Beyond programming

CS 199 Computer Science for Beginners
Spring 2009 – Lois Delcambre
Week

Introduction

- We’ve covered:
  - Python data types
  - Python expressions
  - Python statements
- We’ve written approximately 30 programs – that allow students to practice the various data types, expressions, and statements
- It’s time to step back; talk about this …

You've learned a lot of Python

- Look at the reserved word list (p. 30)
- How many of these have you learned?

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	yield
def	finally	in	print	

if, elif, else – for control structures

You've learned a lot of Python

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	yield
def	finally	in	print	

def, return, import, from – used when you define functions and import functions in modules

You've learned a lot of Python

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	yield
def	finally	in	print	

Logical expressions that evaluate to true or false

You've learned a lot of Python

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	yield
def	finally	in	print	

For loop and while loop:

- break – jump out, past the loop
- continue – jump to the bottom of this loop and keep going
- pass – “do nothing” - a placeholder

You've learned a lot of Python

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	yield
def	finally	in	print	

print statement

del and in – used with iterables (list, set, tuple)

Python statements (documentation)

- 6. Simple statements
  - 6.1. Expression statements
  - 6.2. Assignment statements
  - 6.3. The assert statement
  - 6.4. The pass statement
  - 6.5. The del statement
  - 6.6. The print statement
  - 6.7. The return statement
  - 6.8. The yield statement
  - 6.9. The raise statement
  - 6.10. The break statement
  - 6.11. The continue statement
  - 6.12. The import statement
  - 6.13. The global statement
  - 6.14. The exec statement
- 7. Compound statements
  - 7.1. The if statement
  - 7.2. The while statement
  - 7.3. The for statement
  - 7.4. The try statement
  - 7.5. The with statement
  - 7.6. Function definitions
  - 7.7. Class definitions
- 8. Top-level components
  - 8.1. Complete Python programs
  - 8.2. File input
  - 8.3. Interactive input
  - 8.4. Expression input

You’ve used modules

- A module is a file:
  - with your program in it
  - with a function definition in it
  - with a set of (often related) functions in it
- You can:
  - run a module (using Run in IDLE)
  - import a module
- You’ve seen:
  - The string and math modules
  - There are dozens (hundreds?) of others

Many programming languages have the statements that you've learned

- Java has:
  - assignment statements
  - for loops
  - while loops
  - do/while loops
  - break
  - if, else
  - print
  - return

We haven't talked about classes and objects (object-oriented programming or oop)

- Object-oriented programming is one paradigm for programming
- Procedural programming is another paradigm for programming (we have been using Python in a procedural way; we write one Python statement after another)
- Functional programming is another paradigm – where everything is a function (no loop; no assignment statements)

How is a program used?

- It might be run… for a long time … unchanged. Have you heard about the Y2K problem?
- It might be part of something larger – that evolves over time.
- It might be used by others, exactly as it is written – if you put it in a module.
- It might be used and modified by others, e.g., in an open source setting.

Therefore … we need it to be:

- It might be run… for a long time … unchanged. Have you heard about the Y2K problem? Correct – for all inputs
- It might be part of something larger – that evolves over time. Testable(again & again)
- It might be used by others, exactly as it is written – if you put it in a module. Readable & Understandable
- It might be used and modified by others, e.g., in an open source setting. Readable & Understandable

How do we know a program is correct – for all inputs?

- In general, we can't …
- Alan Turing proved in 1936 that it is not possible to write a program that can take as input: a program and an input and decide whether or not the program will finish or will run forever.
- This is called the "halting problem".
- Thus, we say it is "undecidable"

Halting problem (cont.)

- So … don’t expect to generate a bunch of code … and then expect some automatic procedure to figure out whether it’s correct.
  (A general purpose automatic procedure can’t even figure out whether or not a program will finish or not.)
- “Shortest lie in computer science: ‘it works.’”

What about testing?

- Very important.
- You should be systematic in your testing. Figure out your "boundary conditions" the smallest/largest input that your program should work for. Then test:
  - The smallest input
  - A range of typical/normal inputs; variations
  - The largest input
  - An input smaller than the smallest
  - An input larger than the largest

Testing (cont.)

- “Testing can never completely identify all the defects within software.”
- Som software is released for use before it is bug-free (in many cases).
- Some software is life-critical so proper testing is really important.
- [Note that for some programs – it is possible to prove, mathematically, that they meet their specification.]

So … how should we develop software? How do we begin?

- It is generally a bad idea to write one big program (i.e., one big function called main()) that does everything. Why?
- Given what we know so far in Python, how can we decompose a program into smaller pieces (where each piece could be written AND tested) separately? That is, what language feature should we use?

So … how should we develop software? How do we begin?

- It is generally a bad idea to write one big program that does everything. Why?
  - People don't want to read long programs
  - Longer programs are (usually) more complex because some variables may be used in more than one place; thus some parts of the program may depend on others.
- Given what we know so far in Python, …what language feature should we use?
  - Functions! Each function can be written and tested. It doesn't "see" variables except those that are input and those that it returns.

So, how do we decide which functions to write?

- Top-down design
  - Break the big program into a series of small ones. (Assume, for the moment, that you – or someone else – will be able to write the functions that you need.)

Example of top-down design

- Over years of playing racquetball, Denny has noticed a strange quirk in the game. He often competes with players who are just a little bit better than he is. In the process, he always seems to lose the vast majority of matches.
- Susie suggests that they can write a computer program to figure out if it's just the nature of the game (or not).

Racquetball

- To start the game, one of the players puts the ball into play—by serving. *The players then* alternate hitting the ball to keep it in play in 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 server wins the rally, the server earns one point. Players can only score points during their service.
- If the server does NOT win the rally then NO point is awarded and service passes to the other player.
- The first player to reach 15 points wins the game, in the amateur version of scoring. Book uses this.

Specification of a Racquetball Simulation Program

Input

The program first prompts for and gets the service probabilities of the two players (called "Player A" and "Player B"). Then the program prompts for and gets the number of games to be simulated.

Output

The program will provide a series of initial prompts such as the following:

What is the prob. player A wins a serve?

What is the prob. player B wins a serve?

How many games to simulate?

The program will print out a nicely formatted report showing the number of games simulated and the number of wins and winning percentage for each player. Here is an example:

Games Simulated: 500

Wins for A: 268 (53.6%)

Wins for B: 232 (46.4%)

Notes: All inputs are assumed to be legal numeric values, no error or validity checking is required.

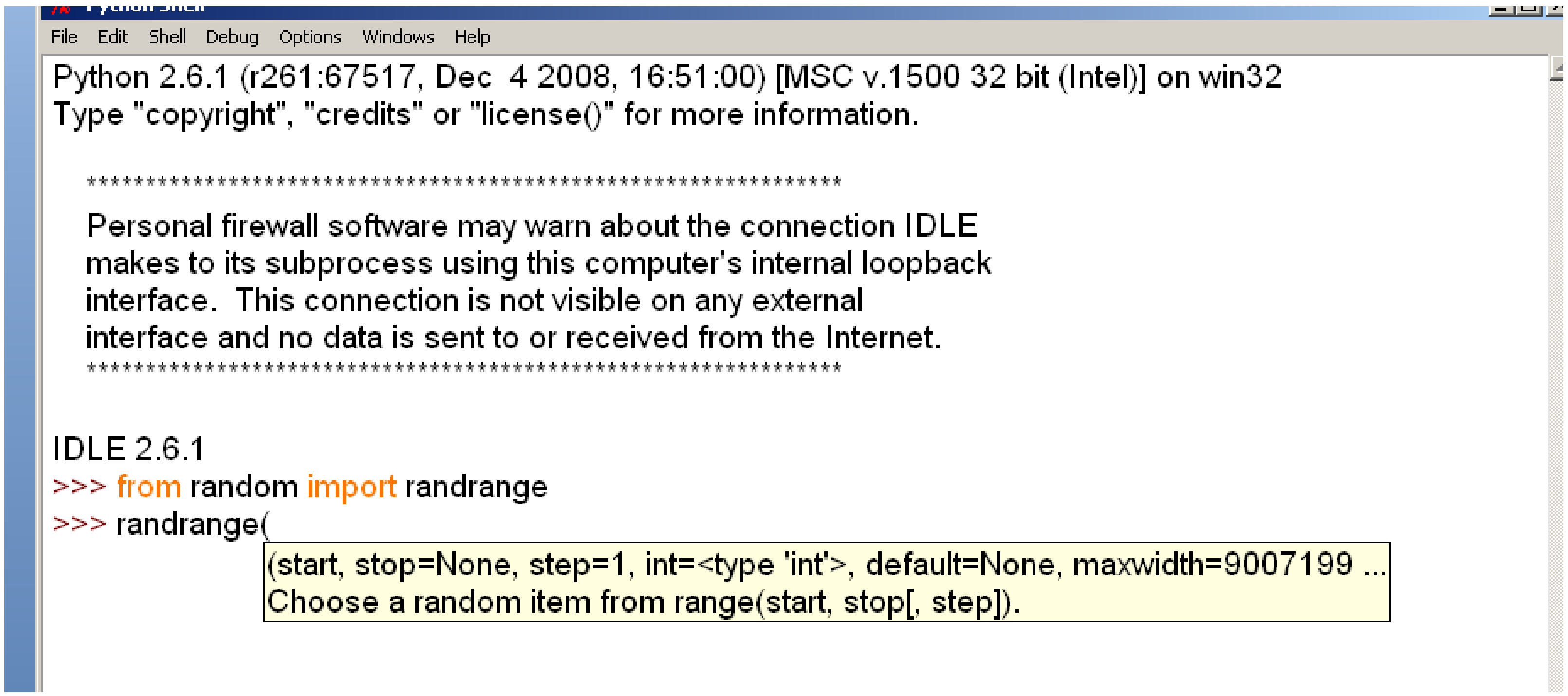
In each simulated game, player A serves first.

How do we figure out who wins/loses a serve?

- We know the probability that player A wins his/her serve. We know the probability that player B wins his/her serve.
- We need to "flip a coin".
- We need to use a random number generator. (Also called a pseudo random number generator.) It returns a value between 0 and 1.

Python has a *random* module

randrange(start, stop, step) returns an integer in the requested range



```
Python 2.6.1 (r261:67517, Dec  4 2008, 16:51:00) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 2.6.1
>>> from random import randrange
>>> randrange(
             (start, stop=None, step=1, int=<type 'int'>, default=None, maxwidth=9007199 ...
             Choose a random item from range(start, stop[, step]).
```

random() returns a floating point number between 0 and 1

```
>>> import random as r
>>> r.random()
0.40664291180330725
>>> r.random()
0.73249446651324879
```

To determine whether player A wins the rally, when they serve:

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

This can be coded in Python like this:

```
if random() < prob for player A:
        score = score + 1
```

Top-down design: get started

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

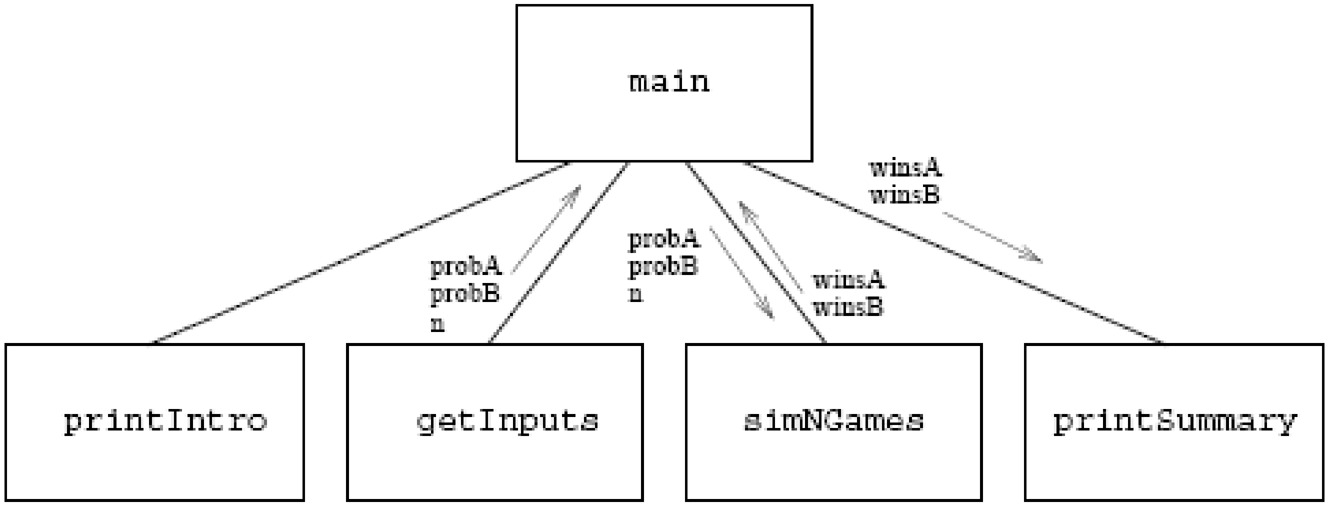
```
def main():
    printInstructions()
    probA, probB, n = getInputs()
    winsA, winsB = simNGames(n, probA, probB)
    printSummary(winsA, winsB)
```

Notice, at this point,

- You may not know how to write each of these functions. Imagine that you will …
- You are choosing the inputs and outputs for each function. This is a very important step! You are defining the interface for the function. (Also, called the signature.)
- You can write the functions:

```
def printSummary(a, b):
    pass
```

Structure chart – for what we have so far



At this point, you might code the functions that you understand … or (instead) focus on the ones you DON'T know how to write. You can continue doing top-down design. Which ones do you know how to write?

Now, consider each function

We know how to write printIntro. Here’s one way:

```
def printIntro():
    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."
```

What about getInputs()?

We know how to write this as well:

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

What about simNGames

(This is the hard one; what shall we do?)

Here's some pseudocode:

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

Code for simNGames

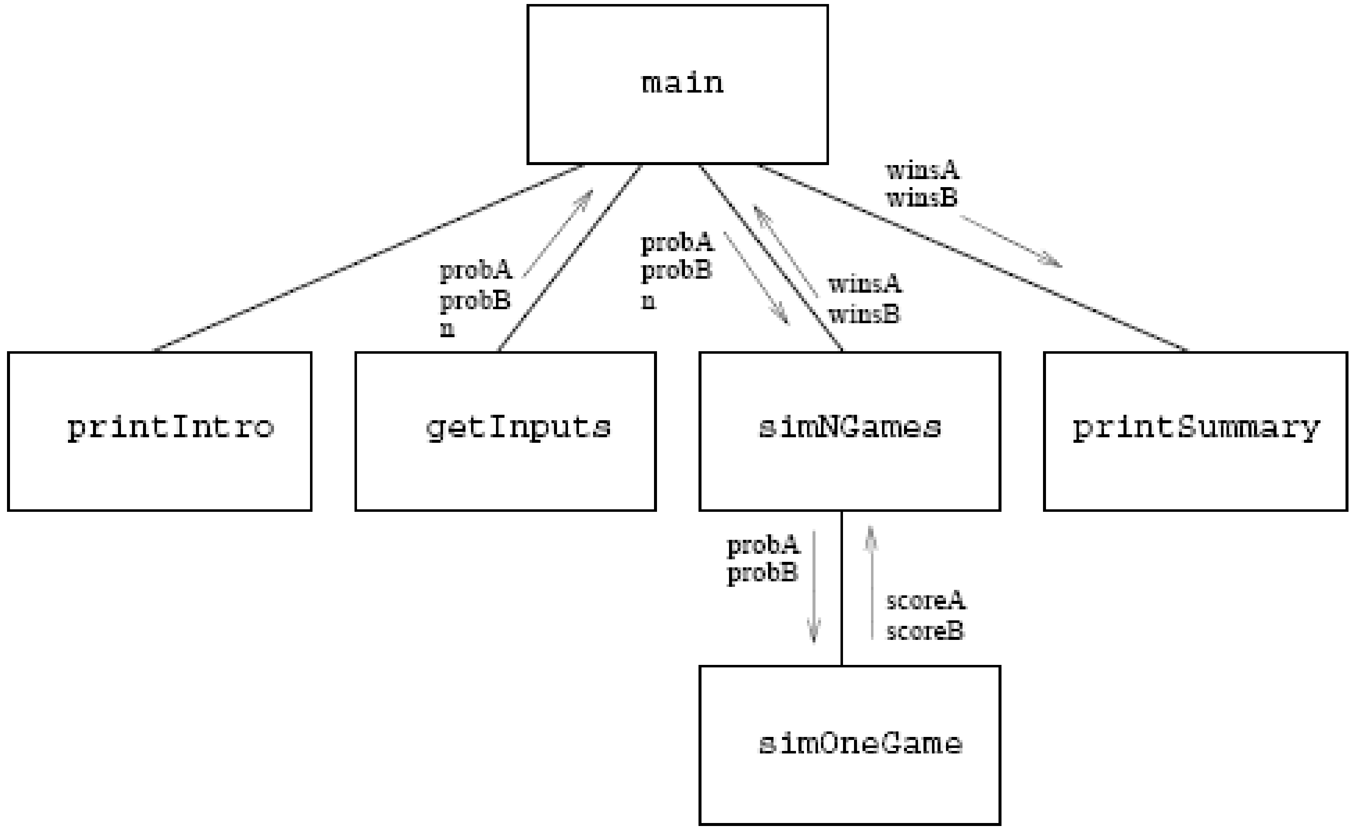
```
def simNGames(n, probA, probB):
# Simulates n games and returns winsA and winsB
        winsA = 0
        winsB = 0
        for i in range(n):
            scoreA, scoreB = simOneGame(probA, probB)
```

We did it again; we invented/imagined a function called simOneGame (it needs to know the probability that player A wins his/her serve and the probability that player B wins his/her serve).

Accumulate wins, based on simOneGame

```
def simNGames(n, probA, probB):
    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
```

Updated Structure Diagram



Continue … top-down design

How do we write simOneGame?

Pseudocode:

```
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
```

Getting started simOneGame

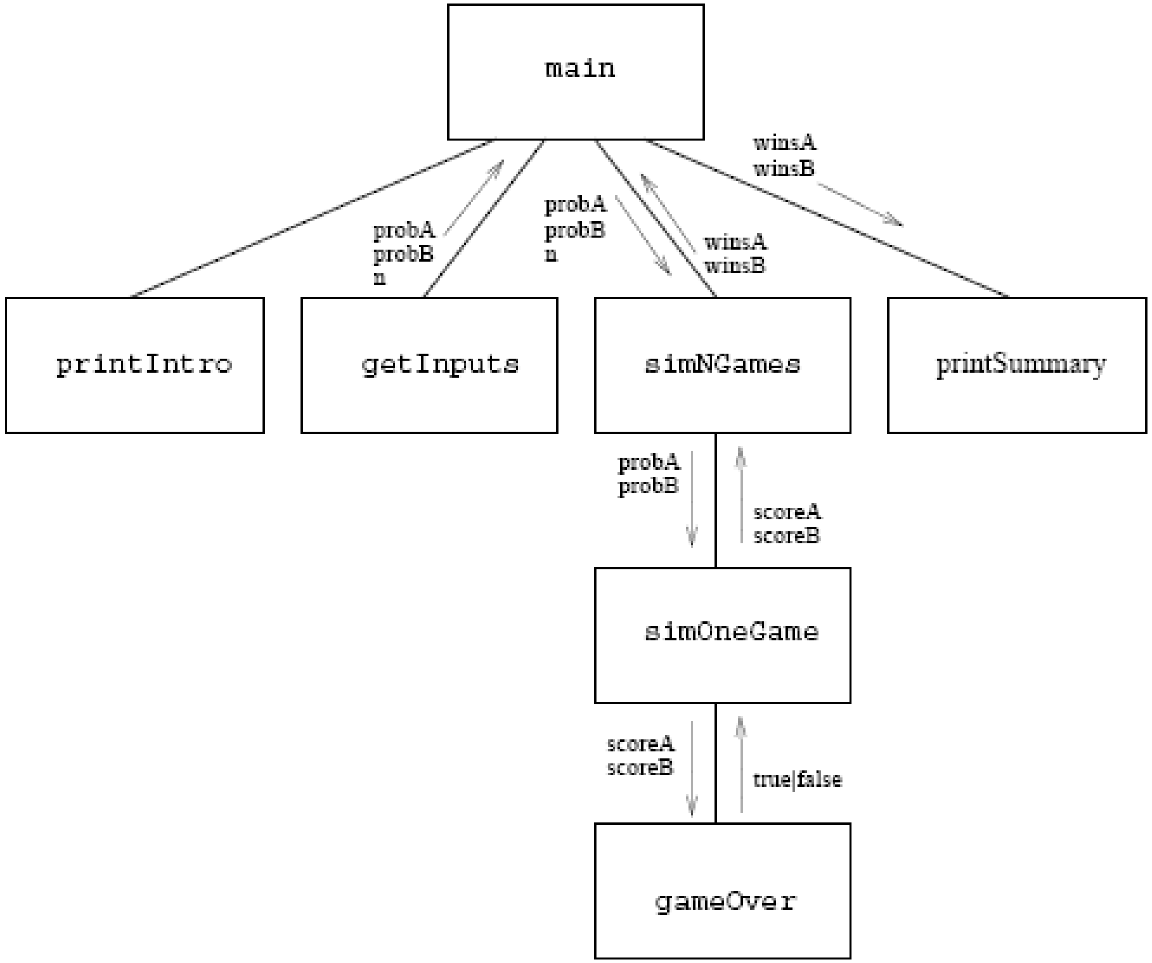
```
def simOneGame(probA, probB):
        scoreA = 0
        scoreB = 0
        serving = "A"
        while <condition>:
```

What is the condition? We need to keep this function running until the game is over. We can invent a function for that:

gameOver returns a Boolean

```
def simOneGame(probA, probB):
        scoreA = 0
        scoreB = 0
        serving = "A"
        while not gameOver(scoreA, scoreB):
```

New structure chart



Complete simOneGame

```
def simOneGame(probA, probB):
    scoreA = 0
    scoreB = 0
    serving = "A"
    while gameNotOver(scoreA, scoreB):
        if serving == "A":
            if random() < probA:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < probB:
                scoreB = scoreB + 1
            else:
                serving = "A"
    return scoreA, scoreB
```

Another function: gameOver

Pseudocode

```
def gameOver(a,b):
    # a and b represent scores for a racquetball game
    # RETURNS true if the game is over, false otherwise.
```

Python code

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

We also need to write the function printSummary; but that's easy.
See pages 280-282 for the complete program, in your book.

Analysis and Design

- Some jobs require that you ONLY do design.
- Some jobs require that you ONLY gather requirements (analysis).
- Some jobs require that you ONLY test software.
- Some jobs require that you do all of these.
- (Not everyone writes code all day long.)