THE UNIVERSITY OF WESTERN AUSTRALIA

Lecture 22
Simulation and Design - I

Objectives

- To understand the potential applications of simulation as a way to solve real-world problems.
- To understand pseudorandom numbers and their application in Monte Carlo simulations.
- To understand and be able to apply top-down design technique in writing complex programs.

Simulation

- *Simulation* can solve real-world problems by modeling real-world processes to provide otherwise unobtainable information.
  - *Physical techniques unavailable or very expensive*
- Computer simulation is used to predict the weather, design aircraft, create special effects for movies, etc.

Simulating Racquetball Games

- Denny often plays racquetball with players who are slightly better than he is.
- Denny usually loses his matches!
- Shouldn't players who are *a little* better win *a little* more often?
- Susan suggests that they write a simulation to see if slight differences in ability can cause such large differences in scores.

Background

- Racquetball is played between two players using a racquet to hit a ball in a four-walled court.
  - *Similar to squash*
- One player starts the game by putting the ball in motion – *serving*.
- Players try to alternate hitting the ball to keep it in play, referred to as a *rally*. The rally ends when one player fails to hit a legal shot.

Background

- The player who misses the shot loses the rally. If the loser is the player who served, serving passes to the other player.
- If the server wins the rally, a point is awarded and keeps serving. Players can only score points when they are serving.
- The first player to reach 15 points wins the game.

Analysis and Specification

- 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.
- Example: Players with a 0.60 probability win a point on 60% of their serves.
- The program will prompt the user to enter the service probability for both players and then simulate multiple games of racquetball.
- The program will then print a summary of the results.

Analysis and Specification

- Input: The program prompts for and gets the service probabilities of players A and B. The program then prompts for and gets the number of games to be simulated.

```
What is the probability player A wins a serve?
What is the probability player B wins a serve?
How many games to simulate?
```

- Output: The program then prints out a nicely formatted report showing the number of games simulated and the number of wins and the winning percentage for each player.

```
Games simulated: 500
Wins for A: 268 (53.6%)
Wins for B: 232 (46.4%)
```

- Assumption: Assume that all inputs are valid

PseudoRandom Numbers

- When we say that player A wins 50% of the time, that doesn't mean they win every other game. Rather, it's more like a coin toss.
- Overall, half the time the coin will come up heads, the other half the time it will come up tails, but one coin toss does not effect the next (it's possible to get 5 heads in a row, with probability 1/32).
- Many simulations require events to occur with a certain likelihood. These sorts of simulations are called Monte Carlo simulations because the results depend on "chance" probabilities.
- *Nothing random about computers but need to have a source of "random" numbers*

PseudoRandom Numbers

- A pseudorandom number generator works by starting with a seed value – current date and time. This value is given to a function to produce a "random" number.
- The next time a random number is required, the current value is fed back into the function to produce a new number.
- This sequence of numbers appears to be random, but if you start the process over again with the same seed number, you'll get the same sequence of "random" numbers.
- Python provides the random library module that contains a number of functions for working with pseudorandom numbers

PseudoRandom Numbers

- These functions derive an initial seed value from the computer's date and time when the module is loaded, so each time a program is run a different sequence of random numbers is produced.
- The four functions of greatest interest are `randrange`, `choice`, `random` and `seed`.
  - *`randrange()` – randomly select an integer value from a range. `range()` rules apply*
    - E.g. `randrange(1,6)` returns a number from `[1,2,3,4,5]` and `randrange(5,105,5)` returns a random multiple of 5 between 5 and 100, inclusive. (End value not included, as usual)

PseudoRandom Numbers

- *`choice()` – Choose a random member of a list*
  - E.g. `choice(['A', 'B'])`
- *`random()` – Returns a random number in the range [0 .. 1.0) (0 can be returned, but not 1.0*

```
>>> import random as rnd
>>> rnd.random()
0.79432800912898816
>>> rnd.random()
0.00049858619405451776
```

- *`seed()` – Assign the random number generator a fixed starting point (to give reproducible behaviour during testing)*

PseudoRandom Numbers

- The racquetball simulation makes use of the `random` function to determine if a player has won a serve.
- Suppose a player's probability when they serve is 70%, or 0.70.

```
if <player wins serve>:
   score = score + 1
```

- We need to insert a probabilistic function that will succeed 70% of the time.

PseudoRandom Numbers

- The racquetball simulation makes use of the `random` function to determine if a player has won a serve.
- Suppose we generate a random number between 0 and 1. Exactly 70% of the interval 0..1 is to the left of 0.7.
- So 70% of the time the random number will be < 0.7, and it will be ≥ 0.7 the other 30% of the time.
  - *The = goes on the lower end since the random number generator can produce a 0 but not a 1.*
- If `prob` represents the probability of winning the serve, the condition `random.random() < prob` will succeed with the correct probability.

```
if random.random() < prob:
    score += 1
```

Top-Down Design

- In top-down design, a complex problem is expressed as a solution in terms of smaller, simpler problems.
- These smaller problems are then solved by expressing them in terms of smaller, simpler problems.
- This continues until the problems are trivial to solve. The smaller pieces are then put back together as a solution to the original problem!

Top-Down Design

- Typically a program uses the *input*, *process*, *output* pattern.
- The algorithm for the racquetball simulation:

  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

- Whatever we don't know how to do, we'll ignore for now.
- Assume that all the components needed to implement the algorithm have been written already, and that your task is to finish this top-level algorithm using those components.

Top-Down Design

- First we print an introduction.
- This is easy, and we don't want to bother with it.

```
def main():
    printIntro()
```

- We assume that there's a `printIntro()` function that prints the instructions!

Top-Down Design

- The next step is to get the inputs.
- We know how to do that! Let's assume there's already a component that can do that called `getInputs()`.
- `getInputs()` gets the values for `probA`, `probB`, and `n`.

```
def main():
    printIntro()
    probA, probB, n = getInputs()
```

Top-Down Design

- If you were going to simulate the game by hand, what inputs would you need?
  - *probA*
  - *probB*
  - *n*
- What values would you need to get back?
  - *The number of games won by player A*
  - *The number of games won by player B*
- These must be the outputs from the `simNGames` function.

Top-Down Design

- We now know that the main program must look like this:

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

- What information would you need to be able to produce the output from the program?
- You need to know how many wins there were for each player – these will be the inputs to the next function.

Top-Down Design

- The complete main program:

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

- The name, parameters, and expected return values of these functions have been specified. This information is known as the interface (API) or signature of the function.
  - `main()` *in the programming projects was an API*

Separation of Concerns

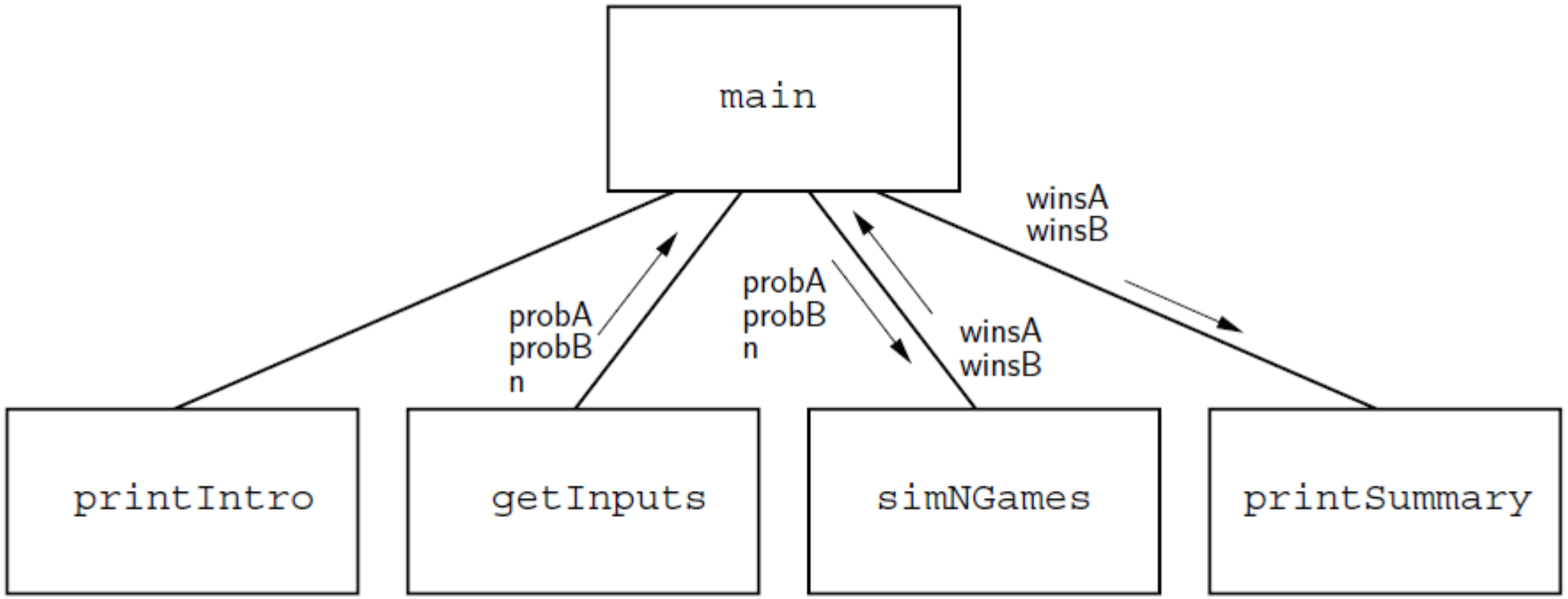
- Having this information (the *signatures*), allows us to work on each of these pieces independently.

- For example, as far as `main()` is concerned, *how* `simNGames` works is not a concern as long as passing the number of games and player probabilities to `simNGames` causes it to return the correct number of wins for each player.

Separation of Concerns

- In a structure chart (or module hierarchy), each component in the design is a rectangle.
- A line connecting two rectangles indicates that the one above uses the one below.
- The arrows and annotations show the interfaces between the components.

Separation of Concerns



Separation of Concerns

- At each level of design, the interface tells us which details of the lower level are important.
- The general process of determining the important characteristics of something and ignoring other details is called abstraction.
- The top-down design process is a systematic method for discovering useful abstractions.

Second Level Design

- The next step is to repeat the process for each of the modules defined in the previous step!
- The `printIntro` function should print an introduction to the program. The code for this is straightforward.

```
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 ability of each player')
    print("is indicated by a probability (a number between 0 and 1)")
    print("that the player wins the point when serving. Player A")
    print("always has the first serve.\n")
```

- Since there are no new functions, there are no changes to the structure chart.

Second Level Design

- In `getInputs`, we prompt for and get three values, which are returned to the main program.

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

Designing simNGames

- This function simulates *n* games and keeps track of how many wins there are for each player.
- "Simulate *n* games" sounds like a counted loop, and tracking wins sounds like a good job for accumulator variables.

```
Initialize winsA and winsB to 0
loop n times
  simulate a game
  if playerA wins
    Add one to winsA
  else
    Add one to winsB
```

Designing simNGames

We already have the function signature:

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

With this information, it's easy to get started!

```
def simNGames(n, probA, probB):
  # Simulates n games of racquetball between players A, B
  # RETURNS number of wins for A, number of wins for B
  winsA = 0
  winsB = 0
  for i in range(n):
```

Designing simNGames

- The next thing we need to do is simulate a game of racquetball. We're not sure how to do that, so let's put it off until later!
- Let's assume there's a function called `simOneGame` that can do it.
- The inputs to `simOneGame` are easy – the probabilities for each player. But what is the output?

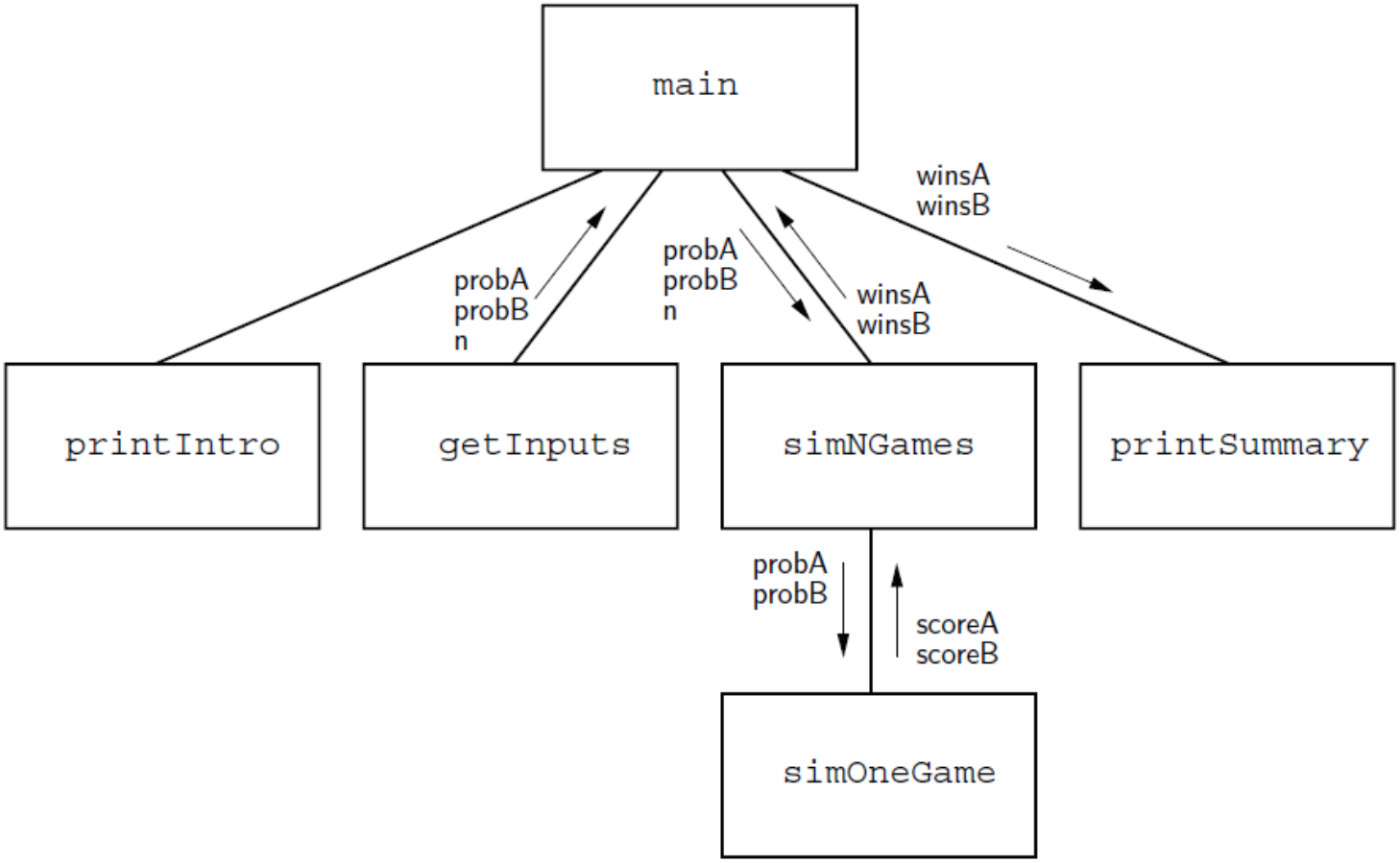
Designing simNGames

- We need to know who won the game. How can we get this information?
- The easiest way is to pass back the final score.
- The player with the higher score wins and gets their accumulator incremented by one.

Designing simNGames

```
def simNGames(n, probA, probB):
  # Simulates n games of racquetball between players A, 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 += 1
    else:
      winsB += 1
  return winsA, winsB
```

Designing simNGames



Third-Level Design

- The next function we need to write is `simOneGame`, where the logic of the racquetball rules lies.
- Players keep doing rallies until the game is over, which implies the use of an indefinite loop, since we don't know ahead of time how many rallies there will be before the game is over.
- We also need to keep track of the score and who's serving. The score will be two accumulators, so how do we keep track of who's serving?
- One approach is to use a string value that alternates between "A" or "B".

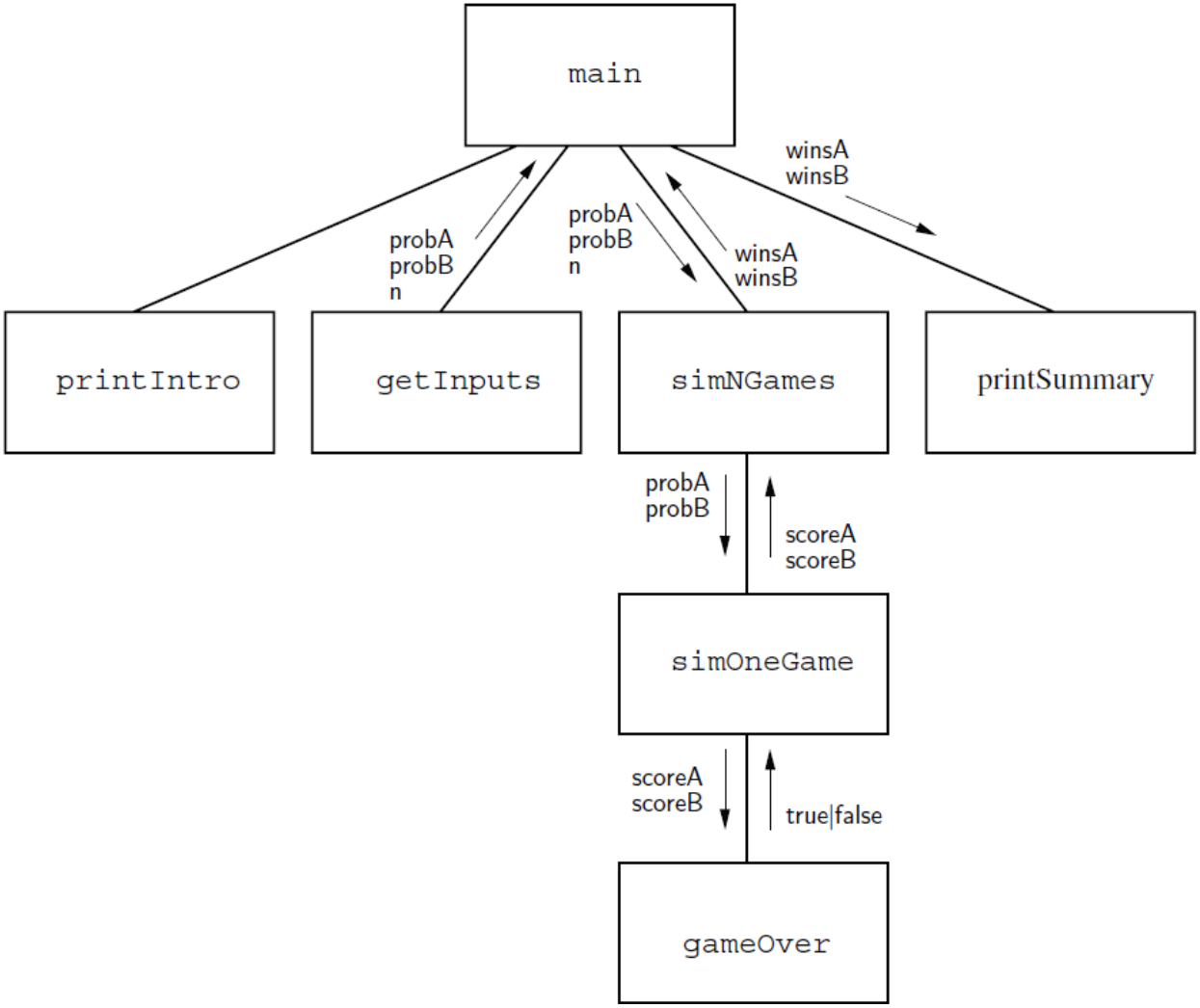
Third-Level Design

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

```
def simOneGame(probA, probB):
  scoreA = 0
  scoreB = 0
  serving = "A"
  while <condition>:
```

- What will the condition be?? Let’s take the two scores and pass them to another function that returns `True` if the game is over, `False` if not.

Third-Level Design



Third-Level Design

- At this point, `simOneGame` looks like this:

```
def simOneGame(probA, probB):
    # Simulates a single game or racquetball between A, B
    # RETURNS A's final score, B's final score
    serving = "A"
    scoreA = 0
    scoreB = 0
    while not gameOver(scoreA, scoreB):
```

- Inside the loop, we need to do a single serve. We'll compare a random number to the probability of winning to determine if the server wins the point: `random() < prob`
- The probability we use is determined by who is serving, contained in the variable `serving`.

Third-Level Design

- If A is serving, then we use A's probability, and based on the result of the serve, either update A's score or change the service to B.

```
if serving == "A":
    if random() < probA:
       scoreA += 1
    else:
       serving = "B"
```

Third-Level Design

- Likewise, if it's B's serve, we'll do the same thing with a mirror image of the code.

```
if serving == "A":
    if rnd.random() < probA:    # A wins the serve
       scoreA += 1
    else:                       # A loses service
       serving = "B"
else:
    if rnd.random() < probB:    # B wins the serve
       scoreB += 1
    else:                       # B loses service
       serving = "A"
```

Third-Level Design

```
def simOneGame(probA, probB):
        # Simulates a single game or racquetball between players A, B
        # RETURNS A's final score, B's final score
        serving = "A"
        scoreA = 0
        scoreB = 0
        while not gameOver(scoreA, scoreB):
            if serving == "A":
                if rnd.random() < probA:
                    scoreA += 1
                else:
                    serving = "B"
            else:
                if rnd.random() < probB:
                    scoreB += 1
                else:
                    serving = "A"
        return scoreA, scoreB
```

Finishing Up

There's just one tricky function left, `gameOver`. Here's what we know:

```
def gameOver(a,b):
    # a and b are scores for players in a racquetball game
    # RETURNS true if game is over, false otherwise
```

- According to the rules, the game is over when either player reaches 15 points. We can check for this with the boolean expression: `a==15 or b==15`

Finishing Up

So, the complete code for gameOver looks like this:

```
def gameOver(a,b):
    # a and b are scores for players in a racquetball game
    # RETURNS true if game is over, false otherwise
    return a == 15 or b == 15
```

`printSummary` is equally simple!

```
def printSummary(winsA, winsB):
    # Prints a summary of wins for each player.
    n = winsA + winsB
    print("\nGames simulated:", n)
    print("Wins for A: {0} ({1:0.1%})".format(winsA,winsA/n))
    print("Wins for B: {0} ({1:0.1%})".format(winsB, winsB/n))
```

- Notice % formatting on the output

Summary of the Design Process

- We started at the highest level of our structure chart and worked our way down.
- At each level, we began with a general algorithm and refined it into precise code.
- This process is sometimes referred to as step-wise refinement.

Summary of the Design Process

1. Express the algorithm as a series of smaller problems.
2. Develop an interface for each of the small problems.
3. Detail the algorithm by expressing it in terms of its interfaces with the smaller problems.
4. Repeat the process for each smaller problem.