- Thus, to implement the first line of the algorithm we just write:

```
prWinPt, numGames = getInputs()
```

- Similarly, we suppose the existence of a function that simulates several games with for a given point win probability.
  - this function will need to take `prWinPt` and `numGames` as parameters;
  - it will need to return the number of games the player wins.
- We can therefore write:

```
wins = simulateNGames(prWinPt, numGames)
```

First-level design

- Finally, we suppose there is a function `printSummary` that displays the number and proportion of wins.
- This function will need `wins` and `numGames` as arguments:

  `printSummary(wins, numGames)`

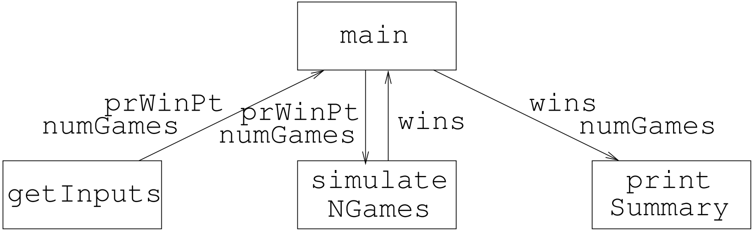
- We can combine these functions into a `main` function:

```
def main():
    prWinPt, numGames = getInputs()
    wins = simulateNGames(prWinPt, numGames)
    printSummary(wins, numGames)
```

- This function will be executed when we run the program.

First-level design

- We can illustrate this design using a **structure diagram**:



- This shows the functions as rectangles and the information flow as arrows:
  - parameter names are shown on downward arrows; and
  - returned values are shown on upward arrows

Second-level design

- We are now left with three **sub-problems**: writing the functions `getInputs`, `simulateNGames` and `printSummary`.
- Let's begin with the easiest—`getInputs`.
- This can simply use two `input`s to get the required values from the user, and then return these values:

```
def getInputs():
    prWinPt = float(input("Prob. of winning a point: "))
    numGames = int(input("Games to simulate: "))
    return prWinPt, numGames
```

Second-level design

- Writing `simulateNGames` is more difficult. Clearly:
  - it needs to contain a loop to simulate each of the games; and
  - it has to keep a count of how many games the player wins.
- The following pseudo-code seems a good start:

```
def simulateNGames(prWinPt, numGames):
    wins = 0
    loop numGames times:
        simulate a game using prWinPt
        if player wins:
            wins = wins + 1
    return wins
```

- The difficult part here is to simulate a game, so let's assume that there's a function `simulateGame` that does this!

Second-level design

- The `simulateGame` function will be given argument `prWinPt`.
- What might it return? ... there are many options.
- Let's suppose it returns the final points scored by the player and the opponent during the game.
- We can then complete `simulateNGames` as follows:

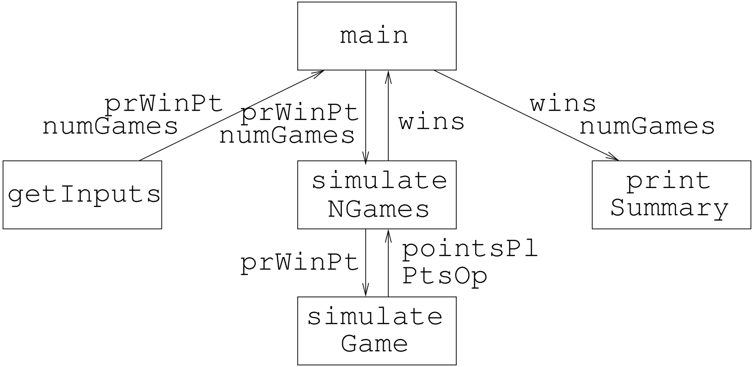
```
def simulateNGames(prWinPt, numGames):
    wins = 0
    for game in range(numGames):
        pointsPl, pointsOp = simulateGame(prWinPt)
        if pointsPl > pointsOp:
            wins = wins + 1
    return wins
```

Second-level design

- Let's complete the second-level design by implementing the `printSummary` function.
- This takes the number of games won and the total number of games played as parameters.
- It is fairly easy to write, as follows:

```
def printSummary(wins, numGames):
    proportion = wins / numGames
    print("Wins:", wins, end="  ")
    print("Proportion: {0:0.2f}".format(proportion))
```

Second-level structure diagram



Third-level design

- We have now completed the second-level design and need to write `simulateGame`.
- This should simulate a game between the player & opponent and when the game is over, return their points scored.
- Here's a reasonable pseudo-code solution, which uses randomly generated numbers to determine the outcome of each point:

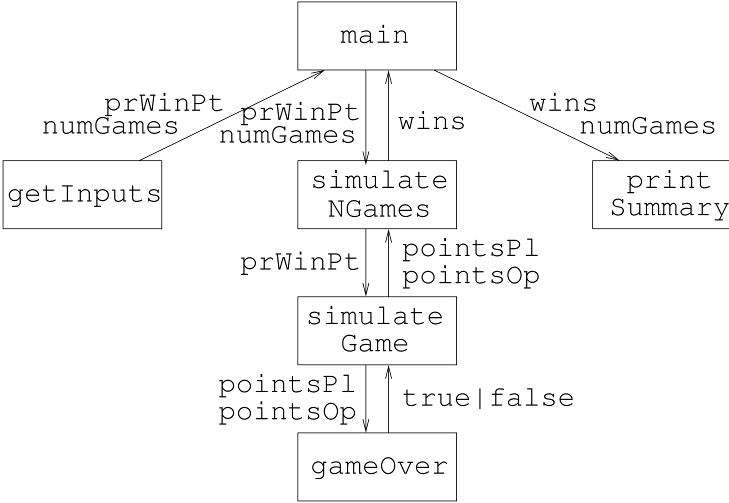
```
pointsPl, pointsOp = 0, 0
while game is not over:
    if random() < prWinPt:
        pointsPl = pointsPl + 1
    else:
        pointsOp = pointsOp + 1
return pointsPl, pointsOp
```

Third-level design

- The most difficult part is determining when the game is over.
- We assume there is a function `gameOver` that does this (it returns a Boolean), given the scores of the two players. Then:

```
def simulateGame(prWinPt):
    from random import random
    pointsPl, pointsOp = 0, 0
    while not gameOver(pointsPl, pointsOp):
        if random() < prWinPt:
            pointsPl = pointsPl + 1
        else:
            pointsOp = pointsOp + 1
    return pointsPl, pointsOp
```

Final structure diagram



Fourth-level design

- Our remaining problem is to write the `gameOver` function.
- Although the scoring of a tennis game (using "love", 15, 30, 40, "deuce, "advantage") seems complicated, it is equivalent to:
  - players scoring points 0, 1, 2, ... and
  - a player has won the game when he (or she) has scored at least 4 points and has at least two points more than the opponent.
- Therefore, we can use a Boolean expression to determine whether a game is over:

```
def gameOver(pointsPl, pointsOp):
    return (pointsPl >= 4 or pointsOp >= 4) and  \
               abs(pointsPl - pointsOp) >= 2
```

Executing the program

- Normally, a program is written to its own file; we might suppose our program will be stored in a file tennis.py.
- The above functions can appear in the file in any sensible order.
- We need to add, right at the bottom of the program file, the single statement:

```
main()
```

  that will call the `main` function when we execute the program.
- (The program is executed in the usual way.)
- Let's test the program to check it behaves as we would expect...

Executing the program

- If the player loses every point, we'd expect him to lose every game:

```
Probability of winning a point: 0
Games to simulate: 1000
Wins: 0    Proportion: 0.00
```

- Similarly, if the player wins every point, we'd expect him to win every game:

```
Probability of winning a point: 1
Games to simulate: 1000
Wins: 1000    Proportion: 1.00
```

- Question: What is another good value to test the program with?

Executing the program

- What if he wins 40% of his points?

```
Probability of winning a point: .4
Games to simulate: 1000
Wins: 256    Proportion: 0.26
```

- Let's see what happens if the player improves a bit:

```
Probability of winning a point: .6
Games to simulate: 1000
Wins: 741    Proportion: 0.74
```