Statistical Computing (36-350)

Lectures 1 and 2: Introduction to the course; Data and Data Structures

Cosma Shalizi

26 and 28 August 2013

Agenda

- Course Overview and Mechanics
- Basic, built-in data types
- Basic, built-in functions and operators
- First data structures: Vectors and arrays
- Matrices
- Lists
- Data frames
- Structures of structures

Why good statisticians learn how to program

INDEPENDENCE: otherwise, you rely on someone else always having made exactly the right tool for you, and giving it to you
HONESTY: otherwise, you end up distorting the problem to match the tools you happen to have
CLARITY: turning your method into something a machine can do forces you to discipline your thinking and make it communicable; and science is public

How this class will work

First half: general programming, with statistical illustrations
Second half: computational tasks especially relevant to statistics, using general programming ideas
The class will be *very* cumulative
$\therefore$ *Keep up with the readings and exercises*

Mechanics

Two lectures a week on concepts and methods
Lab to try out stuff and get immediate feedback
HW to do longer and more complicated things, building on labs
Exam to check understanding
Project to show actual competence
Keep up with the readings and exercises

Assignments, office hours, class notes, grading policies, useful links on R: `http://www.stat.cmu.edu/~cshalizi/statcomp`
Blackboard for a grade-book and for turning in homework *only*, check the class website for everything else

Books

Required

Matloff *Art of R Programming*: textbook; expect to read some of it most weeks

Teetor *The R Cookbook*: reference; like the help files, but by subject/task, not command; check it when stumped about how to do something

Spector *Data Manipulation*: textbook; mostly for the second half of the class

Optional but Recommended

Chambers *Software for Data Analysis*: much more about how R works, and good programming techniques

Chang *R Graphics Cookbook*: like *The R Cookbook*

Brandam and Korf *Unix and Perl:* basic tools for serious computing

Spufford *Red Plenty*: Remaking the world through the power of computational data analysis

The class in a nutshell: functional programming

Several models of how to write code; we will use **functional programming**

2 sorts of things (**objects**): **data** and **functions**

Data things like 7, "seven", 7.000, the matrix $\begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$

Functions things like log, + (two arguments), or < (two arguments), mod (two arguments), `mean` (one argument)

Definition

A function is a machine which turns input objects (**arguments**) into an output object (**return value**), possibly with **side effects**, according to a definite rule

Programming is writing functions to transform inputs into outputs
Good programming ensures the transformation is done easily and correctly
Machines are made out of machines; functions are made out of functions, like $f(a,b) = a^2 + b^2$

> *The route to good programming is to take the big transformation and break it down into smaller ones, and then break those down, until you come to tasks which the built-in functions can do*

Before functions, data

Different kinds of data object
All data is represented in binary format, by **bits** (TRUE/FALSE, YES/NO, 1/0)

Booleans Direct binary values: `TRUE` or `FALSE` in R

Integers whole numbers (positive, negative or zero), represented by a fixed-length block of bits

Characters fixed-length blocks of bits, with special coding
strings = sequences of characters

Floating point numbers a fraction (with a finite number of bits) times an exponent, like 1.87×10^6, but in binary form

Missing or ill-defined values `NA`, `NaN`, etc.

Operators

Unary - for arithmetic negation, ! for Boolean

Binary usual arithmetic ones, plus ones for modulo and integer division; take two numbers and give a number:

```
> 7+5
[1] 12
> 7-5
[1] 2
> 7*5
[1] 35
> 7/5
[1] 1.4
> 7^5
[1] 16807
> 7 %% 5
[1] 2
> 7 %/% 5
[1] 1
```

Comparisons are also binary operators; they take two objects, like numbers, and give a Boolean:

```
> 7 > 5
[1] TRUE
> 7 < 5
[1] FALSE
> 7 >= 7
[1] TRUE
> 7 <= 5
[1] FALSE
> 7 == 5
[1] FALSE
> 7 != 5
[1] TRUE
```

You can also compare strings, but that depends on R details in non-obvious ways (is `And` before or after `an`?)

Boolean operators, for “and” and “or”:

```
> (5 > 7) & (6*7 == 42)
[1] FALSE
> (5 > 7) | (6*7 == 42)
[1] TRUE
```

(Will see special doubled forms, `||` and `&&`, later on)

Peculiarities of floating-point numbers

The more bits in the fraction part, the more precision
The R floating-point data type is a `double`, a.k.a. `numeric`

back when memory was expensive, the now-standard number of bits was twice the default
Finite precision $\Rightarrow$ arithmetic on `doubles` $\neq$ arithmetic on $\mathbb{R}$.

```
> 0.45 == 3*0.15
[1] FALSE
```

Often ignorable, but not always:

- rounding errors tend to accumulate in long calculations
- when results should be $\approx$ `0`, errors can flip signs:

  ```
  > 0.45-3*0.15
  [1] 5.551115e-17
  ```
- usually better to use `all.equal()` than exact comparisons

  ```
  (0.5-0.3) == (0.3-0.1)
  all.equal(0.5-0.3,0.3-0.1)
  ```

Peculiarities of Integers

Creating a whole number in the terminal doesn't make an integer; it makes a double, which just so happens to have no fractional part

```
> is.integer(7)
[1] FALSE
```

This looks just the same as an integer

```
> as.integer(7)
[1] 7
```

To test for being a whole number, use `round()`:

```
> round(7) == 7
[1] TRUE
```

More types

`typeof()` function returns the type
`is.`*foo*`()` functions return Booleans for whether the argument is of type *foo*:

```
> typeof(7)
[1] "double"
> is.numeric(7)
[1] TRUE
> is.na(7)
[1] FALSE
> is.character(7)
[1] FALSE
> is.character("7")
[1] TRUE
> is.character("seven")
[1] TRUE
> is.na("seven")
[1] FALSE
```

`as.`*foo*`()` tries to "cast" argument into something of type *foo* When you try to combine things of different types, R will try to convert to a type which makes sense, silently, and protest if not

```
> as.character(5/6)
[1] "0.833333333333333"
> as.numeric(as.character(5/6))
[1] 0.8333333
> 6*as.character(5/6)
Error in 6 * as.character(5/6) : non-numeric argument to binary operator
> 6*as.numeric(as.character(5/6))
[1] 5
> 5/6 == as.numeric(as.character(5/6))
[1] FALSE
```

(why is that last false?)
Remember you can compare strings:

```
> as.character(5/6) > 0
[1] TRUE
> as.character(5/6) > 0.5
[1] TRUE
> as.character(5/6) > 1
[1] FALSE
> as.character(5/6) > "z"
[1] FALSE
```

Data can have names

We can give names to data objects; these give us **variables**
A few are built in

```
> pi
[1] 3.141593
```

The **assignment operator** is <- or =

```
> approx.pi <- 22/7
> approx.pi
[1] 3.142857
> diameter.in.cubits = 10
> approx.pi*diameter.in.cubits
[1] 31.42857
```

Using names and variables makes code: easier to design, easier to debug, less prone to bugs, easier to improve, and easier for others to read
Avoid "magic constants"; use named variables

Example: resource allocation ("mathematical programming")

Factory makes cars and trucks, using labor and steel

- a car takes 40 hours of labor and 1 ton of steel
- a truck takes 60 hours and 3 tons of steel
- resources: 1600 hours of labor and 70 tons of steel each week

Can it make 20 trucks and 8 cars?

```
> 60*20 + 40*8 <= 1600
[1] TRUE
> 3*20 + 1*8 <= 70
[1] TRUE
```

How about 20 trucks and 9 cars?

```
> 60*20 + 40*9 <= 1600
[1] TRUE
> 3*20 + 1*9 <= 70
[1] TRUE
```

How about 20 trucks and 10 cars?

Could just write it out *again*, but this is

- boring and repetitive
- error-prone (what if I forget to change the number of cars in line 2, or type 50 when I mean 60?)
- obscure if we come back to our work later (what *are* these numbers?)

```
> hours.car <- 40; hours.truck <- 60
> steel.car <- 1; steel.truck <- 3
> available.hours <- 1600; available.steel <- 70
> output.trucks <- 20; output.cars <- 10
> hours.car*output.cars + hours.truck*output.trucks <= available.hours
[1] TRUE
> steel.car*output.cars + steel.truck*output.trucks <= available.steel
[1] TRUE
```

Now if something changes we just need to change the appropriate variables, and re-run the last two lines
A step towards **abstraction**

First data structure: vectors

Group related data values into one object, a **data structure**
A **vector** is a sequence of values, all of the same type

```
> x <- c(7, 8, 10, 45)
> x
[1]  7  8 10 45
> is.vector(x)
[1] TRUE
```

`c()` function returns a vector containing all its arguments in order
`x[1]` is the first element, `x[4]` is the 4th element, `x[-4]` is a vector containing all but the fourth element
`vector(length=6)` returns an empty vector of length 6; helpful for filling things up later

```
weekly.hours <- vector(length=5)
weekly.hours[5] <- 8
```

Vector arithmetic

Operators apply to vectors "pairwise":

```
> y <- c(-7, -8, -10, -45)
> x+y
[1] 0 0 0 0
```

Recycling: repeat elements in shorter vector when combined with longer

```
> x + c(-7,-8)
[1]  0  0  3 37
```

Single numbers are vectors of length 1 for purposes of recycling:

```
> x + 1
[1]  8  9 11 46
```

Can also do pairwise comparisons:

```
> x > 9
[1] FALSE FALSE  TRUE  TRUE
```

Note: returns Boolean vector
Boolean operators work pairwise; but written double, combines individual values into a single Boolean:

```
> (x > 9) & (x < 20)
[1] FALSE FALSE  TRUE FALSE
> (x > 9) && (x < 20)
[1] FALSE
```

To compare whole vectors, best to use `identical()` or `all.equal`:

```
> x == -y
[1] TRUE TRUE TRUE TRUE
> identical(x,-y)
[1] TRUE
> identical(c(0.5-0.3,0.3-0.1),c(0.3-0.1,0.5-0.3))
[1] FALSE
> all.equal(c(0.5-0.3,0.3-0.1),c(0.3-0.1,0.5-0.3))
[1] TRUE
```

Functions on vectors

Lots of functions take vectors as arguments:

- `mean()`, `median()`, `sd()`, `var()`, `max()`, `min()`, `length()`, `sum()` all return single numbers
- `sort()` returns a new vector
- `hist()` takes a vector of numbers and produces a histogram, a highly structured object, with the side-effect of making a plot
- similarly `ecdf()` produces a cumulative-density-function object
- `summary()` gives a five-number summary of numerical vectors
- `any()` and `all()` are useful on Boolean vectors

Addressing vectors

Vector of indices:

```
> x[c(2,4)]
[1]  8 45
```

Vector of negative indices

```
> x[c(-1,-3)]
[1]  8 45
```

(why not `8 10`?)
Boolean vector:

```
> x[x>9]
[1] 10 45
> y[x>9]
[1] -10 -45
```

`which()` takes a Boolean vector and gives a vector of indices for the TRUE values; useful with tests:

```
> places <- which(x > 9)
> y[places]
[1] -10 -45
```

Named components

You can give names to elements or components of vectors

```
> names(x) <- c("v1","v2","v3","fred")
> names(x)
[1] "v1"   "v2"   "v3"   "fred"
> x[c("fred","v1")]
fred   v1
  45    7
```

note the labels; not actually part of the value
`names(x)` is just another vector (of characters):

```
> names(y) <- names(x)
> sort(names(x))
[1] "fred" "v1"   "v2"   "v3"
> which(names(x)=="fred")
[1] 4
```

Back to resource allocation

Use vectors to group thing together

```
> hours <- c(hours.car,hours.truck)
> steel <- c(steel.car,steel.truck)
> output <- c(output.cars,output.trucks)
> available <- c(available.hours,available.steel)
```

could make it

```
> all(hours[1]*output[1]+hours[2]*output[2] <= available[1],
+     steel[1]*output[1]+steel[2]*output[2] <= available[2])
[1] TRUE
```

or even

```
> all(c(sum(hours*output), sum(steel*output)) <= available)
[1] TRUE
```

. . . but then we'd have to remember the ordering of components in each vector, and *always* use that order
Use names instead:

```
> names(hours) <- c("cars", "trucks")
> names(steel) <- names(hours); names(output) <- names(hours)
> names(available) <- c("hours","steel")
> all(hours["cars"]*output["cars"] + hours["trucks"]*output["trucks"] <=
+     available["hours"],
+     steel["cars"]*output["cars"] + steel["trucks"]*output["trucks"] <=
+     available["steel"])
[1] TRUE
```

Better, but not as concise. Now try:

```
> needed <- c(sum(hours*output[names(hours)]),
+             sum(steel*output[names(steel)]))
> names(needed) <- c("hours","steel")
> all(needed <= available[names(needed)])
[1] TRUE
```

Not perfect programming, but better
What would we have to change to start allowing for motorcycles?

Vector structures, starting with arrays

Most more complicated structures in R are made by adding bells and whistles to vectors, so "vector structures"
Most useful: arrays

```
> x.arr <- array(x,dim=c(2,2))
> x.arr
     [,1] [,2]
[1,]    7   10
[2,]    8   45
```

Note: filled the first column, then the 2nd; `dim` tells it how many rows and columns
Can have $3, 4, \ldots n$ dimensional arrays; `dim` is then a vector of length n

Some properties of the array:

```
> dim(x.arr)
[1] 2 2
> is.vector(x.arr)
[1] FALSE
> is.array(x.arr)
[1] TRUE
> typeof(x.arr)
[1] "double"
> str(x.arr)
 num [1:2, 1:2] 7 8 10 45
> attributes(x.arr)
$dim
[1] 2 2
```

Note: `typeof()` returns the type of the *elements*
Note: `str()` gives the **structure**: here, a numeric array, with two dimensions, both indexed 1–2, and then the actual numbers
Exercise: try all these with `x`

Accessing and operating on arrays

Can access a 2-D array either by pairs of indices or by the underlying vector:

```
> x.arr[1,2]
[1] 10
> x.arr[3]
[1] 10
```

Omitting an index means "all of it":

```
> x.arr[c(1:2),2]
[1] 10 45
> x.arr[,2]
[1] 10 45
```

Using a vector-style function on a vector structure will go down to the underlying vector, *unless* the function is set up to handle arrays specially:

```
> which(x.arr > 9)
[1] 3 4
```

Many functions *do* preserve array structure:

```
> y.arr <- array(y,dim=c(2,2))
> y.arr + x.arr
     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

Others specifically act on each row or column of the array separately:

```
> rowSums(x.arr)
[1] 17 53
```

We will see a lot more of this idea

Matrices

In R, a matrix is a specialization of a 2D array

```
> factory <- matrix(c(40,1,60,3),nrow=2)
> factory
     [,1] [,2]
[1,]   40   60
[2,]    1    3
> is.array(factory)
[1] TRUE
> is.matrix(factory)
[1] TRUE
```

could also specify `ncol`, and/or `byrow=TRUE` to fill by rows.
Element-wise operations with the usual arithmetic and comparison operators (e.g., `factory/3`)
Compare whole matrices with `identical()` or `all.equal()`

Matrix multiplication has a special operator:

```
> six.sevens <- matrix(rep(7,6),ncol=3)
> six.sevens
     [,1] [,2] [,3]
[1,]    7    7    7
[2,]    7    7    7
> factory %*% six.sevens # [2x2] * [2x3]
     [,1] [,2] [,3]
[1,]  700  700  700
[2,]   28   28   28
> six.sevens %*% factory # [2x3] * [2x2]
Error in six.sevens %*% factory : non-conformable arguments
```

Multiplying by a vector:

```
> output <- c(10,20)
> factory %*% output
     [,1]
[1,] 1600
[2,]   70
> output %*% factory
     [,1] [,2]
[1,]  420  660
```

R silently casts the vector as a row or column matrix

Matrix transpose:

```
> t(factory)
     [,1] [,2]
[1,]   40    1
[2,]   60    3
```

Matrix determinant:

```
> det(factory)
[1] 60
```

Extracting or replacing the diagonal:

```
> diag(factory) # What's the diagonal of the matrix?
[1] 40  3
> diag(factory) <- c(35,4) # Change it
> factory # See that it changed
     [,1] [,2]
[1,]   35   60
[2,]    1    4
> diag(factory) <- c(40,3) # Set it back for later
```

Creating a diagonal matrix or an identity matrix:

```
> diag(c(3,4))
     [,1] [,2]
[1,]    3    0
[2,]    0    4
> diag(2)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Inverting a matrix:

```
> solve(factory)
             [,1]       [,2]
[1,]  0.05000000 -1.0000000
[2,] -0.01666667  0.6666667
> factory %*% solve(factory) # Check that this does what I claim
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Why is it called `solve()`?

Solving the linear system $\mathbf{A}\vec{x} = \vec{b}$, for unknown vector $\vec{x}$:

```
> available <- c(1600,70)
> solve(factory,available)
[1] 10 20
> factory %*% solve(factory,available)
     [,1]
[1,] 1600
[2,]   70
```

Names in Matrices

We can name either rows or columns or both, with `rownames()` and `colnames()`
These are just character vectors, and we use the same function to get and to set their values
Names help us understand what we're working with
Names can be used to coordinate different objects

Example: someone gets the order of cars and trucks wrong

```
> rownames(factory) <- c("labor","steel")
> colnames(factory) <- c("cars","trucks")
> factory
       cars trucks
labor    40     60
steel     1      3
> output <- c(20,10)
> names(output) <- c("trucks","cars")
> available <- c(1600,70)
> names(available) <- c("labor","steel")
> factory %*% output # But we've got cars and trucks mixed up!
       [,1]
labor 1400
steel   50
> factory %*% output[colnames(factory)]
       [,1]
labor 1600
steel   70
> all(factory %*% output[colnames(factory)] <= available[rownames(factory)])
[1] TRUE
```

Notice: Last lines don't have to change if we add motorcycles as output or rubber and glass as inputs (abstraction again)

Doing the same thing to each row or column

Take the mean: `rowMeans()`, `colMeans()`: input is matrix, output is vector. Also `rowSums()`, etc.
`summary()`: Applies vector-style summary to each column
`apply()`, takes 3 arguments: the array or matrix, then 1 for rows and 2 for columns, then name of the function to apply to each

```
> rowMeans(a.matrix)
[1] 22.5  6.0
> apply(a.matrix,1,mean)
[1] 22.5  6.0
```

What would `apply(a.matrix,1,sd)` do?

Lists

Sequence of values, *not* necessarily all of the same type

```
> my.distribution <- list("exponential",7,FALSE)
> my.distribution
[[1]]
[1] "exponential"

[[2]]
[1] 7

[[3]]
[1] FALSE
```

Most of things which you can do with vectors you can also do with lists

Accessing bits of lists

Access: can use [] as with vectors
or use [[]], but only with a single index (and it drops names and structures)

```
> is.character(my.distribution)
[1] FALSE
> is.character(my.distribution[[1]])
[1] TRUE
> my.distribution[2]^ 2
Error in my.distribution[2]^2 : non-numeric argument to binary operator
> my.distribution[[2]]^2
[1] 49
```

(What happens when you try [[]] on a vector?)

Add to lists with `c()` (also works with vectors):

```
> my.distribution <- c(my.distribution,7)
> my.distribution
[[1]]
[1] "exponential"
[[2]]
[1] 7
[[3]]
[1] FALSE
[[4]]
[1] 7
```

Chop off the end of a list by setting length to something smaller (also works with vectors):

```
> length(my.distribution)
[1] 4
> length(my.distribution) <- 3
> my.distribution
[[1]]
[1] "exponential"
[[2]]
[1] 7
[[3]]
[1] FALSE
```

Names in lists

We can name some or all of the elements of a list

```
> names(my.distribution) <- c("family","mean","is.symmetric")
> my.distribution
$family
[1] "exponential"
$mean
[1] 7
$is.symmetric
[1] FALSE
```

Then we access by name, using $ (which removes names and structure):

```
> my.distribution$family
[1] "exponential"
> my.distribution[["family"]]
[1] "exponential"
> my.distribution["family"]
$family
[1] "exponential"
```

Using names when we make the list:

```
> another.distribution <- list(family="gaussian",mean=7,sd=1,is.symmetric=TRUE)
> another.distribution
$family
[1] "gaussian"
$mean
[1] 7
$sd
[1] 1
$is.symmetric
[1] TRUE
```

or after:

```
> my.distribution$was.estimated <- FALSE
> my.distribution[["last.updated"]] <- "2011-08-30"
> my.distribution
$family
[1] "exponential"
$mean
[1] 7
$is.symmetric
[1] FALSE
$was.estimated
[1] FALSE
$last.updated
[1] "2011-08-30"
```

Remove an entry in the list by assigning it the value `NULL`; try `my.distribution$was.estimated<-NULL`

Key-Value Pairs

Lists give us a way to store and look data up by name rather than number (**key-value pairs**, **dictionary**, **associative array**, **hash**)
If all our distributions have a component named `family`, we can look it up by name without caring where it is in the list
(More abstraction)

Data frames

Data frame = classic data table, with n rows for cases, and p columns for variables

Lots of the really-statistical parts of R presume data frames

Not just a matrix because every column can be of a different type

A hybrid of a matrix and a list; can access columns either like a matrix or like named parts of a list

Many functions for matrices also work on data frames (`rowSums()`, `summary()`, `apply()`, ...)

Cannot do matrix multiplication on a data frame even if it's all numbers

Example:

```
> a.matrix <- matrix(c(35,8,10,4),nrow=2)
> colnames(a.matrix) <- c("v1","v2")
> a.matrix
     v1 v2
[1,] 35 10
[2,]  8  4
> a.matrix$v1 # The $ access operator doesn't work on a matrix
Error in a.matrix$v1 : $ operator is invalid for atomic vectors
> a.data.frame <- data.frame(a.matrix,logicals=c(TRUE,FALSE))
> a.data.frame
  v1 v2 logicals
1 35 10     TRUE
2  8  4    FALSE
> a.data.frame$v1 # But $ does work on a data frame
[1] 35  8
> a.data.frame[,"v1"]
[1] 35  8
> a.data.frame[1,]
  v1 v2 logicals
1 35 10     TRUE
> colMeans(a.data.frame)
      v1       v2 logicals
    21.5      7.0      0.5
```

We can add rows or columns to an array or data-frame with `rbind()` and `cbind()`, but be careful about forced type conversions

```
> rbind(a.data.frame,list(v1=-3,v2=-5,logicals=TRUE))
  v1 v2 logicals
1 35 10     TRUE
2  8  4    FALSE
3 -3 -5     TRUE
> rbind(a.data.frame,c(3,4,6))
  v1 v2 logicals
1 35 10        1
2  8  4        0
3  3  4        6
```

Structures of structures

Lists of lists, lists of vectors, lists of lists of lists of vectors...
This **recursion** lets us build arbitrarily complicated data structures from the basic ones
Lots of complicated objects are lists of data structures

Example: Eigenstuff

`eigen()` finds eigenvalues and eigenvectors of a matrix
return value is a list of a vector (of eigenvalues) and a matrix (of eigenvectors)

```
> eigen(factory)
$values
[1] 41.556171  1.443829

$vectors
           [,1]       [,2]
[1,] 0.99966383 -0.8412758
[2,] 0.02592747  0.5406062

> class(eigen(factory))
[1] "list"
> str(eigen(factory))
List of 2
 $ values : num [1:2] 41.56 1.44
 $ vectors: num [1:2, 1:2] 0.9997 0.0259 -0.8413 0.5406
```

With complicated objects, you can access parts of parts (of parts...):

```
> factory %*% eigen(factory)$vectors[,2]
           [,1]
[1,] -1.2146583
[2,]  0.7805429
> eigen(factory)$values[2] * eigen(factory)$vectors[,2]
[1] -1.2146583  0.7805429
> eigen(factory)$values[2]
[1] 1.443829
> eigen(factory)[[1]][[2]] # NOT [[1, 2]]
[1] 1.443829
```

Summary

- We write programs by composing functions to manipulate data
- The basic data types let us represent Booleans, numbers, and characters
- Data structures let us group related data together: vectors and arrays let us group values of the same type
- Matrices act like you'd hope they would
- Lists let us combine different types of data
- Data frames are hybrids of matrices and lists, for classic tabular data
- Use variables rather than constants
- Name components of structures make data more meaningful and control access
- Recursion lets us build complicated structures