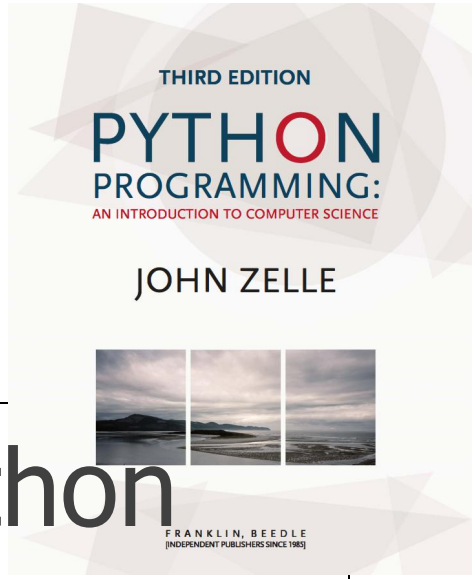


An Introduction to Python Programming

Chapter 9: Simulation and Design

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 and spiral design** techniques in writing complex programs.

Objectives

- To understand **unit-testing** and be able to apply this technique in the implementation and debugging of complex programming.

Simulating Racquetball

- ***Simulation*** can solve **real-world problems** by **modeling** real-world processes to provide otherwise **unobtainable information**.
- In racquetball games, Should players who are *a **little*** **better** win *a **little*** **more often**?

Analysis and Specification

- **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, referred to as a ***rally***.

Analysis and Specification

- the basic outline of the game：
  - 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.

Analysis and Specification

- About 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 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?
```

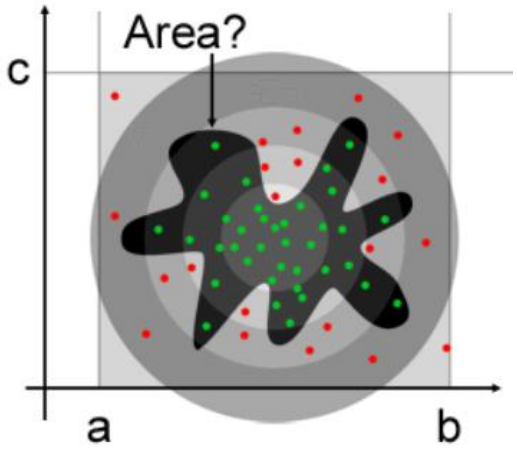
```
Games Simulated: 500
Wins for A: 268 (53.6%)
Wins for B: 232 (46.4%)
```

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).

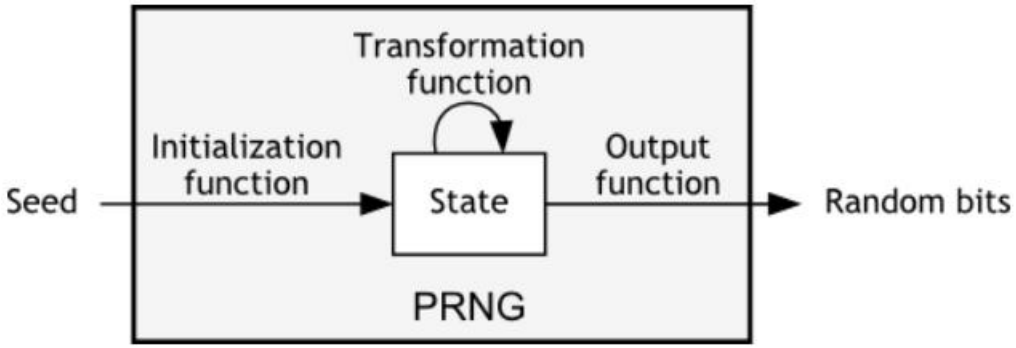
PseudoRandom Numbers

- The **service probability** provides a likelihood to win the game,but our racquetball player should win or lose rallies **randomly.**
- Many simulations require ***Monte Carlo*** **algorithms**: the results depend on **"chance" probabilities**.



PseudoRandom Numbers

- Generate random (*pseudorandom*) numbers:
  - A **pseudorandom number** generator works by starting with a ***seed*** value.
  - The **next time** a random number is required, the current value is fed back into the function to produce a new number.



PseudoRandom Numbers

- Python provides a library module that contains a number of functions for working with 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.

PseudoRandom Numbers

- The two functions: **`randrange`** and **`random`**.
- The **`randrange`** function is used to select a **pseudorandom int** from a given range.
- The **`random`** function is used to generate **pseudorandom floating point** values.

PseudoRandom Numbers

```
>>> from random import randrange
>>> randrange(1,6)
5
>>> randrange(1,6)
1
>>> randrange(1,6)
2
>>> randrange(1,6)
2
>>> randrange(1,6)
5
>>> randrange(1,6)
3
>>> randrange(1,6)
5
>>> randrange(1,6)
2
>>> randrange(1,6)
2
>>> randrange(1,6)
4
>>> randrange(1,6)
4
```

All values will appear an (approximately) equal number of times.

PseudoRandom Numbers

```
>>> from random import random
>>> random()
0.27743857840227626
>>> random()
0.20141560594819174
>>> random()
0.3130287693394088
>>> random()
0.24561243196158378
>>> random()
0.2311337802999267
>>> random()
0.9369582103532702
>>> random()
0.39378316015486203
>>> random()
0.32161492697535254
```

PseudoRandom Numbers

- The racquetball simulation makes use of the **`random`** function to determine if a player has won a serve.
  - Suppose a player's service probability is 70%, or 0.70.
  - Suppose we generate a random number between 0 and 1. Exactly 70% of the interval 0..1 is to the left of 0.7.
  - The **＝ goes on the upper end** since the random number generator can produce a 0 but not a 1.

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

Top-Down Design

- In ***top-down design***, a complex problem is **expressed** as a solution in terms of smaller, simpler problems.
- Typically a program uses the ***input***, ***process***, ***output*** pattern.

```
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
```

Top-Down Design:Top-Level Design

- For the **racquetball simulation**，the complete **main** program:

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

Top-Down Design:Separation of Concerns

- The **original problem** has now been decomposed into four independent tasks:

  `printIntro`

  `getInputs`

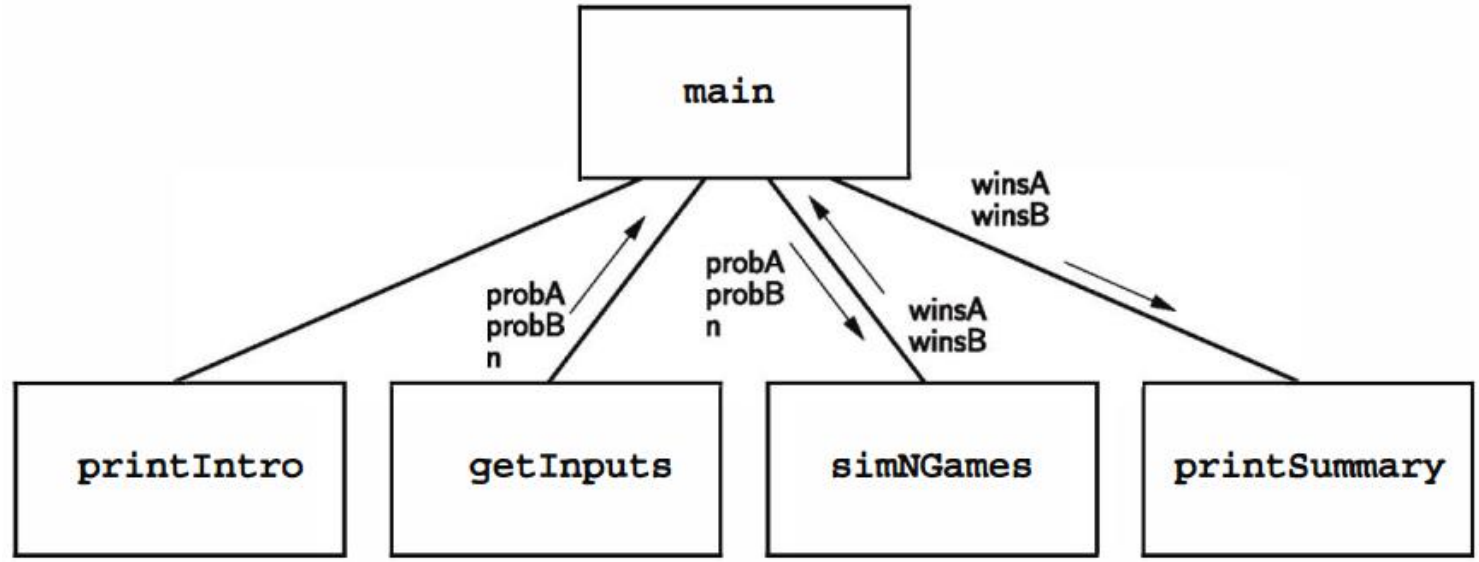
  `simNGames`

  `printSummary`

- The **name, parameters, and expected return values** of these functions have been **specified**. This information is known as the ***interface*** or ***signature*** of the function.

Top-Down Design:Separation of Concerns

- In the following ***structure chart*** (or ***module hierarchy***), each component in the design is a rectangle.



Top-Down Design: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***.

Top-Down Design:Second-Level Design

- The **`printIntro`** function :

```
def printIntro():
    print("This program simulates a game of racquetball between two")
    print('players called "A" and "B". The ability 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.")
```

Top-Down Design:Second-Level Design

- The **`getInputs`** function:

```
def getInputs():
    # Returns the three simulation parameters probA, probB and n
    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
```

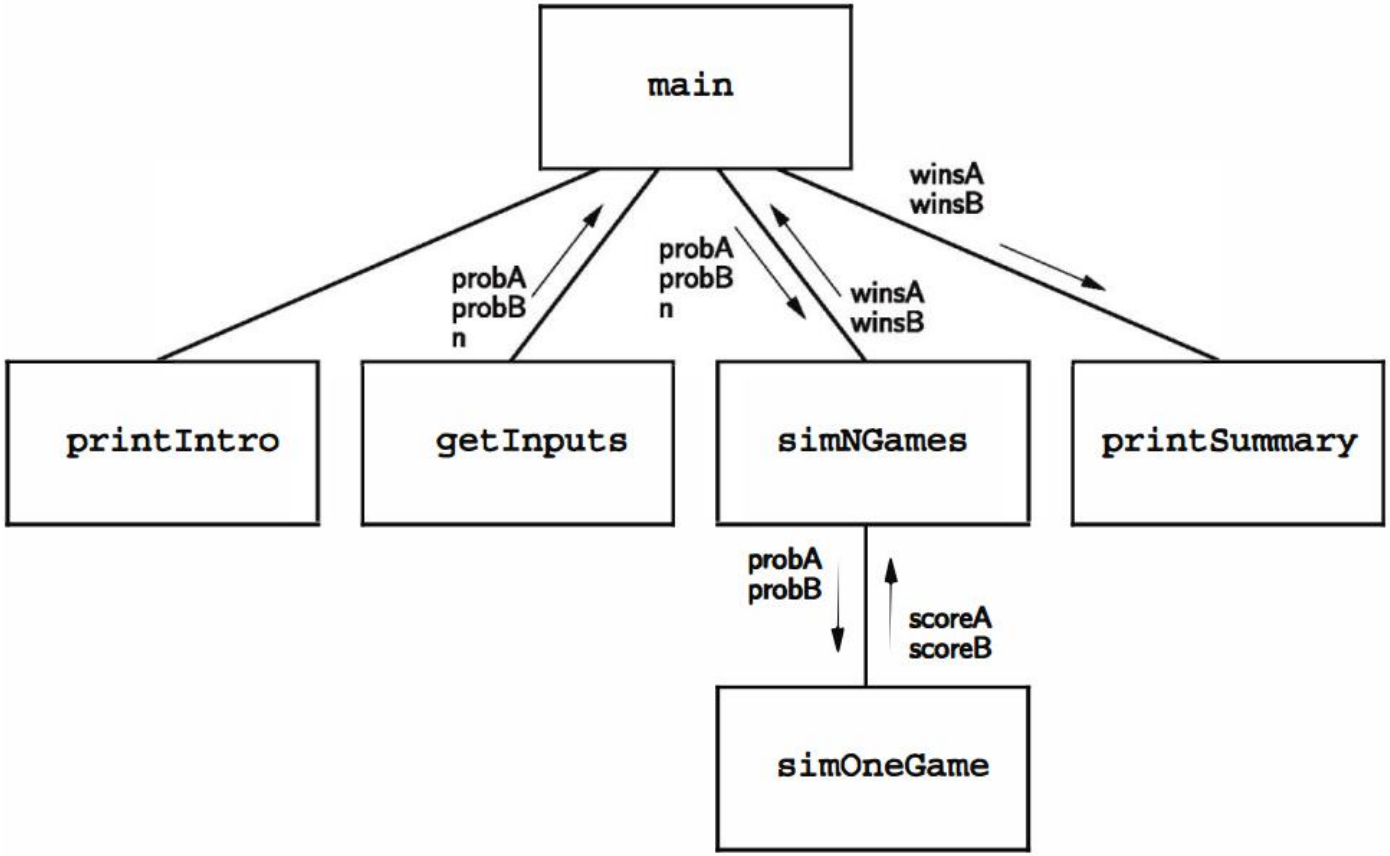
Top-Down Design:Designing simNGames

```
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
```

Top-Down Design:Designing simNGames

```
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
```

Top-Down Design:Designing simNGames

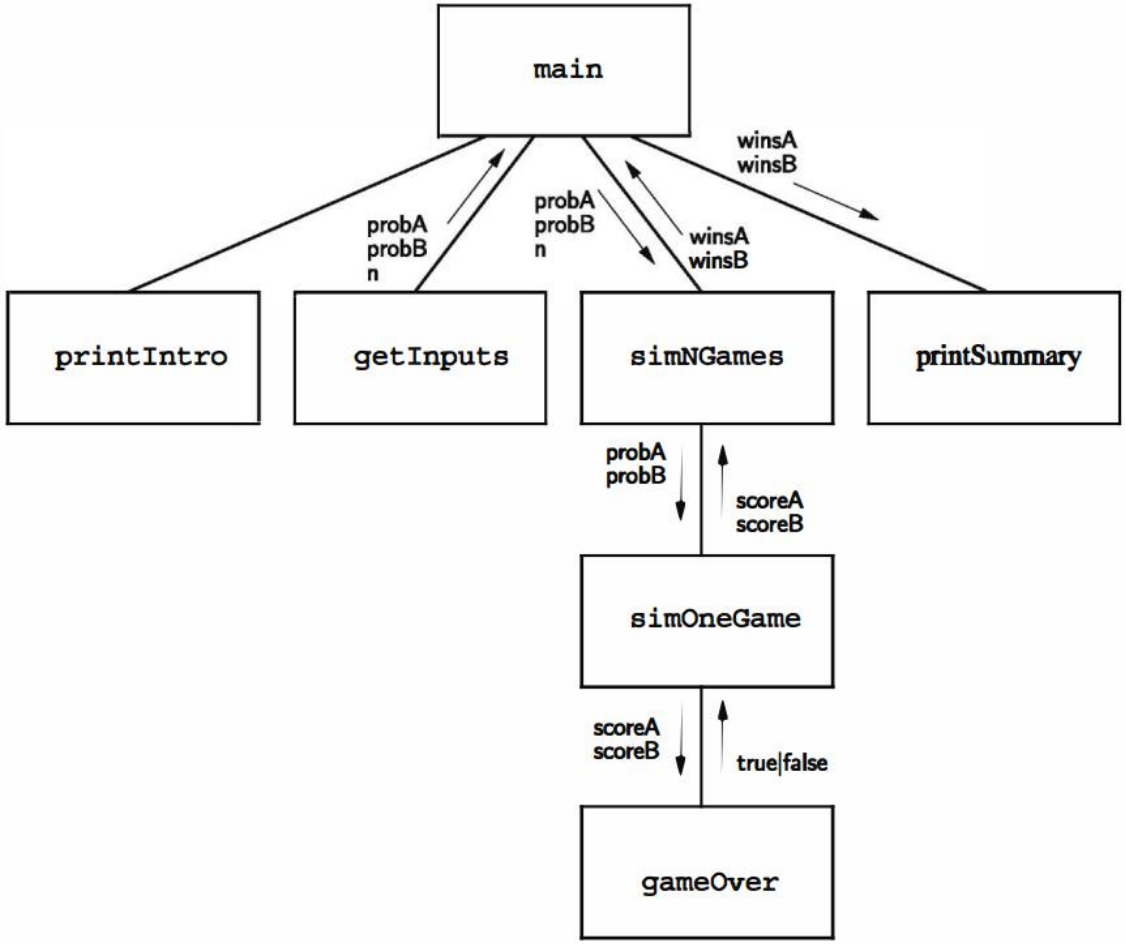


Top-Down Design:Third-Level Design

- The logic of the racquetball rules lies in the function of **`simOneGame`**.

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

Top-Down Design:Third-Level Design



Top-Down Design:Third-Level Design

- We'll **compare a random number** to t**he provided probability** to determine if the server wins the point. (`random() < prob`)
- So:

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

Top-Down Design:Finishing Up

- There's just one tricky function left, **gameOve.**

```
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
```

- Let's go to P298 to see the complete program!
  - The type specifier **%** :

```
>>> print(format(0.5,'0.1%'))
50.0%
>>> print(format(0.4234,'3.1%'))
42.3%
```

Top-Down Design:Summary of the Design Process

- This process is sometimes referred to as ***step-wise refinement***.
  - Express the algorithm as **a series of smaller problems**.
  - Develop an **interface** for each of the small problems.
  - **Detail the algorithm** by expressing it in terms of its interfaces with the smaller problems.
  - **Repeat** the process for each smaller problem.

Bottom-Up Implementation

- Even though we've been careful with the design, there's **no guarantee** we haven't introduced some **silly errors**.
- Implementation is best done **in small pieces**.

Bottom-Up Implementation: Unit Testing

- A good way to approach the implementation of a **modestly sized program** is to start at the **lowest levels** ,work your way up, **testing each component** as you complete it.

```
>>> gameOver(0,0)
False
>>> gameOver(5,10)
False
>>> gameOver(15,3)
True
>>> gameOver(3,15)
True
```

selected test data that tries all the important cases

Bottom-Up Implementation:

Unit Testing

- Having confidence that **`gameOver`** is functioning correctly, we can go on to **`simOneGame`**.

```
>>> simOneGame(.5,.5)
(13, 15)
>>> simOneGame(.5,.5)
(15, 11)
>>> simOneGame(.3,.3)
(15, 11)
>>> simOneGame(.3,.3)
(11, 15)
>>> simOneGame(.4,.9)
(4, 15)
>>> simOneGame(.4,.9)
(1, 15)
```

```
>>> simOneGame(.9,.4)
(15, 3)
>>> simOneGame(.9,.4)
(15, 0)
>>> simOneGame(.4,.6)
(9, 15)
>>> simOneGame(.4,.6)
(6, 15)
```

Bottom-Up Implementation: Simulation Results

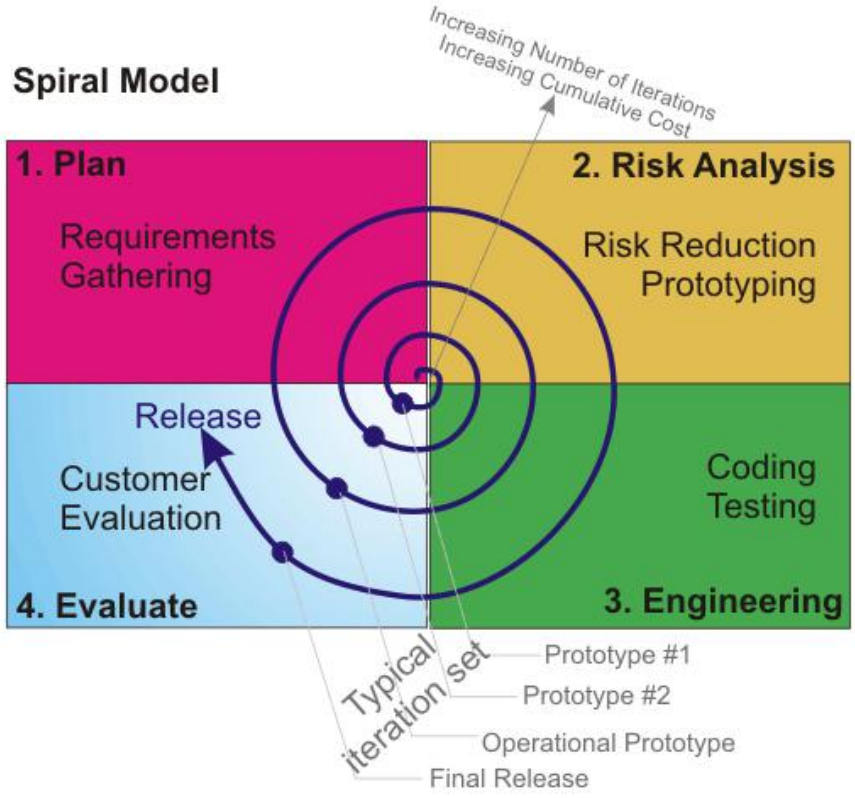
- Suppose Denny wins about 60% of his serves and his opponent is 5% better. How often should Denny win?

```
This program simulates a game of racquetball between two
players called "A" and "B".  The ability of each player is
indicated by a probability (a number between 0 and 1) that
the player wins the point when serving. Player A always
has the first serve.

What is the prob. player A wins a serve? .65
What is the prob. player B wins a serve? .6
How many games to simulate? 5000

Games simulated: 5000
Wins for A: 3360 (67.2%)
Wins for B: 1640 (32.8%)
```

Other Design Techniques：Prototyping and Spiral Development

- Another program development is to start with **a simple version** of a program, then gradually add features until it meets the full specification.
- This initial stripped-down version is called a ***prototype***.
- Prototyping often leads to a ***spiral*** development process.

Other Design Techniques：Prototyping and Spiral Development



Other Design Techniques：Prototyping and Spiral Development

- How could the **racquetball simulation** been done using **spiral development**?
  - Write a prototype where you assume there's a **50-50 chance** of winning any given point, playing **30 rallies**.
  - Add on to the prototype **in stages**： awarding of points, change of service, differing probabilities, etc.

Other Design Techniques：Prototyping and Spiral Development

```
def simOneGame():
    scoreA = 0
    scoreB = 0
    serving = "A"
    for i in range(30):
        if serving == "A":
            if random() < .5:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < .5:
                scoreB = scoreB + 1
            else:
                serving = "A"
        print(scoreA, scoreB)

if __name__ == '__main__': simOneGame()
```

Other Design Techniques：Prototyping and Spiral Development

- The program could be enhanced in phases:
  - **Phase 1: Initial prototype**. Play 30 rallies where the server always has a 50% chance of winning. Print out the scores after each server.
  - **Phase 2:** Add two parameters to represent **different probabilities** for the two players.
  - **Phase 3:** Play the game until one of the players **reaches 15 points**. At this point, we have a working simulation of a single game.
  - **Phase 4:** Expand to play **multiple games**. The output is the count of games won by each player.
  - **Phase 5:** Build the **complete program**. Add interactive inputs and a nicely formatted report of the results.

Other Design Techniques：
The Art of Design

- Spiral development and top-down design are complementary approaches.
- Good design is as much creative process as science, and as such, there are no hard and fast rules.
- You have to learn for yourself through experience. *Practice!*