Teaching an Introduction to Computing

Recap

Computing is one important avenue for

- a broader of collaborative statistician/scientist and impact on science, ...
- giving undergraduates skills that make them useful in data preparation, analysis
- broadening the pool of potential students of statistics
- teaching statistics in "applications first" approach
- allowing graduate students to do better research.
- even for consultants, often want to provide a reusable analysis process, with visualization tools for new data.

Intro. Stat.

"10 years ago, I wouldn't have dreamed of using R in an introductory statistics class; now I wouldn't dream of not!" - Doug Bates

Why teach histograms when we can do show them empirical densities/smoothed histograms.

Why teach probability tables when students will only ever use a computer to find them?

Programming concepts and understanding of the essentials of programming languages form the basis of computing.

Think about the material as if it were the first time you had seen it and what you would or wouldn't understand, find confusing, ambiguous or interesting.

Draw analogies and connections with other languages which you use.

Ask questions about more complex, deep issues as they pertain to the current topic. (Interrupt!)

Goals

To outline what aspects of a language (specifically R) you might want to teach your students

Discuss how to teach this effectively
concepts, examples, motivation, context

Discuss aspects of the language that not all users a familiar with
and to show the logic/simplicity of the language

Please interrupt as you have comments, questions, etc.

Three aspects to statistical programming

- interactive use for exploratory data analysis
- programming numerically intensive tasks
- writing functions/software for reuse

Obviously related, and there is a natural progression.

If we are allocating a single lecture or two to R or MATLAB or SAS, then we need to teach the essentials

Show enough so students can run code you supply, or mimic and adapt code to do slightly different things.

Alternatively, let the students learn for themselves?

Is this really effective? efficient?

- more, but still few books for students to see statistical computing in action,
  many books on statistics!
- students learn minimum & details, not the concepts
  so limited perspective, understanding of possibilities.

Some students won't have seen a "programming language" before

Some may have seen HTML - a declarative language that has no control flow.

Many will be unfamiliar with the command-line REPL - Read-Eval-Print-Loop

Explain the difference between a spreadsheet and a programming environment.

For some students, we have to get them past the "why isn't this like X? It's so lame!"

So need to show them how it is useful, and how the lanuage constructs help them to do things.

So we have to get them to want to do something which involves, e.g. a lot of repetition.
e.g. give them a corpus of e-mail messages, 1 per file, or person per file and ask how many messages do we have?

Get them to consider the language as a servant.

Language Model

Critical to teach the concepts, structure, logic and design of the language

Students will often gravitate to "how to" and mimicing rather than abstracting the specific to gain this "higher understanding" of the language.

Syntax versus Semantics and computational model.

Getting past details allows them to

- reason about problems
- apply their knowledge of one language to another.

What language(s) to use?

How to teach the language

- unlearn other languages!

Identifying the fundamentals

Connections with other languages.

Essentials of a language

Syntax

Data types

Control flow

Semantics

Scope

Libraries

Most of the languages we will use are

- interpreted
- high-level
- garbage collected
- "interactive"
- & have large libraries

R, MATLAB, ... - triple of

- language
- interpreter
- environment (packages)

This session

getting started

assignments & variable lookup

everything is an object & self-describing type

function calls, ...

data types

recycling rule

attributes

vectorized computations

family of apply functions

control flow

Data Types

Vector

- logical, integer, numerical, character, complex
- names
- subsetting

factor, ordered factor

matrix, array

- attributes

List

data.frame

Getting started

Get students familiar with the basics of the environment.

How to start, quit, get help (help.start)

It is useful to show them the elements of a help page and how to read them,
 interpret some of the "ambiguous" content,
 explain some of the terms
 follow the See Also
 connect to the manuals which give some more details

Show them the continuation prompt

```
> 2 *
+  3
```

Do arithmetic - 1 + pi

Simple plots curve(sin, -pi, pi)

Assign results to variables

```
  x =  1 + pi
  print(x)
  x          result of evaluating a non-assignment => print
```

No declarations of variables needed
or type information

Self-describing objects

We don't need type declarations because every object in R (MATLAB, Python, ..., not C) is self-describing.

```
class(1)
class(1L)
class(x)
class(class)
```

Also mode() and typeof()

Explore assignments

- stored in session area - global environment
- see names of existing variables - objects()
- where does curve come from?
  - find("curve")
  - search() and the concept of the search path
    - how does R use this.

What about `+' in 1 + 2?
Does R just know about that?

find("+")
  "package:base"

1 + 2 is actually a function call
  `+`(1, 2)

In fact, everything in R is a function call
  simple, single concept that makes lots of things easy to reason about, and several computational tasks feasible.

Notion of function call is similar to other languages
e.g. shell - find . -name '*.R'

Everything's an object

If we can see 'x', can we see '+'?

Print the value of + - poor choice!
 `+`
 sin

We can pass a function as an argument to a function
  body(sin)

So functions are first class values.

In fact, every value is a first class object.

Function Calls

What's a function?

- for now, a black box that takes zero or more inputs, and returns a value, an object.
- leads to 2 issues:
  - how do we specify the inputs
    parameter matching
  - what sort of values can we pass in and get out?
    data structures

Parameter matching

R has a very different and flexible way of passing arguments to functions

- by position
- by name
- by partial name

Default values for some or all parameters

And in some cases, even parameters that do not have a default value do not need to be specified.

argument matching

3 steps to argument matching.

match all the named arguments that match a parameter name exactly
(match & remove these)

match all the named arguments that partially match ambiguous matches (matches 2 or more parameters) is an error
(match & remove these)

match the remainder arguments to the unmatched parameters by position

others picked up by ... if present

raise error if any left over

Argument matching

Consider the function rnorm()
function (n, mean = 0, sd = 1)

rnorm(10)
mean, sd take default values

rnorm(10, 2) - mean = 2, sd = 1 - default value

rnorm(10, , 2) - mean = 1, sd = 2
rnorm(10, sd = 2)
rnorm(10, s = 2) # partial matching

rnorm(sd = 2, 10) # n = 10, sd = 2

...

Some functions have a parameter named "..."

This means zero or more arguments which do not match any of the other parameters.

Two purposes for ...

collect arbitrary number of arguments together to be processed together,
e.g. sum(..., na.rm = FALSE)
elements are collected into a vector and added
(compare with sum(x) or sum(x, y))

...

arguments that are passed directly to other functions called by this function
e.g. used a lot in graphics functions to pass common named arguments like col, pch to lower-level functions.

Or

lapply(1:3, rnorm, sd = 3)

lapply accepts additional arguments via its ... parameter, and passes them in the call to

FUN(x[i], ...)

R inlines the ... in the call to the function, with the names and all.

Argument matching

arguments that do not match by name or position are collected into the ... "list".

In most languages, ... must come last.
But in R, some functions have additional parameters after the ...

users must specify values for these parameters by using the fully specified parameter name name, e.g.

```
cat(..., file = "", sep = " ")
cat("some text", " to display", file = "myFile", sep ="")
```

Copying objects

Create a collection of 10 random normal values
x = rnorm(10)

Assign x to new variable y
y = x

This is a copy of x
(actually both x and y initially point to a single shared array in memory)

But if we change x, e.g.

```
x[1] = 100
```

y retains its current value and x and y are different.

Copying arguments

When x[1] is evaluated, R recognizes that the data elements are shared by two or more variables and so makes a copy of it and then modifies the first element of the copy. Then it assigns the result to 'x'.

The original shared copy of the vector remains unchanged and is tied to y.

So the computational model is that assignments (almost always) appear to make a copy of their data.

So in a function call,
  foo(x, y)
we create a new frame
evaluate x in the caller's frame
assign the value to the first parameter
Hence we are making a copy of that value.

Any changes made to the first parameter within the body of the function will have no effect on the variables in the caller's frame.

All changes of interest to the caller must be explicitly returned by the function.

Lazy evaluation

In many languages, f(x, y+2, z+3) would first get x, y+2, z+3 and then pass those in the function call.

Unusual feature of R is that expressions passed to a function are not evaluated until the parameter is actually used - lazy evaluation.

Avoids unnecessary computations and copies of data.

Has implications for writing code
if expression has side effects, not certain when or if it will occur

and implications for writing functions

Data Types

Major shift from C/C++, Java, Python, Perl, ...

No scalar (individual value) types

Only containers of values - vectors & lists.

vectors - ordered collections of homogeneous elements
integer, numeric, logical, character, complex

Numbers are vectors of length 1

So everything has a length - length(obj)

Creating vectors

Create empty vectors
 numeric(10) - numeric vector of length 10
 logical(3)

each element initialized to default value (0 in approriate form)

result of function calls
  rnorm(10)

sequences very common:
  3:8 <==> c(3, 4, 5, 6, 7, 8)
  seq(1, 100, by = 10)

combine elements manually
  c(1, 2, 3)

Coercion

Unlike C, R does a lot of implicit coercion to do what it "thinks you mean"
Most of the time very sensible.

c(1L, TRUE)
c(1L, 2.0)
c(TRUE, 2.0, "3.0")

Can also do explicit coercion to a particular type
 as.<type>(obj)

as.numeric(1:10)
as.character(c(1, 2, 3))

Missing Values

We represent missing values with an explicit, unambiguous value rather than having the user/data use an arbitrary value such as "999" or "0"

This is value is the variable NA and R recognizes it as a missing value.
Different from "NA" - a literal string.

We can use it in expressions
```
c(1, NA, 3)
```

Logic of NA computations

Find all the NAs
```
c(1, NA, 10) == NA
[1] NA NA NA
```

```
c(1, 2, 3) + NA
[1] NA NA NA
```

We'll come back to this - recycling rule

So how do we find the NAs

```
is.na(c(1, NA, 10))
```
returns a logical vector

names()

A very useful notion in statistics is that elements/records can be associated with identifiers

And so vectors can have an associated character vector of names

```
c( a = 1, b = 2, c = 3)
names(c)
```

Very relevant for extracting subsets.

Subsetting

One of the more powerful aspects of R is the flexible subsetting.

5 different styles of subsetting

```
x = c(10, 20, 30)
```

By index/position

```
x[ c(1, 3, 2) ]      ->  c(10, 30, 20)
x[ c(1, 2) ]         ->  c(10, 20)
```

0 and NA are special

```
x[ c(0, NA)]  --> NA
```

By logical value
    select only elements for which corresponding subset operand is TRUE
   x[c(TRUE, FALSE, TRUE)] -> c(10, 30)

Use logical operators, !, &, |

```
x = c(1, NA, 10)
x[ ! is.na(x) ]

x = rnorm(1000)
x [ x < 1.96 & x > -1.96]
```

Note the & rather than &&

Negative indexing to drop particular elements

c(1, 2, 3)[-2] --> c(1, 3)

No reall need and can't mix positive and negative indices
e.g. c(1, -2, 3)

By name

x = c(a = 1, b = 2, c = 3)

x[c("a", "b")]

Empty []

Empty indices

x = c(1, 2, 3)

x[]

Get's everything.

Important for element-wise assignment
x[] = NA

matrices

[, drop = TRUE]

Factors

Lists

[[, $

Data.frames

Recycling Rule

```
paste("A", 1:3)
```

Puzzle

```
x = c(1,2,3)
  x[c(TRUE, NA)]
  x[c(1, NA)]
```

Vectorized Computations

We tend to operate on all (or a subset) of the elements, and so do this in a single operation

```
mean(myData)
median(myData)
summary(myData)
hist(myData)
```

R is vectorized, and there is a real benefit to using vectorized

Often see people looping to perform computation

```
ans = 0
for(i in x) ans = ans + x[i]
ans/length(x)
```

Let's compare timings

```
x = rnorm(1000000) - a million N(0, 1)

system.time(sum(x))
   user  system elapsed
  0.002   0.000   0.003

system.time({ans = 0; for(i in x) ans = ans + x[i]})

.................
```

Okay, let's try something smaller

```
x = rnorm(10000) - 10K
system.time(replicate(10, {ans = 0;
                           for(i in x) ans = ans + x[i]
                          }))
  user  system elapsed
  8.074   0.000   8.075

system.time(replicate(10, sum(x)))
  user  system elapsed
  0.002   0.000   0.001     - (inaccurate)
```

6000 - 8000 times slower

Get's worse as x gets longer!

Family of apply functions

Looping over elements of a vector can be expensive, so use internally implemented functions.

But when those functions don't exist, use apply()

- Just like loops, but higher level abstraction and slightly shorter code.
- puts names on the result if names on the object over which we iterate
- easier to read as clear that the iterations don't depend on each other
- Makes this potentially programmatically parellalizable

For vectors, lapply() does the loop for us and return a list.

Generate samples of different sizes
lapply(1:3, rnorm)

Note we can add additional arguments to the function calls via ...

- lapply(1:3, rnorm, sd = 10)
- equivalent to
  lapply(1:3, function(x) rnorm(x, sd = 10))

Often, the results for each element are simple scalars or atomic data types of the same length

the result can be simplified to a vector or a matrix.

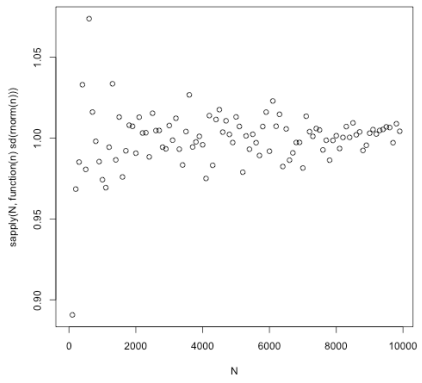
sapply() is a wrapper to lapply() that attempts to make this simplification.

```
sapply(mtcars, range)
      mpg cyl  disp  hp drat    wt qsec vs am gear carb
[1,] 10.4   4  71.1  52 2.76 1.513 14.5  0  0    3    1
[2,] 33.9   8 472.0 335 4.93 5.424 22.9  1  1    5    8
```

N = seq(1, 10000, by = 100)
plot(N, sapply(N, function(n) sd(rnorm(n))))

Note the anonymous function

Each returns a single number



apply() for matrices

For matrices, want to be able to loop over either

rows - 1 or columns - 2

apply(matrix, dimension, function)

```
x = matrix(rpois(4, 10), 2,2)
     [,1] [,2]
[1,]   10   14
[2,]    9    5

apply(x, 1, max)
[1]  14 9

apply(x, 2, which.max)
[1]  2 1
```

apply() generalizes readily to arrays

apply(array, c(dim1, dim2, ...), function)

mapply

lapply/sapply operate over a single vector or list and essentially ignore the order in which they operate on the elements

Sometimes want to iterate over two or more vectors/lists and call a function with the corresponding elements from each, i.e.
  f(x[i], y[i], z[i] ...)

mapply() does this, with the arguments in revere order from lapply/sapply.

mapply(FUN, x, y, z)

```
n = c(20, 30, 40)
mu = c(1, 10, 20)
sd = c(1, 3, 5)

mapply(rnorm, n, mu, sd)
```

Return value is a list.
Can simplify it with SIMPLIFY = TRUE

tapply/by

Groups the "records" of one object based on unique combinations one or more factors and applies a function to each group

Table apply
Like GROUP BY in a SQL

```
options(digits = 3)
tapply(mtcars$mpg, mtcars$cyl, mean)
   4    6    8
26.7 19.7 15.1
```

with()

Having to repeat the data frame in which two or more variables can be found is tedious, error-prone and doesn't generalize well
e.g. tapply(mtcars$mpg, mtcars$cyl)

In the past, people would use attach() to make the variables of a data.frame available via the search() path.
(Bad idea!)

Now, use the with() function

```
with(mtcars, tapply(mpg, cyl, mean))
```

Accepts a data frame, list or environment as the context in which to find the relevant variables.

Control Flow

For interactive use, one can avoid control-flow for a long time, using vectorized functions, *apply(), replicate().

Control flow brings us close to programming and developing functions.

However, they are language constructs that can be used at the prompt:

- if-else (& the function ifelse())
- for()
- while() & repeat { }
- switch()

Expressions

R commands, "programs", function bodies, control flow bodies, etc. are made up of expressions

simple expressions: f(x, y + g(2), n = n)

compound expressions:
{ $expr_1$; $expr_2$; .... ; $expr_n$}

separated by ; or on separate lines.

Every expression has a value

for simple expression, just the result
if a top-level assignment, marked as invisible()

```
1 ; 2
```

for compound: result of last expression evaluated

```
{ 1 ; 2}
```

And with

```
{  x = runif(1)
   if(x > .5)
       return(1)
   y = 2
   x + y
}    - value is either 1 or x + y
```

If no expression is evaluated, the result is NULL.

```
f = function(){}
f()
```

```
if(FALSE) 1.0
```

if-else

Same if(condition)
expression
else
expression
as in many languages

else part is optional (i.e. if(cond) expression is okay)

```
if(x > 1) 1 else if(x < 1) 2 else 3
```

if() else is one of the very few places that interactive language differs from that in scripts/files

```
 if(x < 1)
   1
 else
    2
```

REPL won't wait for the else

REPL won't wait for the else!

if-else

A somewhat unusual feature of if-else is that it is an expression and so returns a value
and we can assign the result to a variable

C, Java, ...: if(x < 1) y = 2 else y = 3

But in R

```
y = if(x < 1) 2 else 3
```

if(condition) ...

The condition must be a scalar logical,
i.e. logical vector of length 1

Often times, we end up with the evaluation of the condition returning a logical vector with more than one element.

Two reasons:

- should use any() or all() for the correct condition
- mistaking & and && or | and ||

for() loop

Prefer apply() or vectorized functions to explicit looping, but when iterations depend on each other often convenient & clear - never essential.

Higher iteration than in C, R can loop over elements of a container

for(var in container) expression

For each iteration, var is assigned the next element of the container and the expression evaluated with var bound to that value.

next, break

Within the body of the for() loop (i.e. the expression), can skip to the next iteration without evaluating the remainder of the body's expressions using

   next

(continue in C)

And to terminate the loop prematurely, use break

for(i in 1:4) { if(i == 2) break; print(i)}

while(cond) expression

For loops over the elements of the container present when the evaluation of the for(var in x) ... is started. So finite

Often want to repeat doing something until a condition changes, e.g. convergence of an optimization, etc.

Usual while(condition) expression construct

Often
```
while(TRUE) {
    x = f(value)
    if(test(x)) break
     ...}
```

Can do the test the condition: while(test(x)) ...., but order of first test matters. No do ... while

repeat{}

But there is a repeat {}

Almost always written as
```
   do something
   if(condition) break
```
or
```
    if(condition) break
    do something
```

switch()

if(cond) expr else if(cond) else if(cond) ...
starts to get tedious if cond is testing the value of the same variable

Essentially have a collection of expressions associated with different values of a variable, so a table.

switch(expr, value=expr, value=expr, ...)
switch(expr, expr-1, expr-2, expr-3, ...)

If expr returns a string, match value to names of other arguments (i.e. the value =), evaluate matching expr.

If expr returns an integer i, evaluate expr-i

```
centre =
    switch(type,
           mean = mean(x),
           median = median(x),
           trimmed = mean(x, trim = .1))
```

Random number generation algorithm by name.

Also, see match.arg()

Why is R slow? Interpreted Languages

In approximate order of importance

- Bad programming by users!
- Copying data
- Boxing and unboxing of scalars
  - interpreted - not compiled & no type information
- Argument matching

Copying Data

Because of this copy on assignment and so copy arguments when passed in a function call, there are potentially lots & lots of copies of data

Slows computations & machine down by

- spending time [garbage collecting], allocating memory and copying
- build up of memory that often needs to be swapped in and out from disk.

Only copy when it has to (copy on change/write), but conservative

Boxing & Unboxing of scalars

for(i in x) ans = ans + x[i]

enormous amount of repeated computations in accessing each element of x
check the type each time

Vectorized form that determines the type just once and access the internal array (in C) of elements natively avoids repeating this.

So functions that are implemented as .Primitive(), .Internal() and with .C/.Call/.FORTRAN are "fast" as

- they can exploit this low-level internal representation
- they use a machine-code compiled language

Compilation & Type specification

Languages such as C/C++/FORTRAN are compiled to machine instructions
Java to byte code that run on a virtual machine (VM)
(an abstracted computer running as an app. on a computer)
Python, Perl also to a VM

Resulting code has much more information about operations & context than evaluating/interpreting each expression one a time.

R has to look up the +, [function each iteration of
for(i in x) ans = ans + x[i]

So if run multiple times, cost of compilation becomes marginal

If R had (optional) type specification with which we could annotate function parameters, local variables, return types with their explicit types,
we could generate code that would

- can streamline the computations to avoid unboxing
- put values in registers providing very fast access to values for the CPU
- find the appropriate method/function ahead of time for + & [in our loop example.

Argument matching

Function calls are expensive.

Matching exact names, partial names, by position and then the elements of the ... parameter

4 steps so 4 loops (over small and increasingly smaller collections)

If done inside a loop a lot of times, this can become expensive.

But still not even remotely a good reason to not split code into separate functions - good software engineering.

Kernighan & Pike - TPoP

Style
(Write code to be read by a human. It may be you.)

Use descriptive names for variables & functions,
short names for local variables

Indent code appropriately

Parenthesize to resolve ambiguity

Define "variables" for constants that might change!

Using existing functions
Spend time searching.

Style

Comment code & expressions.
Don't belabour the obvious

Don't comment bad code, rewrite it.

Don't contradict the code in comments

Clarify, don't confuse.

Testing

Verify code is doing what you expect, for all cases.

Spend time developing tests for your code.

Create self-contained tests.

Compare independent implementations

Test incrementally, not at the end.

Adapt your estimates for how long something will take, know that it won't be right the first time.

Portability

If students send you code to run, they need to ensure that is portable.

R is mostly portable

But write code that handles

- references to external data,
- directories, file systems, line endings can be difficult
- graphics devices
- different languages, locales, encodings.