Statistical Computing (36-350)

Lecture 1: Introduction to the course; Data

Cosma Shalizi and Vincent Vu

29 August 2011

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

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

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

How this class will work

First half: general programming, with statistical illustrations
Second half: computational tasks especially relevant to statistics, using general programming ideas

How this class will work

First half: general programming, with statistical illustrations
Second half: computational tasks especially relevant to statistics, using general programming ideas
Lecturing split between Prof. Vu and myself as convenient

How this class will work

First half: general programming, with statistical illustrations

Second half: computational tasks especially relevant to statistics, using general programming ideas

Lecturing split between Prof. Vu and myself as convenient

The class will be *very* cumulative

How this class will work

First half: general programming, with statistical illustrations
Second half: computational tasks especially relevant to statistics, using general programming ideas
Lecturing split between Prof. Vu and myself as convenient
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
Exams to check understanding

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
Exams to check understanding
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 will be used only for a grade-book and for turning in homework, check the class website for everything else

Books

Braun and Murdoch: textbook; required; expect to have to read some of it every week, at least for the first half

Books

Braun and Murdoch: textbook; required; expect to have to read some of it every week, at least for the first half

Teetor: reference; required; like the help files, but organized by subject/task and not by command; consult it when stumped about how to do a particular thing

Books

Braun and Murdoch: textbook; required; expect to have to read some of it every week, at least for the first half
Teetor: reference; required; like the help files, but organized by subject/task and not by command; consult it when stumped about how to do a particular thing
Chambers: optional; much more about details of R, good programming practices, and advanced programming techniques

First assignments

Homework # 1 goes out Wednesday; due at the start of class, Wednesday 7 September

Read chapters 1 and 2 of Braun and Murdoch by Friday

First assignments

Homework # 1 goes out Wednesday; due at the start of class, Wednesday 7 September

Read chapters 1 and 2 of Braun and Murdoch by Friday

Keep up with the readings and exercises

The class in a nutshell: functional programming

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

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**

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 $\begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$

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), $\bmod$ (two arguments), `mean` (one argument)

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)

Function: 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

Programming is writing functions to transform inputs into outputs
Good programming ensures the transformation is done easily and correctly

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 $f(a, b) = a^2 + b^2$

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

Before functions, data

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

Before functions, data

Different kinds of data object

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

Direct binary values: Booleans (TRUE/FALSE in R)

Before functions, data

Different kinds of data object

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

Direct binary values: Booleans (TRUE/FALSE in R)

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

Before functions, data

Different kinds of data object

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

Direct binary values: Booleans (TRUE/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

Before functions, data

Different kinds of data object

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

Direct binary values: Booleans (TRUE/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

More about floating-point numbers

The more bits in the fraction part, the more precision

The R floating-point data type is a `double`, because the now-standard number of bits used to be twice the standard precision (back when memory was more expensive)

a.k.a. `numeric`

More about floating-point numbers

The more bits in the fraction part, the more precision

The R floating-point data type is a `double`, because the now-standard number of bits used to be twice the standard precision (back when memory was more expensive)

a.k.a. `numeric`

Finite precision means arithmetic on `doubles` doesn't match arithmetic on real numbers

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

More about floating-point numbers

The more bits in the fraction part, the more precision

The R floating-point data type is a `double`, because the now-standard number of bits used to be twice the standard precision (back when memory was more expensive)

a.k.a. `numeric`

Finite precision means arithmetic on `doubles` doesn't match arithmetic on real numbers

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

Often but not always ignorable; rounding errors tend to accumulate in long calculations

Particularly troublesome when results should be close to zero, since then errors could flip signs, etc.

```
> 0.45-3*0.15
[1] 5.551115e-17
```

Operators

Unary operators: - 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`?)

Finite precision leads to weirdness with floating points and exact comparisons; `all.equal()` can usually deal with it:

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

Boolean operators, for "and" and "or":

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

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
```

Integers: subtleties

Creating a whole number in the console 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
```

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 read for others, easier to modify later, easier to design, less prone to errors
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 69 when I mean 60?)
- obscure if we come back to our work later (what are any of 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
```

Notice 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

Summary

This class will teach you how to program for data analysis
We write programs by composing functions to manipulate data
Basic data types (Booleans, characters, numbers) and their functions
Basic data structures (vectors, arrays) and their functions
Using variables, rather than constants, is the first step of abstraction
Next time, more data structures: matrices, lists, data frames, structures of structures