3a - ggplot2 & Program Design

Noud van Giersbergen

University of Amsterdam

2021-04-13

Topics

- Miscellaneous
  - Programming Functions
  - Formatted Output Using `sprintf()`
- The `ggplot2` Package
- Simulation and Program Design

Programming Functions

- Functions are only executed when called
- PC labs: handy if you use separate files
- Include calling a function in the file and hit the "source" button!

```r
pyramid <- function(height){
  if(height>0 & height<9){
    for(i in 1:height){
      cat(rep(" ",height-i),rep("#",i),"\n",sep="")
    }
  }
}
pyramid(3)
```

```
  #
 ##
###
```

Formatted Output using `sprintf()`

- `cat(sprintf(string,...))`: displays formatted output
- `sprintf(string,...)`: returns a string containing a formatted combination of text and variable values
  - `%d`: integer
  - `%md`: the field width (`m`)
  - `%f`: floating point
  - `%m.nf`: the field width (`m`) and the precision (`n`)
  - `%s`: string

```
cat(sprintf("  (i): pi=%f\n",pi))
cat(sprintf(" (ii): pi=%.2f\n",pi))
cat(sprintf("(iii): pi=%11.7f\n",pi))
```

```
  (i): pi=3.141593
 (ii): pi=3.14
(iii): pi=  3.1415927
```

Formatted Output - Examples

```
1/6
```

```
[1] 0.1666667
```

```
sprintf('%.4f', 1/6)
```

```
[1] "0.1667"
```

```
cat(sprintf("Hi, %s, %s and %s","Tick","Trick","Track"))
```

```
Hi, Tick, Trick and Track
```

```
cat(sprintf("Hello, %s, pi is 6 decimals equals %.6f\n","Donald",pi))
```

```
Hello, Donald, pi is 6 decimals equals 3.141593
```

```
sprintf("I woke up at %s:%s%s a.m.", 8, 0, 5)
```

```
[1] "I woke up at 8:05 a.m."
```

Formatted Output - Table Example

- Leibniz formula for π : $1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots = \frac{\pi}{4}$
- Hence, $\pi \approx 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots\right)$

```
sum=1
for(i in 1:10){
  sum=sum+(-1)^i/(2*i+1)
  cat(sprintf("Leibniz approximation after %2d terms: %10.6f\n",i,4*sum))
}
```

```
Leibniz approximation after  1 terms:   2.666667
Leibniz approximation after  2 terms:   3.466667
Leibniz approximation after  3 terms:   2.895238
Leibniz approximation after  4 terms:   3.339683
Leibniz approximation after  5 terms:   2.976046
Leibniz approximation after  6 terms:   3.283738
Leibniz approximation after  7 terms:   3.017072
Leibniz approximation after  8 terms:   3.252366
Leibniz approximation after  9 terms:   3.041840
Leibniz approximation after 10 terms:   3.232316
```

The `ggplot2` package

- Produces layered statistical graphics
- Uses an underlying “grammar” to build graphs layer-by-layer rather than providing premade graphs
- Is easy enough to use without any exposure to the underlying grammar, but is even easier to use once you know the grammar
- Allows the user to build a graph from concepts rather than recall of commands and options

Elements of grammar of graphics

1. **Data**: variables mapped to aesthetic features of the graph
2. **Geoms**: objects/shapes on the graph
3. **Stats**: statistical transformations that summarize data,(e.g mean, confidence intervals)
4. **Scales**: mappings of aesthetic values to data values. Legends and axes visualize scales
5. **Coordinate systems**: the plane on which data are mapped on the graphic
6. **Faceting**: splitting the data into subsets to create multiple variations of the same graph (paneling)

The `ggplot()` function and aesthetics

Example syntax for ggplot() specification (italicized words are to be filled in by you):

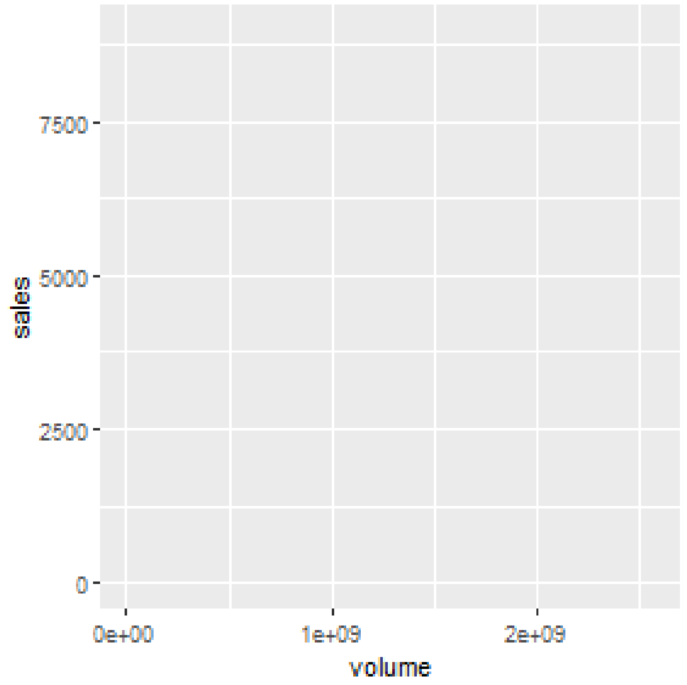
> ggplot(*data*, aes(x=*xvar*, y=*yvar*))

- *data*: name of the data.frame that holds the variables to be plotted
- x and y: aesthetics that position objects on the graph
- *xvar* and *yvar*: names of variables in data mapped to x and y
- Notice that the aesthetics are specified inside aes(), which is itself nested inside of `ggplot()`
- The aesthetics specified inside of `ggplot()` are inherited by subsequent layers

Layers and overriding aesthetics

- Specifying just `x` and `y` aesethetics alone will produce a plot with just the 2 axes

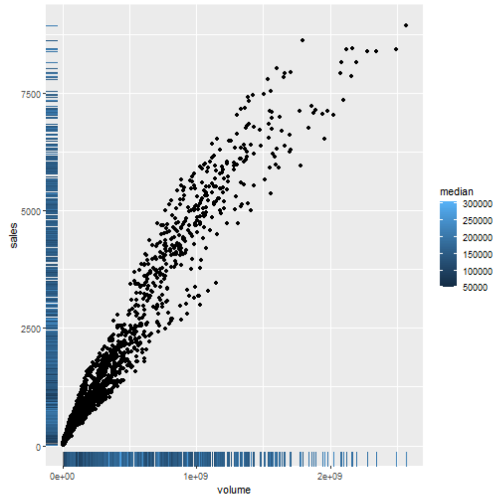
```
library(ggplot2)
library(MASS)
ggplot(data = txhousing, aes(x=volume, y=sales))
```



Layers and overriding aesthetics

- We add layers with the character `+` to the graph to add graphical components
- Layers consist of geoms, stats, scales, and themes
- Remember that each subsequent layer inherits its aesthetics from `ggplot()`
- However, specifying new aesthetics in a layer will override the aesthetics specified in `ggplot()`

```
ggplot(txhousing, aes(x=volume, y=sales)) +
  geom_point() +
  geom_rug(aes(color=median))
```



Aesthetics

- Aesthetics are the visual properties of objects on the graph
- Commonly used aesthetics:
  - **x**: positioning along x-axis
  - **y**: positioning along y-axis
  - **color**: color of objects; for 2-d objects, the color of the object’s outline (compare to fill below)
  - **fill**: fill color of objects
  - **linetype**: how lines should be drawn (solid, dashed, dotted, etc.)
  - **shape**: shape of markers in scatter plots
  - **size**: how large objects appear
  - **alpha**: transparency of objects (value between 0, transparent, and 1, opaque – inverse of how many stacked objects it will take to be opaque)

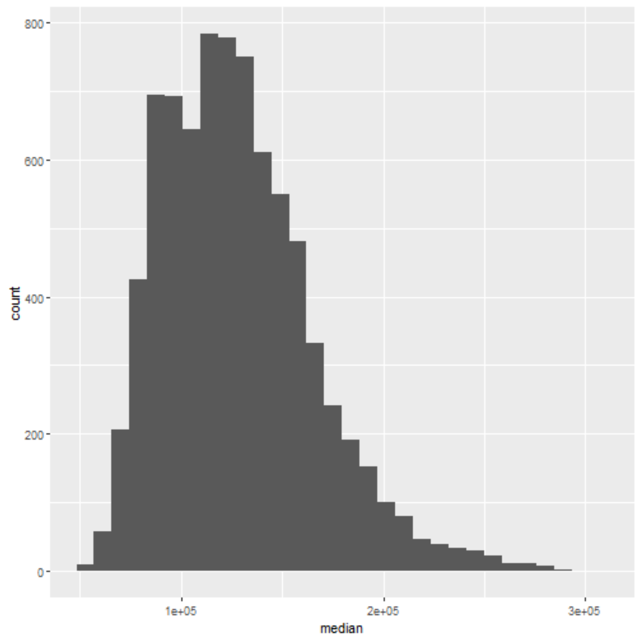
Geoms

- Geom functions differ in the geometric shapes produced for the plot
- Some example geoms:
  - **geom_bar()**: bars with bases on the x-axis
  - **geom_boxplot()**: boxes-and-whiskers
  - **geom_density()**: density plots
  - **geom_histogram()**: histogram
  - **geom_line()**: lines
  - **geom_point()**: points (scatterplot)
  - **geom_text()**: text

Histograms

- Histograms are popular choices to depict the distribution of a continuous variable.
- `geom_histogram()` cuts the continuous variable mapped to x into bins, and count the number of values within each bin

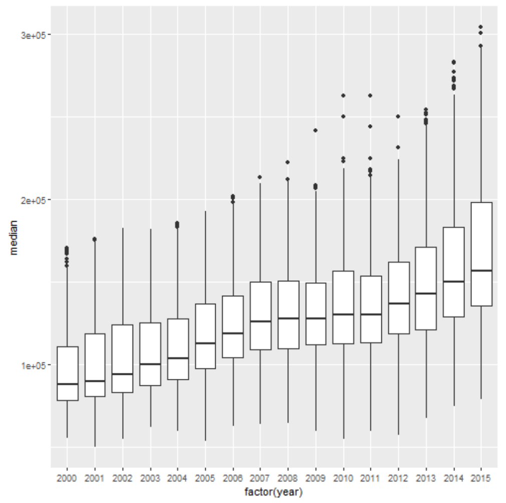
```
ggplot(txhousing, aes(x=median)) +
  geom_histogram()
```



Boxplots

- Boxplots compactly visualize particular statistics of a distributions:
    - lower and upper hinges of box: first and third quartiles
    - middle line: median
    - lower and upper whiskers: (hinge−1.5×IQR) and (hinge+1.5×IQR) where IQR is the interquartile range (distance between hinges)
    - dots: outliers

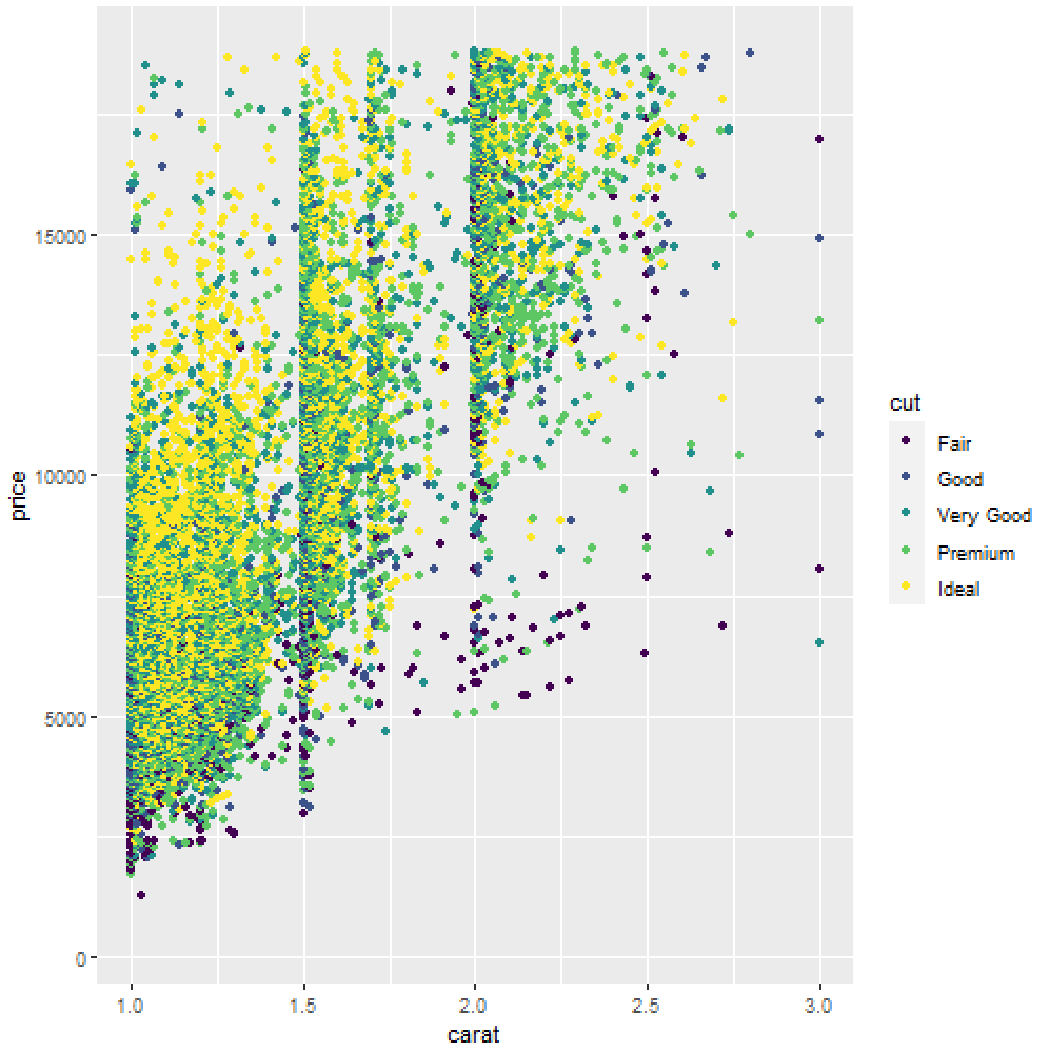
```
ggplot(txhousing, aes(x=factor(year), y=median)) + geom_boxplot()
```



Modifying axis limits and titles

- **lims()**, **xlim()**, **ylim()**: set axis limits
- **xlab()**, **ylab()**, **ggtitle()**, **labs()**: give labels (titles) to x-axis, y-axis, or graph
- **labs** can set labels for all aesthetics and title

```r
ggplot(diamonds, aes(x=carat, y=price, color=cut)) + geom_point() +
  xlim(c(1,3)) # cut ranges from 0 to 5 in the data
```

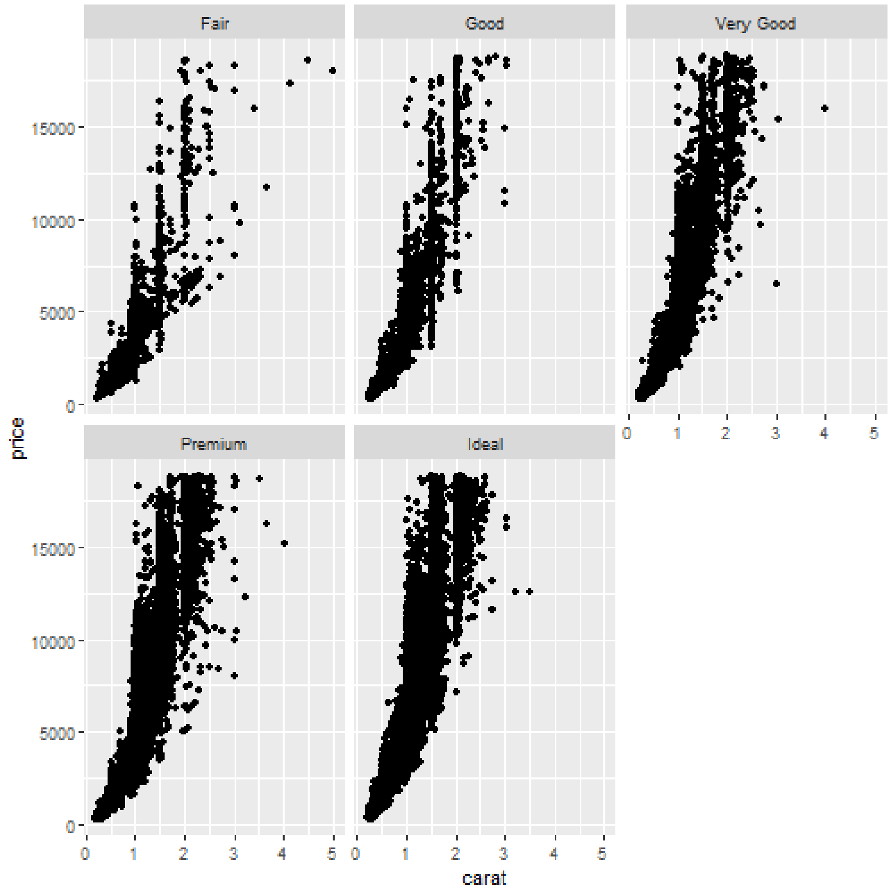


Faceting (paneling)

- Split plots into small multiples (panels) with the faceting functions, `facet_wrap()` and `facet_grid()`
- The resulting graph shows how each plot varies along the faceting variable(s)
  - **facet_wrap()**: wraps a ribbon of plots into a multirow panel of plots. Inside `facet_wrap()`, specify `~`, then a list of splitting variables, separated by `+`. The number of rows and columns can be specified with arguments `nrow` and `ncol`

Faceting (paneling)

```
ggplot(diamonds, aes(x=carat, y=price)) + geom_point() +
facet_wrap(~cut) # create a ribbon of plots using cut
```



Objectives Simulation and Program Design

- 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 techniques in writing complex programs

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.
- 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
- 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
- **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 serve?
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)
- 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
- 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
- R contains a number of functions for working with pseudorandom numbers
- The two functions of greatest interest are `sample` and `runif`

```
set.seed(1)
runif(1)
runif(1)
set.seed(1)
runif(1)
runif(1)
```

```
[1] 0.2655087
[1] 0.3721239
[1] 0.2655087
[1] 0.3721239
```

```
set.seed(2)
runif(1)
runif(1)
set.seed(3)
runif(1)
runif(1)
```

```
[1] 0.1848823
[1] 0.702374
[1] 0.1680415
[1] 0.8075164
```

PseudoRandom Numbers

- The `sample.int` function is used to select a pseudorandom int from a given range:

  > sample.int(n, size = n, replace = FALSE)

- `sample.int(6,1)` returns some number from [1,2,3,4,5,6]
- The `sample` function simulates a random draw from a vector

  > sample(x, size, replace = FALSE)

- `sample(5:100,5, replace = TRUE)` returns a multiple of 5 between 5 and 100, inclusive

```r
sample.int(6,1)
sample(5:100,5, replace = TRUE)
```

```
[1] 4
[1] 40 62 99 12 24
```

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

```r
set.seed(2)
sample.int(6,5, replace = TRUE)
```

```
[1] 5 6 6 1 5
```

PseudoRandom Numbers

```r
hist(sample.int(6,100000, replace = TRUE))
```

PseudoRandom Numbers

- The `runif` function is used to generate pseudorandom floating point values
- Without the parameters `min` and `max`, it returns values uniformly distributed between 0 and 1

> runif(n, min = 0, max = 1)

```r
runif(1)
runif(1)
runif(5)
```

```
[1] 0.3215832
[1] 0.2685355
[1] 0.14816342 0.24918026 0.08018354 0.16109553 0.18446037
```

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
- 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

PseudoRandom Numbers

- If `prob` represents the probability of winning the server, the condition `runif(1) < prob` will succeed with the correct probability

```
if( runif(1) < 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-Down Design

- Typically a program uses the *input*, *process*, *output*
- 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
```

- 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-Down Design

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

```
printIntro()
```

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

```
printIntro()
probA, probB, n <- getInputs()
```

Top-Down 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 the main script
- 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 script must look like this:

```
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
- The complete main program:

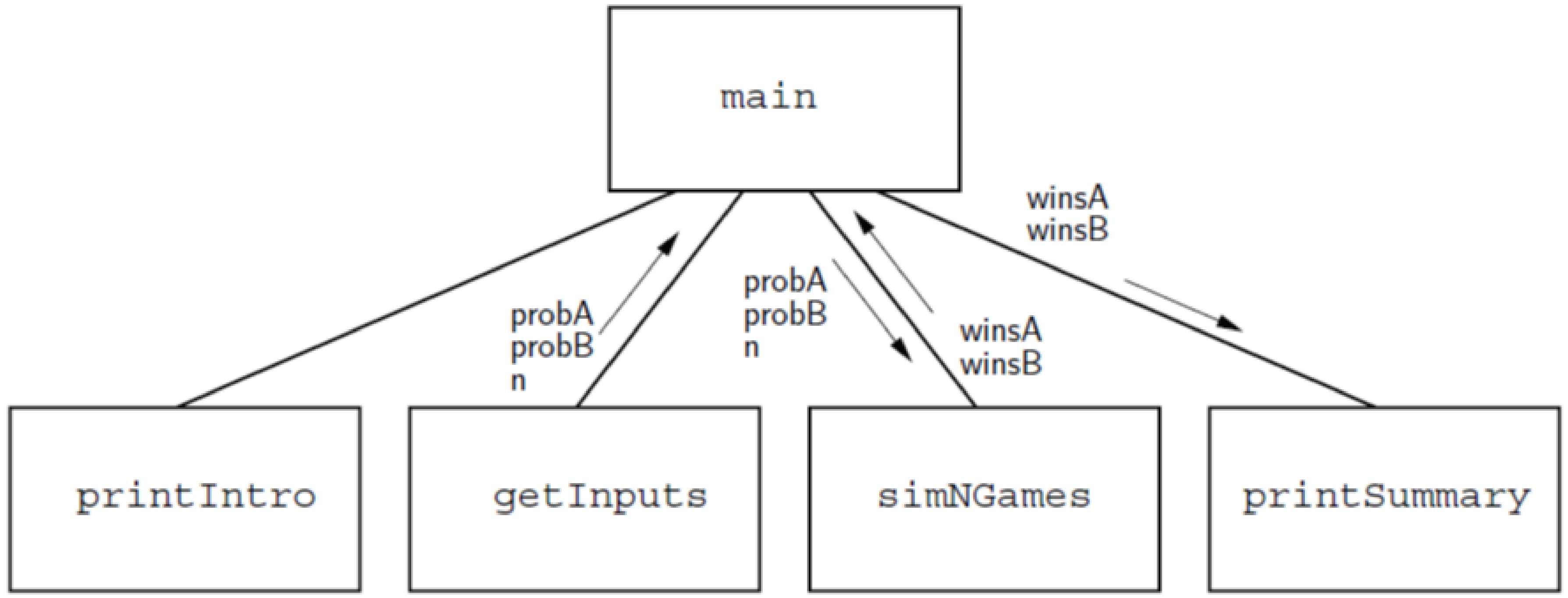
```
printIntro()
probA, probB, n <- getInputs()
winsA, winsB <- simNGames(n, probA, probB)
printSummary(winsA, winsB)
```

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
- Having this information, 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
- 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

- 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

```
printIntro <- function(){
  # Prints an introduction to the program
  cat("This program simulates a game of racquetball between two\n")
  cat('players called "A" and "B".  The abilities of each player is\n')
  cat("indicated by a probability (a number between 0 and 1) that\n")
  cat("the player wins the point when serving. Player A always\n")
  cat("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 script

```
getInputs <- function(){
  # RETURNS a vector with probA, probB, number of games to simulate
  a = as.numeric(readline("What is the prob. player A wins a serve? "))
  b = as.numeric(readline("What is the prob. player B wins a serve? "))
  n = as.numeric(readline("How many games to simulate? "))
  return(c(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**:

```
simNGames <- function(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 1:n){
  }
}
```

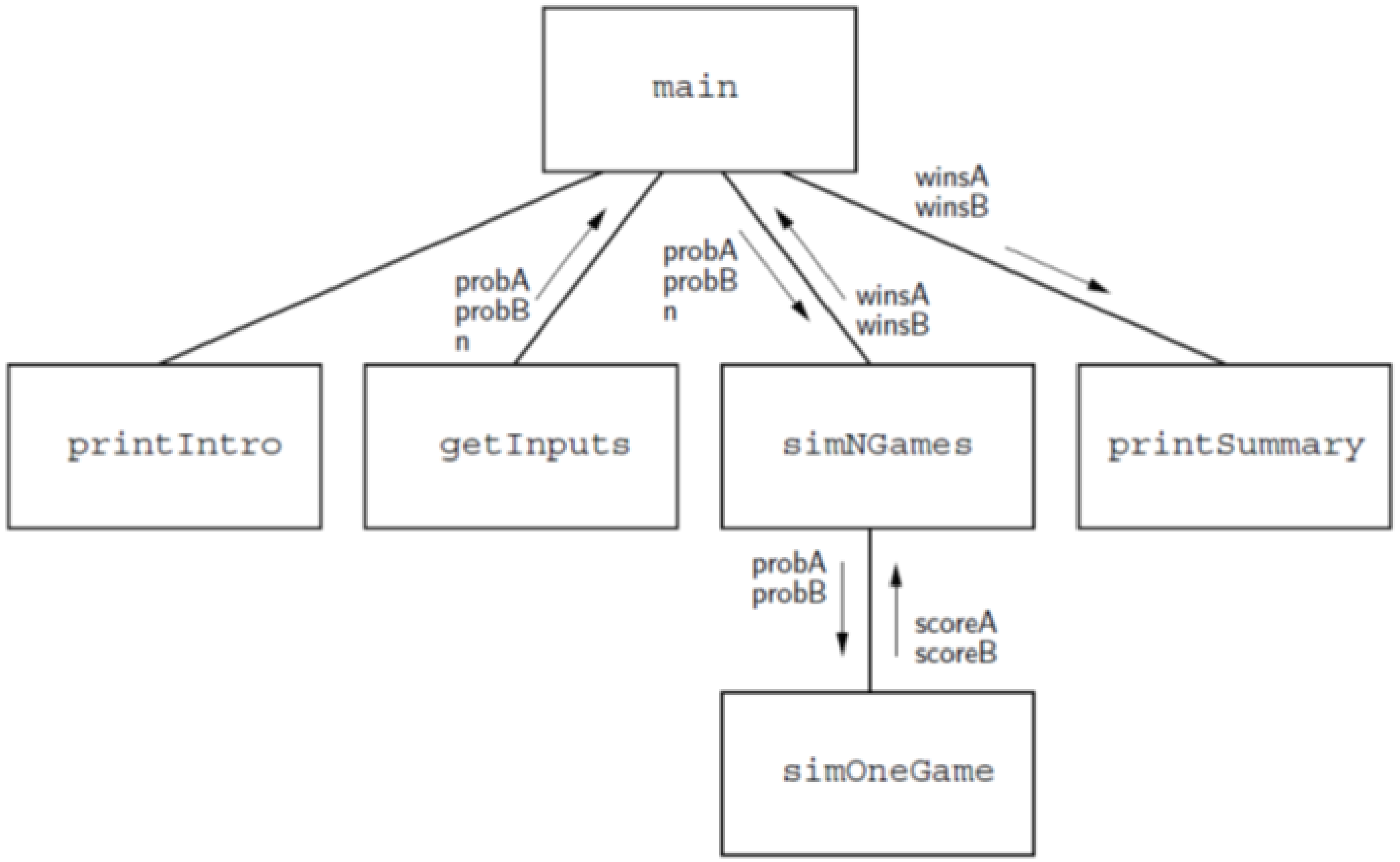
- The next thing we need to do is simulate a game of racquetball. We're not sure how to do that, so let's put it off until later!
- Let's assume there's a function called `simOneGame` that can do it
- The inputs to `simOneGame` are easy – the probabilities for each player. But what is the output?

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

```r
simNGames <- function(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 1:n){
    scores <- simOneGame(probA, probB)
    scoreA <- scores[1]; scoreB <- scores[2]
    if(scoreA>scoreB)
      winsA <- winsA + 1
    else
      winsB <- winsB + 1
  }
  return(c(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 scores 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".

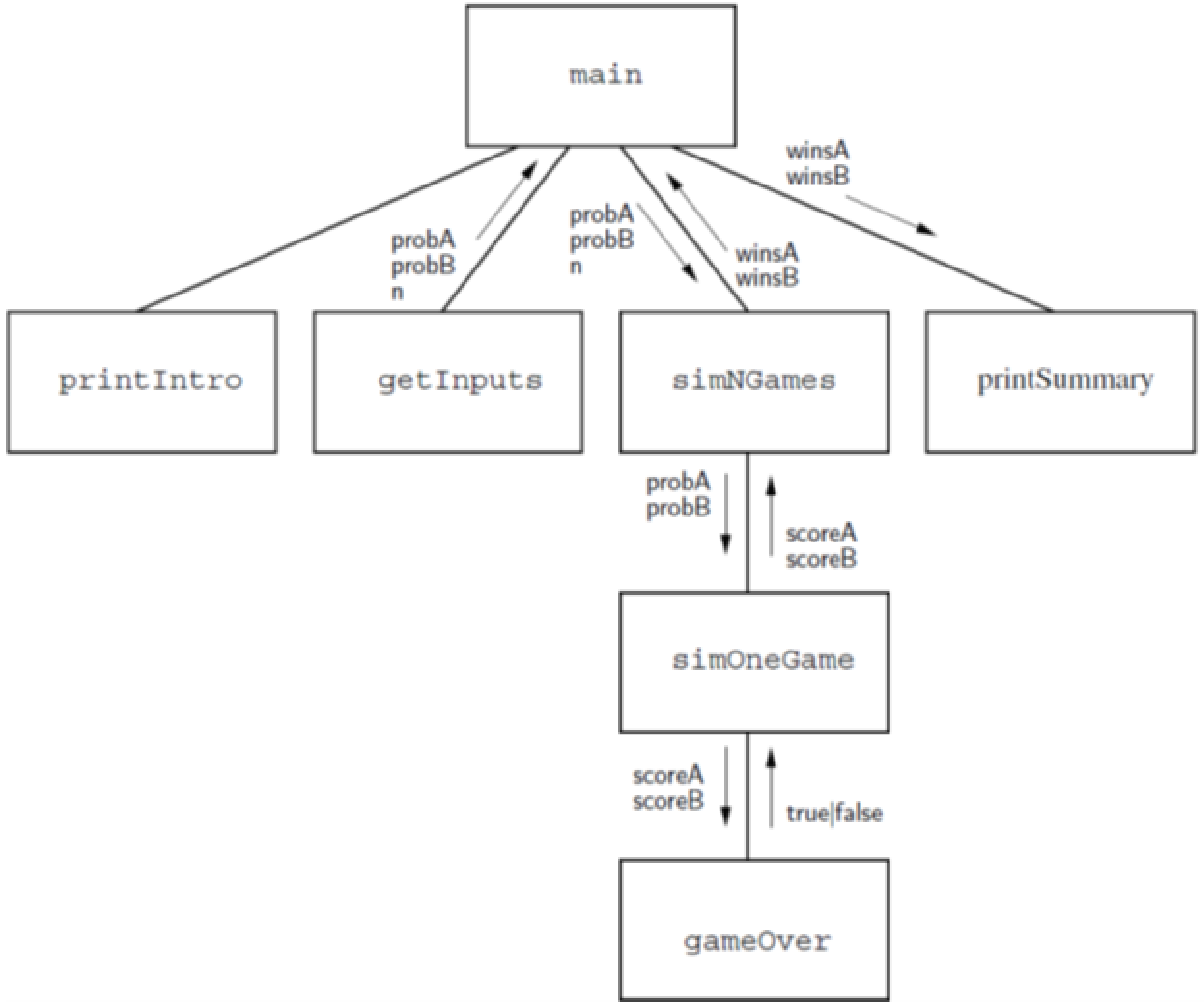
```
Initialize scores to 0
Set serving to "A"
Loop while game is not over{
  Simulate one serve of whichever player is serving
  update the status of the game }
Return scores
```

Third-Level Design

```
simOneGame <- function(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

- 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
- `(runif(1) < prob)`
- The probability we use is determined by whom is serving, contained in the variable `serving`
- 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( runif(1) < 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

```r
simOneGame <- function(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( !gameOver(scoreA, scoreB) ){
    if(serving == "A"){
      if( runif(1) < probA )
        scoreA = scoreA + 1
      else
        serving = "B"
    } else {
      if( runif(1) < probB )
        scoreB = scoreB + 1
      else
        serving = "A"
    }
  }
  return(c(scoreA, scoreB))
}
```

Finishing Up

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

```
gameOver <- function(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:

```r
gameOver  <- function(a,b){
    # a and b are scores for players in a racquetball game
    # RETURNS true if game is over, false otherwise
    return( (a == 15) | (b == 15) )
}
```

- printSummary is equally simple!

```r
printSummary <- function (winsA, winsB){
  # Prints a summary of wins for each player.
  n <- winsA + winsB
  cat(sprintf("\nGames simulated: %d\n", n))
  cat(sprintf("Wins for A: %d (%.2f%%)\n",winsA, 100*winsA/n))
  cat(sprintf("Wins for B: %d (%.2f%%)\n",winsB, 100*winsB/n))
}
```

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 Unit Testing

- 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
- 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 execute various routines/functions to ensure they work properly

Unit Testing

- We could start with the `gameOver` function

```r
gameOver(0,0)
```

```
[1] FALSE
```

```r
gameOver(5,10)
```

```
[1] FALSE
```

```r
gameOver(15,10)
```

```
[1] TRUE
```

```r
gameOver(3,15)
```

```
[1] 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)
```

```
[1] 15  9
```

```
simOneGame(.5, .5)
```

```
[1]  8 15
```

```
simOneGame(.4, .9)
```

```
[1]  0 15
```

```
simOneGame(.9, .4)
```

```
[1] 15  0
```

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

```r
printIntro()
pars <- getInputs()
probA <- pars[1]; probB <- pars[1]; n <- pars[3]
results <- simNGames(n, probA, probB)
winsA <- results[1]; winsB <- results[2];
printSummary(winsA, winsB)
```

```
What is the prob. player A wins a serve? 0.65
What is the prob. player B wins a serve? 0.6
How many games to simulate? 5000

Games simulated: 5000
Wins for A: 3329 (66.58%)
Wins for B: 1671 (33.42%)
```

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

The Art of Design

- Good design is as much creative process as science, and as such, there are no hard and fast rules
- The best advice?

Practice, practice, practice

- https://projecteuler.net/