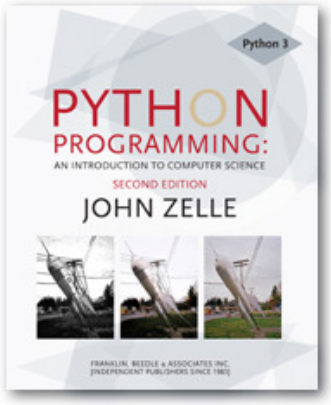
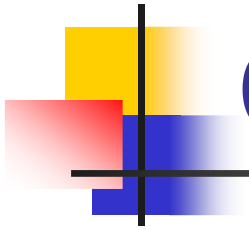
Python Programming: An Introduction to Computer Science

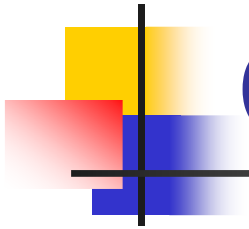


Chapter 9
Simulation and Design



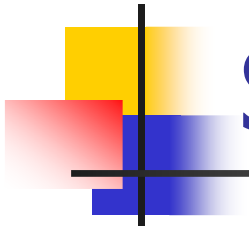
Objectives

- 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.



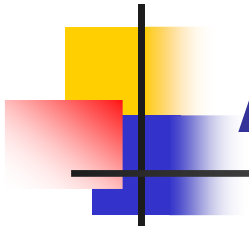
Objectives

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



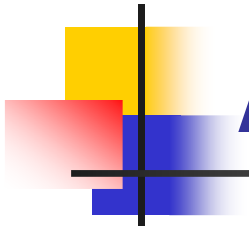
Simulating Racquetball

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



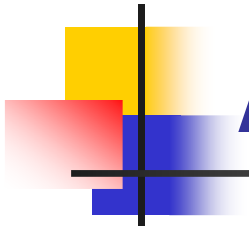
A Simulation Problem

- Denny Dibblebit 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.



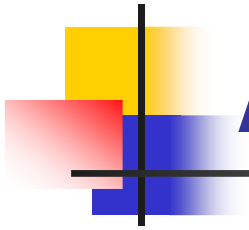
Analysis and Specification

- 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*. The rally ends when one player fails to hit a legal shot.



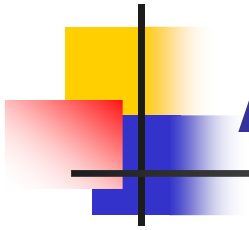
Analysis and Specification

- 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.



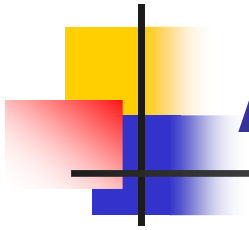
Analysis and Specification

- 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.



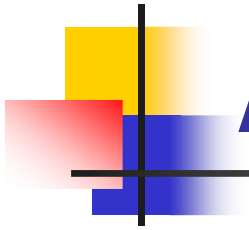
Analysis and Specification

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

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

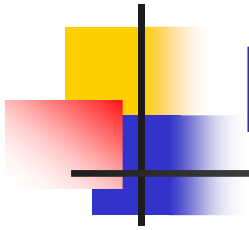
- The program then prints out a nicely formatted report showing the number of games simulated and the number of wins and the winning percentage for each player.

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



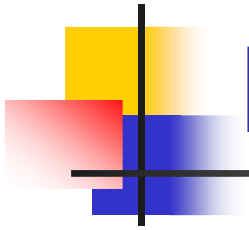
Analysis and Specification

- **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.



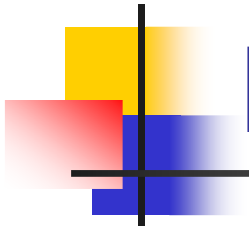
PseudoRandom Numbers

- 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).



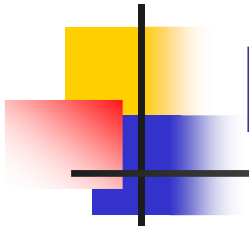
PseudoRandom Numbers

- 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.
- Do you remember the chaos program from chapter 1? The apparent randomness of the result came from repeatedly applying a function to generate a sequence of numbers.



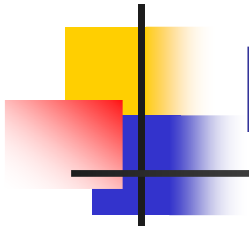
PseudoRandom Numbers

- A similar approach is used to generate random (technically *pseudorandom*) numbers.
- A pseudorandom number generator works by starting with a *seed* value. 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.



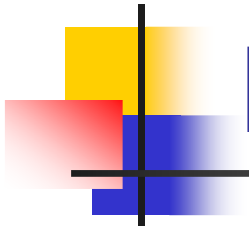
PseudoRandom Numbers

- 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 a library module that contains a number of functions for working with pseudorandom numbers.



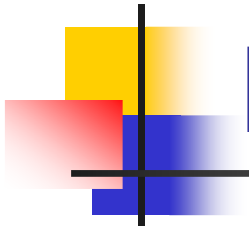
PseudoRandom Numbers

- 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 two functions of greatest interest are `randrange` and `random`.



PseudoRandom Numbers

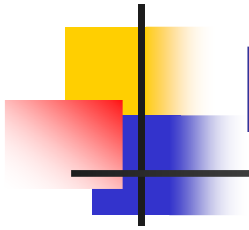
- The `randrange` function is used to select a pseudorandom int from a given range.
- The syntax is similar to that of the `range` command.
- `randrange(1,6)` returns some number from `[1,2,3,4,5]` and `randrange(5,105,5)` returns a multiple of 5 between 5 and 100, inclusive.
- Ranges go up to, but don't include, the stopping value.



PseudoRandom Numbers

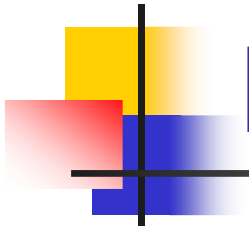
- Each call to `randrange` generates a new pseudorandom int.

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



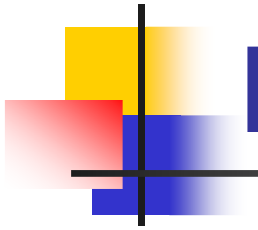
PseudoRandom Numbers

- The value 5 comes up over half the time, demonstrating the probabilistic nature of random numbers.
- Over time, this function will produce a uniform distribution, which means that all values will appear an approximately equal number of times.



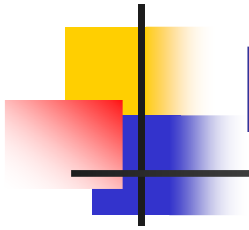
PseudoRandom Numbers

- The `random` function is used to generate pseudorandom floating point values.
- It takes no parameters and returns values uniformly distributed between 0 and 1 (including 0 but excluding 1).



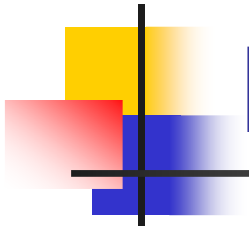
PseudoRandom Numbers

```
>>> from random import random
>>> random()
0.7943280091289881 6
>>> random()
0.0004985861940545177 6
>>> random()
0.1341231400816878
>>> random()
0.9872455453536165 3
>>> random()
0.2142942417503219 7
>>> random()
0.2390358371212714 1
>>> random()
0.7291832884340891 9
```



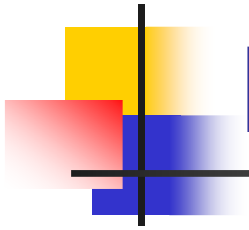
PseudoRandom Numbers

- 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.
- ```
  if <player wins serve>:
      score = score + 1
  ```
- We need to insert a probabilistic function that will succeed 70% of the time.
  ```



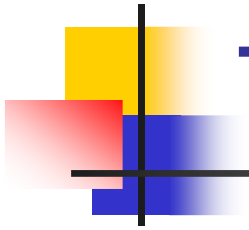
PseudoRandom Numbers

- 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 upper end since the random number generator can produce a 0 but not a 1.)



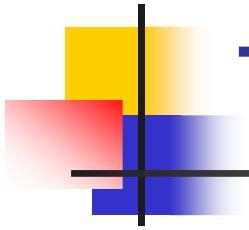
PseudoRandom Numbers

- If `prob` represents the probability of winning the server, the condition `random() < prob` will succeed with the correct probability.
-
```
if random() < prob:
    score = score + 1
```



Top-Down Design

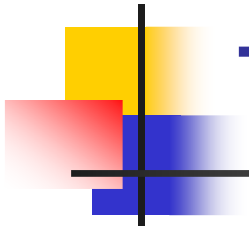
- In *top-down design*, a complex problem is expressed as a solution in terms of smaller, simpler problems.
- These smaller problems are then solved by expressing them in terms of smaller, simpler problems.
- This continues until the problems are trivial to solve. The little pieces are then put back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



Top-Level Design

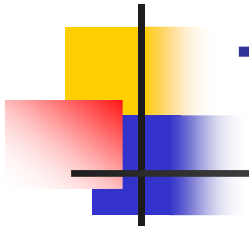
- Typically a program uses the *input*, *process*, *output* pattern.
- The algorithm for the racquetball simulation:

```
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
```



Top-Level Design

- Is this design too high level? 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.

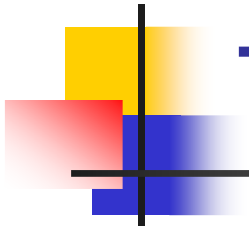


Top-Level Design

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

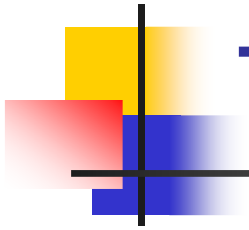
```
def main():
    printIntro()
```

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



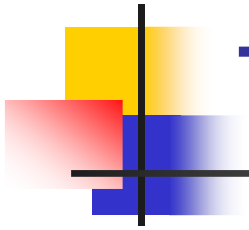
Top-Level Design

- 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()
  ```
  ```



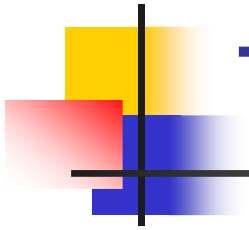
Top-Level Design

- Now we need to simulate *n* games of racquetball using the values of `probA` and `probB`.
- How would we do that? We can put off writing this code by putting it into a function, `simNGames`, and add a call to this function in `main`.



Top-Level 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.

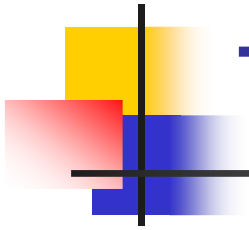


Top-Level Design

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

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

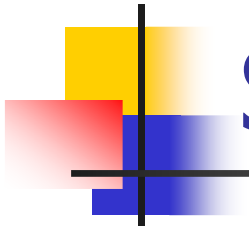
- What information would you need to be able to produce the output from the program?
- You'd need to know how many wins there were for each player – these will be the inputs to the next function.



Top-Level Design

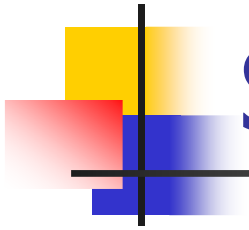
- The complete main program:

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



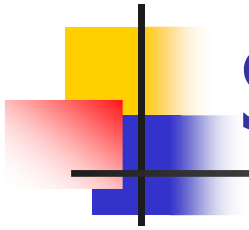
Separation of Concerns

- 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.



Separation of Concerns

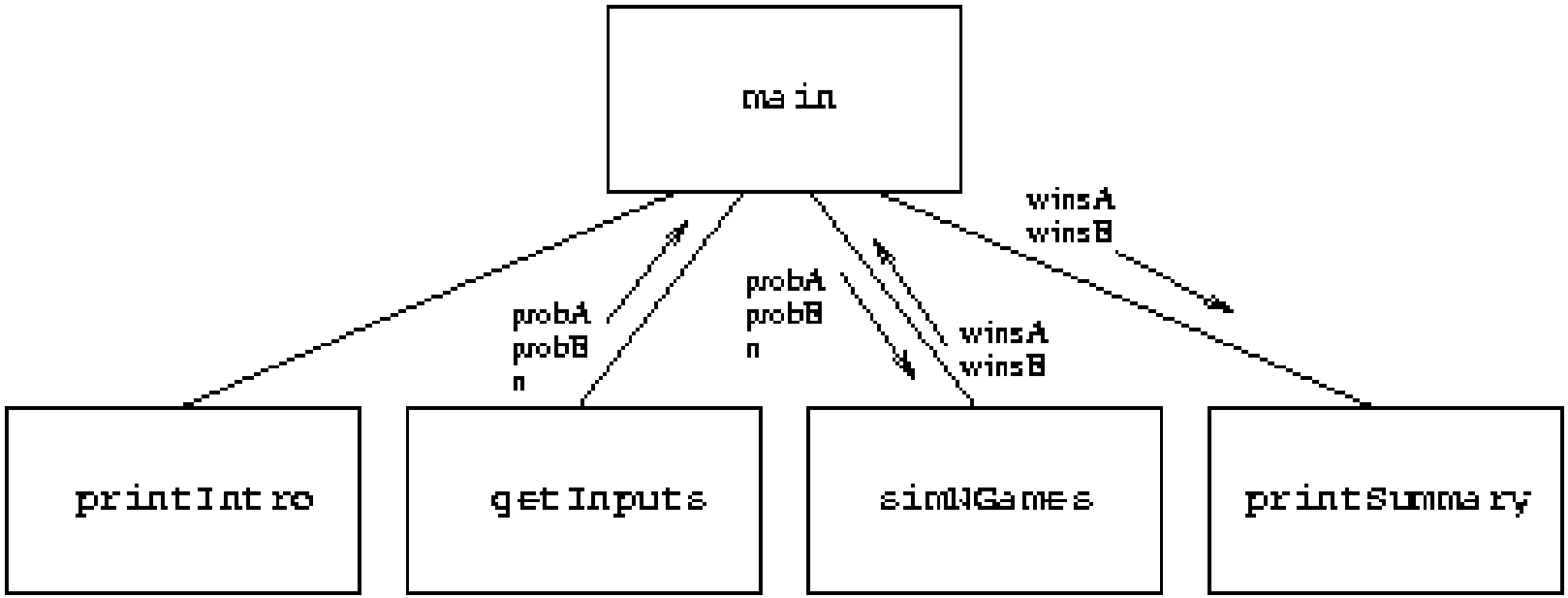
- Having this information (the *signatures*), allows us to work on each of these pieces indepently.
- For example, as far as `main` is concerned, *how* `simNGames` works is not a concern as long as passing the number of games and player probabilities to `simNGames` causes it to return the correct number of wins for each player.

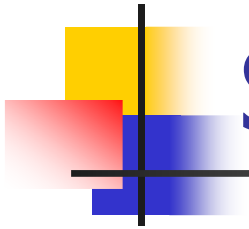


Separation of Concerns

- 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.

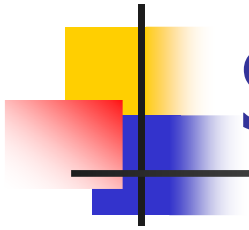
Separation of Concerns





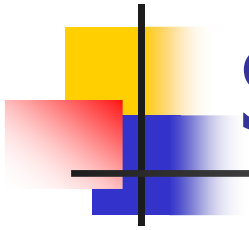
Separation of Concerns

- 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

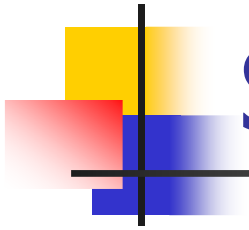
- The next step is to repeat the process for each of the modules defined in the previous step!
- The `printIntro` function should print an introduction to the program. The code for this is straightforward.



Second-Level Design

```
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 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.\n“)
```

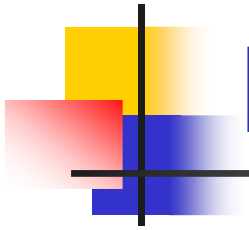
- In the second line, since we wanted double quotes around A and B, the string is enclosed in apostrophes.
- Since there are no new functions, there are no changes to the structure chart.



Second-Level Design

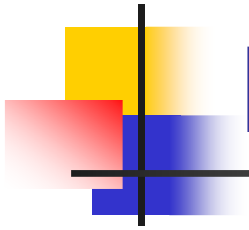
- In `getInputs`, we prompt for and get three values, which are returned to the main program.

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



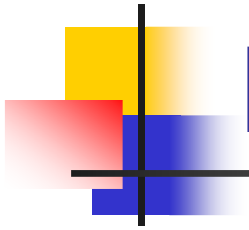
Designing simNGames

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



Designing simNGames

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



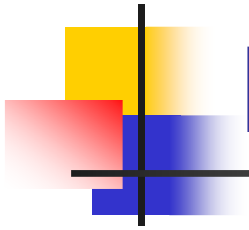
Designing simNGames

- We already have the function signature:

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

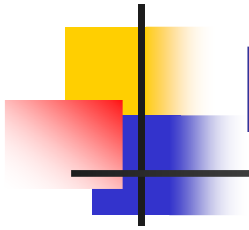
- With this information, it's easy to get started!

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



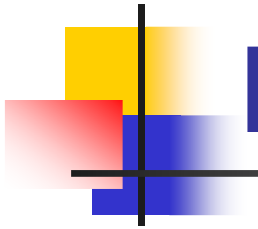
Designing simNGames

- 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

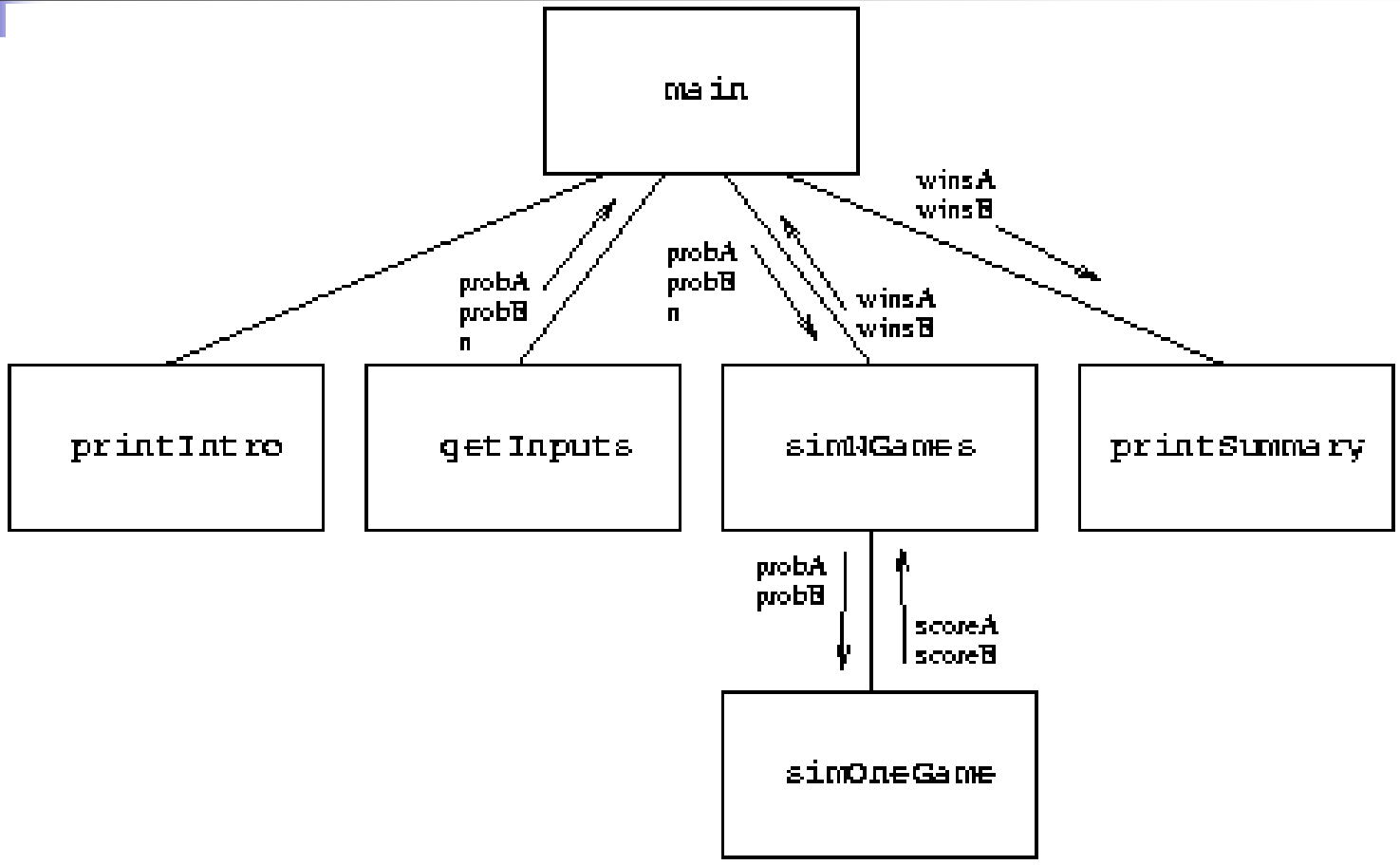
- We need to know who won the game. How can we get this information?
- The easiest way is to pass back the final score.
- The player with the higher score wins and gets their accumulator incremented by one.

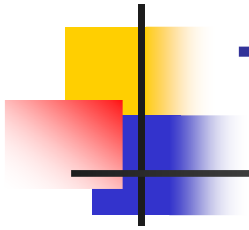


Designing simNGames

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

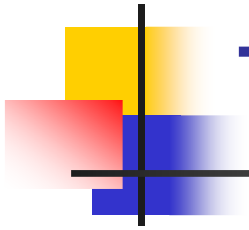
Designing simNGames





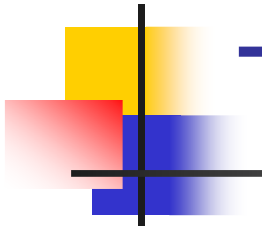
Third-Level Design

- 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.



Third-Level Design

- 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".



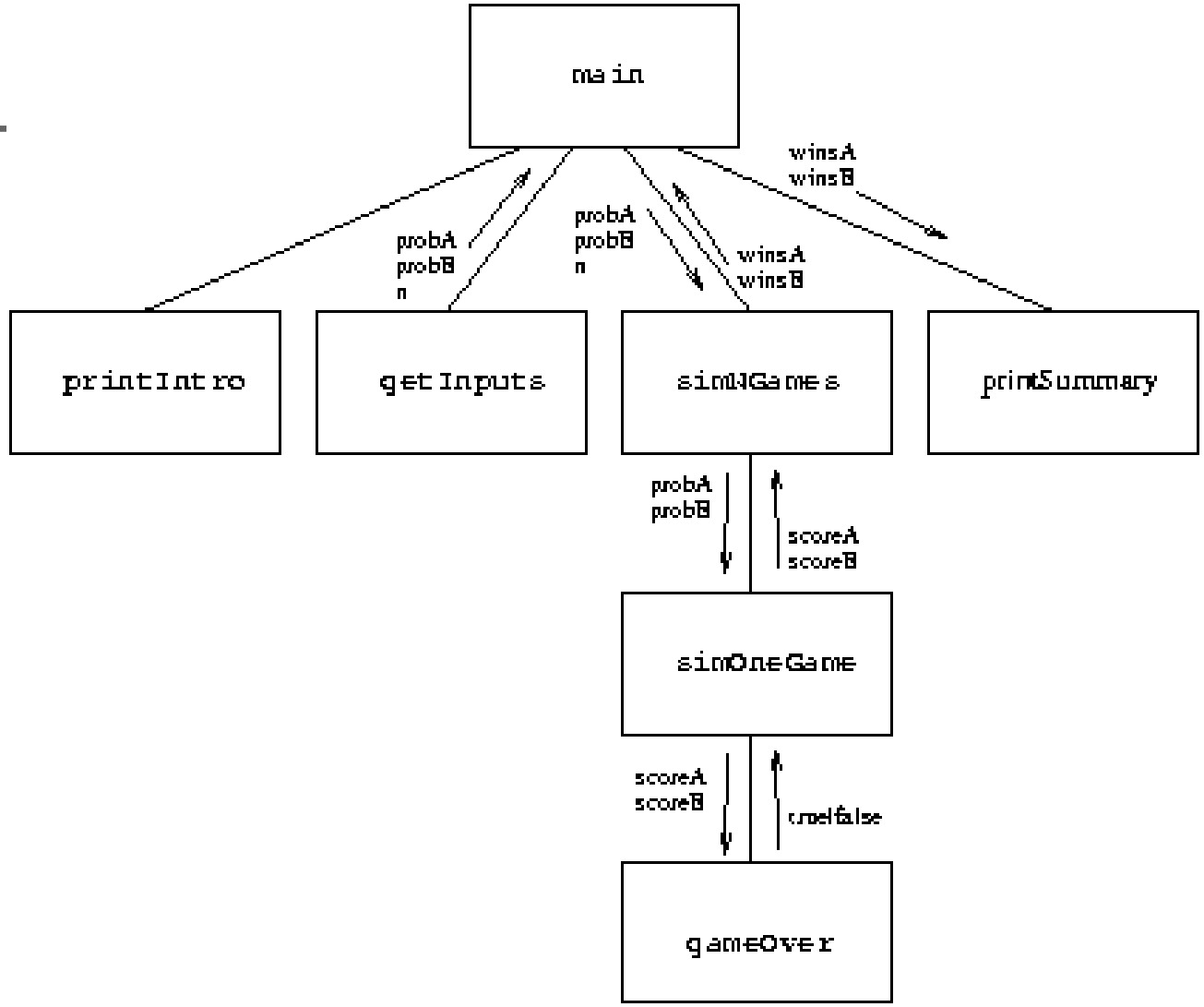
Third-Level Design

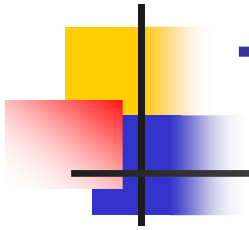
- ```
  Initialize scores to 0
  Set serving to "A"
  Loop while game is not over:
     Simulate one serve of whichever player is serving
     update the status of the game
  Return scores
  ```

- ```
  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

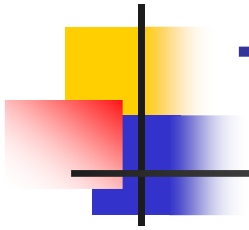




Third-Level Design

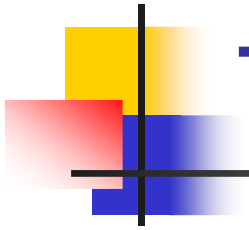
- At this point, `simOneGame` looks like this:

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



Third-Level Design

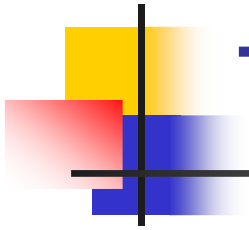
- Inside the loop, we need to do a single serve. We'll compare a random number to the provided probability to determine if the server wins the point (`random() < prob`).
- The probability we use is determined by whom is serving, contained in the variable `serving`.



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.

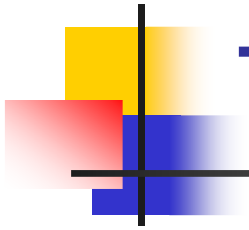
```
if serving == "A":
    if random() < probA:
       scoreA = scoreA + 1
    else:
       serving = "B"
```



Third-Level Design

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

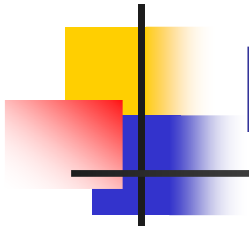
```
if serving == "A":
    if random() < probA:
       scoreA = scoreA + 1
    else:
       serving = "B“
else:
    if random() < probB:
       scoreB = scoreB + 1
    else:
       serving = "A"
```



Third-Level Design

Putting the function together:

```
def simOneGame(probA, probB):
    # Simulates a single game or racquetball between players A and 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 random() < probA:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < probB:
                scoreB = scoreB + 1
            else:
                serving = "A"
    return scoreA, scoreB
```

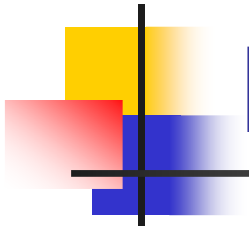


Finishing Up

- 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: `a==15 or b==15`



Finishing Up

- So, the complete code for `gameOver` looks like this:

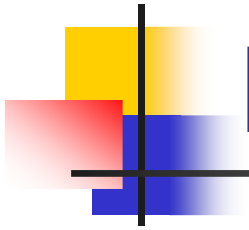
```
def gameOver(a,b):
    # a and b are scores for players in a racquetball game
    # RETURNS true if game is over, false otherwise
    return a == 15 or b == 15
```

- `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)
```

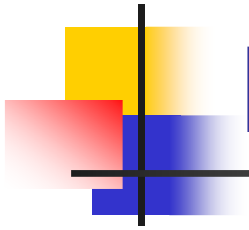
- Notice % formatting on the output

Summary of the Design Process

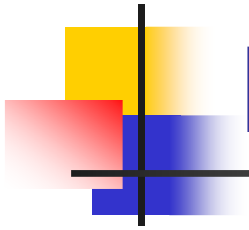


- We started at the highest level of our structure chart and worked our way down.
- At each level, we began with a general algorithm and refined it into precise code.
- This process is sometimes referred to as *step-wise refinement*.

Summary of the Design Process

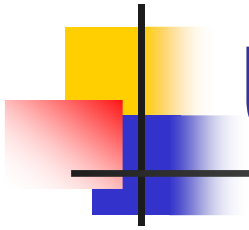


1. Express the algorithm as a series of smaller problems.
2. Develop an interface for each of the small problems.
3. Detail the algorithm by expressing it in terms of its interfaces with the smaller problems.
4. Repeat the process for each smaller problem.



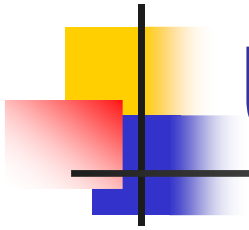
Bottom-Up Implementation

- 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.



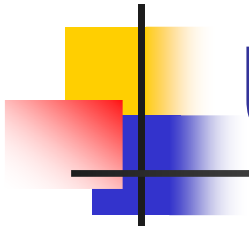
Unit Testing

- A good way to systematically test the implementation of a modestly sized program is to start at the lowest levels of the structure, testing each component as it's completed.
- For example, we can import our program and execute various routines/functions to ensure they work properly.



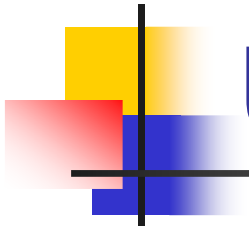
Unit Testing

- We could start with the `gameOver` function.
- ```
  >>> import rball
  >>> rball.gameOver(0,0)
  False
  >>> rball.gameOver(5,10)
  False
  >>> rball.gameOver(15,3)
  True
  >>> rball.gameOver(3,15)
  True
  ```
  ```



Unit Testing

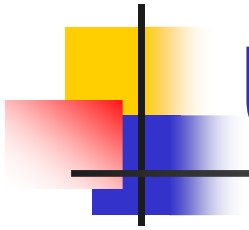
- Notice that we've tested `gameOver` for all the important cases.
  - We gave it 0, 0 as inputs to simulate the first time the function will be called.
  - The second test is in the middle of the game, and the function correctly reports that the game is not yet over.
  - The last two cases test to see what is reported when either player has won.



Unit Testing

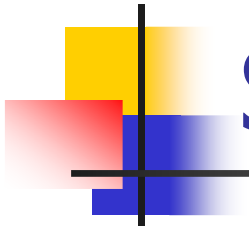
- Now that we see that `gameOver` is working, we can go on to `simOneGame`.

```
>>> simOneGame(.5, .5)
(11, 15)
>>> simOneGame(.5, .5)
(13, 15)
>>> simOneGame(.3, .3)
(11, 15)
>>> simOneGame(.3, .3)
(15, 4)
>>> simOneGame(.4, .9)
(2, 15)
>>> simOneGame(.4, .9)
(1, 15)
>>> simOneGame(.9, .4)
(15, 0)
>>> simOneGame(.9, .4)
(15, 0)
>>> simOneGame(.4, .6)
(10, 15)
>>> simOneGame(.4, .6)
(9, 15)
```



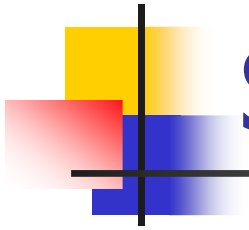
Unit Testing

- When the probabilities are equal, the scores aren't that far apart.
- When the probabilities are farther apart, the game is a rout.
- Testing each component in this manner is called *unit testing*.
- Testing each function independently makes it easier to spot errors, and should make testing the entire program go more smoothly.



Simulation Results

- Is it the nature of racquetball that small differences in ability lead to large differences in final score?
- Suppose Denny wins about 60% of his serves and his opponent is 5% better. How often should Denny win?
- Let's do a sample run where Denny's opponent serves first.



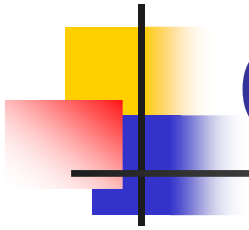
Simulation Results

```
This program simulates a game of racquetball between two
players called "A" and "B".  The abilities 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: 3329 (66.6%)
Wins for B: 1671 (33.4%)
```

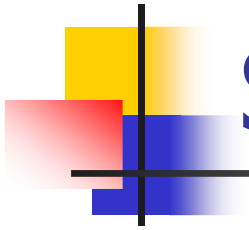
- With this small difference in ability , Denny will win only 1 in 3 games!



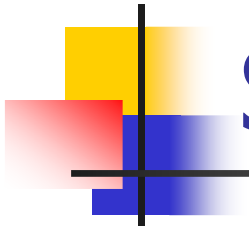
Other Design Techniques

- Top-down design is not the only way to create a program!

Prototyping and Spiral Development



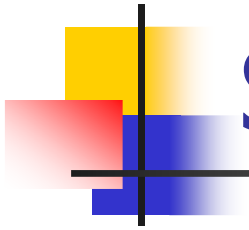
- Another approach to program development is to start with a simple version of a program, and then gradually add features until it meets the full specification.
- This initial stripped-down version is called a *prototype*.



Prototyping and Spiral Development

- Prototyping often leads to a *spiral* development process.
- Rather than taking the entire problem and proceeding through specification, design, implementation, and testing, we first design, implement, and test a prototype. We take many mini-cycles through the development process as the prototype is incrementally expanded into the final program.

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, including awarding of points, change of service, differing probabilities, etc.

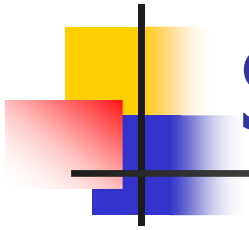
Prototyping and Spiral Development

```
from random import random

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)
```

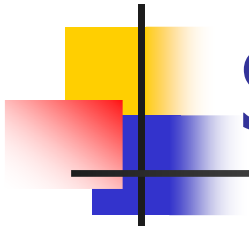
```
>>> simOneGame()
0 0
0 1
0 1
…
2 7
2 8
2 8
3 8
3 8
3 8
3 8
3 8
3 9
3 9
4 9
5 9
```

Prototyping and Spiral Development



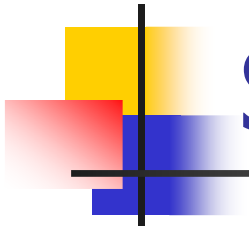
- The program could be enhanced in phases:
  - **Phase 1:** Initial prototype. Play 30 rallies where the server always has a 50% chance of winning. Print out the scores after each server.
  - **Phase 2:** Add two parameters to represent different probabilities for the two players.

Prototyping and Spiral Development

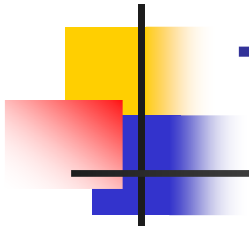


- **Phase 3:** Play the game until one of the players reaches 15 points. At this point, we have a working simulation of a single game.
- **Phase 4:** Expand to play multiple games. The output is the count of games won by each player.
- **Phase 5:** Build the complete program. Add interactive inputs and a nicely formatted report of the results.

Prototyping and Spiral Development

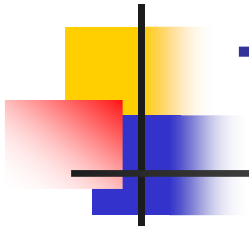


- Spiral development is useful when dealing with new or unfamiliar features or technology.
- If top-down design isn't working for you, try some spiral development!



The Art of Design

- Spiral development is not an alternative to top-down design as much as a complement to it – when designing the prototype you'll still be using top-down techniques.
- Good design is as much creative process as science, and as such, there are no hard and fast rules.



The Art of Design

- The best advice?
- *Practice, practice, practice*