02 - Reviewing and Revisiting R

BIOS:7330 - Grant Brown

1. Intro/Basics
2. Data Structures
3. Part 2: Functions and Environments
   - Functions - Introduction
   - Functions as Objects
   - Special Functions
4. Warnings and Errors

Section Contents

1. Intro/Basics
   - Stupid R Fact of the Day
   - R Performance Tip of the Day

Introduction

- This section isn't an introduction to R, since students are expected to already be productively working with R as a prerequisite to the course.
- If you want to do a more basic review, there are many free resources online, including some linked from ICON/labs
- The purpose of this set of notes is to revisit some R concepts in a bit more detail than you might have seen previously, or at least in a way which will highlight useful topics for improving your R code, and which will set us up for more advanced material to come.

Stupid R Fact of the Day

- I really enjoy working in R, but there are a few design choices which I dislike, especially when using it to write general purpose programs, rather than one-time-use scripts or data analyses.
- I'll try, throughout the course, to point out some of these issues
- Can you see the problem with the following code?

```
# Dummy function to calculate the means
# of the numeric columns of a data frame
num_col_mean <- function(df){
  is_numeric <- sapply(df,class) == "numeric"
  apply(df[,is_numeric], 2, mean)
}
```

Stupid R Fact of the Day

```
# Works:
data(iris)
num_col_mean(iris)

# Broken
iris2 <- iris[,c(1,5)]
num_col_mean(iris2)
```

Stupid R Fact of the Day

- In statistics, we often work with data of multiple dimensions.
- It's easy to 'slice' or subset 2+ dimensional data in R using square bracket notation - but there's a problem
- When we subset by row or column, we generally expect to get back an object which has rows and columns. The first column of an $n \times p$ matrix is naturally represented as an $n \times 1$ column-vector/matrix.
- R doesn't do this, unles we specify: 'drop = FALSE'
- **Exercise:** When might the default indexing behavior be desired/helpful?

Stupid R Fact of the Day

```
# Dummy function to calculate the means
# of the numeric columns of a data frame
num_col_mean <- function(df){
  is_numeric <- sapply(df,class) == "numeric"
  apply(df[,is_numeric,drop=FALSE], 2, mean)
}
```

Stupid R Fact of the Day

```
# Works:
data(iris)
num_col_mean(iris)

# Works
iris2 <- iris[,c(1,5)]
num_col_mean(iris2)
```

R Performance Tip of the Day

Don't grow objects - especially not by overwriting them

- Many languages emphasize the ways in which objects can be modified (often via syntax like: `object.sort()`, and `object.append(new)`)
- In R, this is discouraged, but not impossible (as we'll see later).
- Many users naturally think in this way, however, so they use constructs of the form inside loops:
  1. `X <- c(X, new)`
  2. `X <- rbind(X, new)`
  3. `X <- cbind(X, new)`
- This is bad.

R Performance Tip of the Day

```
system.time({
  out <- c()
  for (i in 1:100000){
    out <- c(out, i)
  }})

system.time({
  out <- numeric(100000)
  for (i in 1:100000){
    out[i] <- i
  }})
```

Section Contents

2 Data Structures
- Atomic Vectors
- Heterogeneous Data Sources

Source Material

Much of this material is either form or informed by from Hadley Wickham's 'Advanced R' book: Advanced R: Data Structures, so all license terms apply.

Data Structures

	Homogeneous	**Heterogeneous**
1d	Atomic Vector	List
2d	Matrix	Data Frame
nd	Array	

Atomic Vectors

- Atomic vectors are one of the simplest data structures in R, but have some surprising features
- Strangely enough, things which you might believe to be 'primitive' data types, like numeric constants, are actually vectors of length 1 in R
- All of the elements of a vector are required to be the same type, and if vectors are combined using 'c', all elements are coerced to the least specific type (according to R): logical $\rightarrow$ int $\rightarrow$ numeric $\rightarrow$ character, etc.
- The family of 'as.__' functions exist to convert between these data types.

Atomic Vectors

- Accessing the 0 element of a vector returns an appropriately typed vector of length 0, which is a common source of errors, especially: `argument is of length zero`.
- R has a handfull of logical values in addition to the usual `TRUE` and `FALSE`, and special numeric values which often end up being used interchangably: `NA, NaN, NULL`
- Useful functions to know, and to experiment with: `is.finite`, `is.na`, `is.nan`, `is.null`
- Sometimes, it's also useful to check the 'class' of something. If you're expecting a 'numeric' value, and you end up receiving a 'logical' value, something has blown up in your code.
- You may also consider using the `identical` function, which always returns a TRUE or a FALSE, unlike the logical equals: "==".

Atomic Vectors

- R has numerous functions beginning with `is.` and `as.`, used for testing whether an object has certain characteristics, and for attempting to convert one object to another type of object.
- Check out the reading material for more details; there are too many to discuss in class
- The last thing to mention about atomic vectors will get a lot more useful when we discuss generic programming: attributes
  - Attributes are 'named' values associated with a particular object, and can be added using the 'attr' function, or the 'structure' function
  - Available attributes can be listed with the 'attributes' function
  - Special attributes used by R include: 'dim', 'names', 'nrow', 'ncol', 'colnames', 'rownames', 'dimnames'

Lists

- Lists are super great (IMHO)
- A list is also a vector, but not an atomic one - members of a list can have different types (including lists of lists).
- Elements of a list can be accessed by name (as in a key-value store), or by index
- By default, you can think of lists as having a single dimension, but this is not a requirement!
- See Example: lists

Data Frames

- Many students confuse data frames and matrices, though they're actually quite different
- 'a data frame is a list of equal-length vectors' (Advanced R)
- As a special case, this may map onto a matrix of numbers - but not necessarily.
- Remember, lists are also a type of vector, so data frames may be considerably stranger than a simple matrix with names attached

Section Contents

3 Part 2: Functions and Environments
- Stupid R Fact of the Day
- R Performance Tip of the Day
- Functions
- Environments

Stupid R Fact of the Day

```
> min()
[1] Inf
Warning message:
In min() : no non-missing arguments to min; returning Inf
```

- When an argument of length zero is passed to the `max` or `min` functions, a non-finite answer is returned.
- This is inconvenient, because one might usually expect computation on missing values to produce missing values.
- In particular, infinite covariate values tend to break statistical procedures, so if data processing steps compute minimums or maximums before a model is fit, this error can sometimes occur.
- **Exercise:** When might the default behavior of 'min' be desired/helpful?

R Performance Tip of the Day

Performace tip: don't write R code as if it's C code

- Students who first learn languages like C tend to use patterns in R which are inefficient.
- In particular, loops in R take a very long time compared to their counterparts in lower level languages. Consider the following two examples:

```
X <- rnorm(1000000); Y <- rnorm(1000000)
system.time(X%*%Y)
vmult <- function(a,b){
  out <- 0.0
  for (i in 1:length(a)){
    out <- out + a[i]*b[i]
  }
  out
}
system.time(vmult(X,Y))
```

R Performance Tip of the Day

- This gets a LOT worse when working with nested loops:

```
X <- matrix(rnorm(1000*1000), nrow = 1000)
Y <- matrix(rnorm(1000*1000), nrow = 1000)
system.time({M1 <- X %*% Y})
matmult <- function(A,B){
  out <- matrix(NA, nrow = nrow(A), ncol = ncol(B))
  for (i in 1:nrow(A)){
    for (j in 1:ncol(B)){
      out[i,j] <- vmult(A[i,], B[,j])
    }
  }
}
system.time({M2 <- matmult(X,Y)})
```

R Performance Tip of the Day

- Many mathematical functions provided by R, including the matrix operations, are already implemented efficiently in C or fortran
- When feasible, try to use these built in functions instead of more explicit R code.

Thinking about R

To understand computations in R, two slogans are helpful:

- *Everything that exists in as object.*
- *Everything that happens is a function call.*

-John Chambers

- To understand the implications of this quote, the reader must understand at least a little bit about functions and objects.
- Objects are collections of data stored in computer memory, for which there exist rules/instructions on how to interact with them.
- Functions are sets of instructions with defined inputs and computed outputs
- In this set of notes, we're going to focus on functions in R, and the environments in which they execute.

Functions

- Unlike many other languages, the 'function' keyword is not simply special syntax for declaring a function
- Instead, it's a special function devoted to creating other functions
- Functions created in this way are known as 'closures' in the functional programming paradigm.
- The other type of function available in R is a 'primitive' function, which includes special functions like '+'
- Most other languages have 'primitive' data types as well, but R does not.

```
# Open help page for 'function' function.
?`function`
```

Functions - Definition

```
# Function definition
f <- function(a,b,c,d,...){

}


# Function call
f(10.5, -2, c=10, d=20, e=10, f=20)
```

Positional Args. Named Args. Extra Args. (dots)

Functions

- One common mistake is to rely entirely on positional arguments.
- As a function is modified and tweaked, the order of these arguments is difficult to keep track of
- For mathematical functions in particular, this can lead to difficult to understand bugs/problems: prefer named arguments.
- This is especially true since R does not, by default, check argument types

Functions - dots

- Many built in R functions have the ability to specify a variable number of extra named arguments via the '...' option, and user defined functions can do so as well.
- The two most common patterns of use are to:
  1. Pass the '...' arguments to another function which is called in the function body
  2. Capture the '...' arguments by passing them to `list`, and process the resulting list.

Functions - dots example

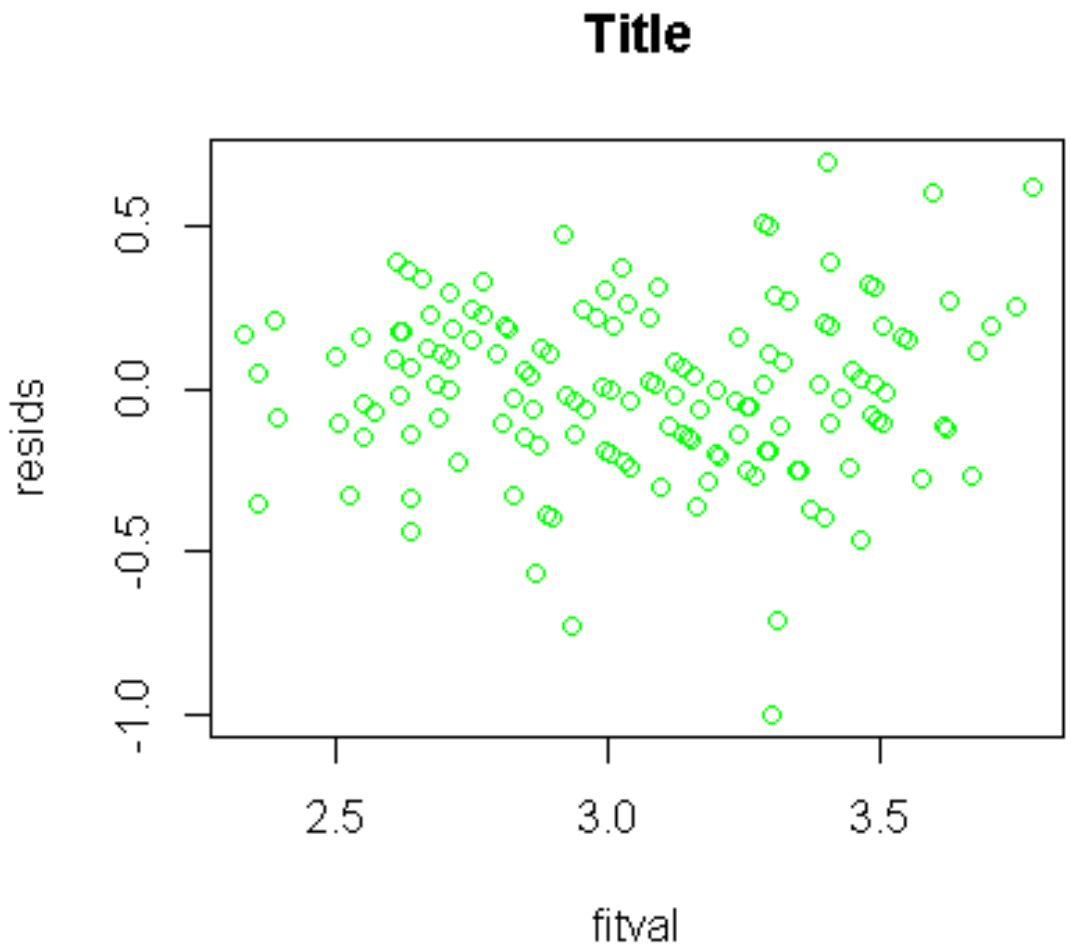
```
f <- function(...){
   argList <- list(...)
   unnamed <- sum(names(argList) == "")
   cat(paste0(unnamed, " unnamed arguments.",
       "\n"))
   cat(paste0(length(argList) - unnamed,
       " named arguments.", "\n"))
   for (n in names(argList)){
      if (n != "") cat(
         paste0("Argument ", n, " is: ", argList[[n]],
         "\n")
      )
   }
}
```

Functions - dots example

```
my_plot_function <- function(lmobj, ...){
  if (length(intersect(c("resids", "fitval"),
                       names(lmobj)))){
    stop("Missing residuals or fitted values")
  }
  resids <- lmobj$residuals
  fitval <- lmobj$fitted.values
  plot(resids ~ fitval, ...)
}

lm.out <- lm(Sepal.Width ~ ., data = iris)
my_plot_function(lm.out, col = "green", main = "Title")
```

Functions - dots example



Functions

- Beyond learning how to define and call functions, it's important to remember that functions are 'first class objects' in R - that is, they can be used in all of the ways other objects can
  1. Functions can be passed as arguments to other functions
  2. Functions can be elements of lists, atomic vectors, matrices, and data frames
  3. Functions have three important components accessible by the `formals`, `body`, and `environment` functions.

Functions - dots example

```
formals(my_plot_function)
body(my_plot_function)
environment(my_plot_function)
```

Functions

- These properties are modifiable using assignment: '<-'
- It's usually a bad idea to do so, however. If you need to provide additional functionality, or have R code treat some functions differently than others, attributes are a better way
- As with other R objects, function attributes can be discovered using the `attributes`, and accessed/set using the `attr` function.

Functions

Generally speaking, R functions are designed to have 'no side effects'; that is, to not modify the state of the computer. Exceptions include:

- `library`
- `setwd`
- `Sys.setenv`
- `Sys.setlocale`
- `plot`, etc.
- `write`, `write.csv`, `save`, `cat`, etc.
- `options`, `par`
- User defined functions which intentionally use environments to create persistent state change.

Functions - Infix

- We've been told that every computation is a function call in R, but there's still a bit of mystery about how functions like '+' work
- Such functions are known as using 'infix notation'
- Consider:
  - Infix: $A + B$
  - Prefix: $+AB$
  - Postfix: $AB+$
  - Function: $\text{add}(A, B)$

Functions - Infix

Common examples of infix functions in R:

- `+`, `-`, `*`, `/`, `%%`, `^` etc.
- `%*%`, matrix/vector multiplication
- `%in%`, to test membership
- `%x%`, Kronecker product
- `%o%`, outer product

All user defined infix function names must begin and end with '%'

Functions - Infix

- Defining infix functions is easy: we simply use the backquote to surround the name.

```
`%+%` <- function(a,b){
  paste0(a,b)
}
```

```
"This is a " %+% "string"
```

- These special functions can actually be called using the usual function notation as well, but it's not particularly common:

```
`+`(2,3)
```

Functions - Replacement

- Replacement functions are the reason why mysterious operations like the following work:

```
colnames(X) <- c("column 1", "column 2")
```

- Replacement functions are defined using the syntax:

```
`function_name<-` <- function(x, arg1, arg2, value)
```

- Replacement functions are sometimes syntactically convenient, but they are not particularly efficient, and work by replacing the underlying object with a modified version.

Environments

Source material: Advanced R: Environments

Environments

- If everything in R is an object, and every action is a function call, we still need to define how a function call accesses objects.
- Objects in R are organized into 'environments', which are nested (via parent/child relationships) data structures allowing R to find the appopriate named objects in different execution contexts
- The 'global' environment is the user workspace, where variables are defined in open code
- Environments are created automatically in some contexts, and can also be user created
- To find an object when called, R looks in the current environment, then the parent, then its parent, all the way up to the empty environment if need be.

Environments

- All names in an environment are unique, and non-ordered (like a set)
- All environments have parents (except emptyenv)
- Loaded packages live as parents of the global environment
- Environments copy/modify by reference, rather than by value
- Useful functions: `environment()`, `globalenv()`, `baseenv()`, `emptyenv()`, `search()`

Creating Environments

- There are several important environment types associated with functions:
  - enclosing, where a function was created. Accessed via `environment(f)`.
  - binding, where a function is currently stored in memory.
  - Executing environments, newly created each time a function is called
  - Caling environments, where a function is called
- There's really no substitute for experimentation when it comes to learning about environments: try out the examples on the Advanced R page.

Section Contents

4 Warnings and Errors
- Handling Stuff

Handling Conditions

- Lots of things can happen during the execution of a program which need to be *handled*, or reacted to.
- Sometimes, a program needs to simply stop executing when unexpected conditions occur, either to alert the programmer and allow them to fix whatever the issue may be, or to prevent damage to data and interruption of other work processes
- Other times, it is simply important to make sure a record of potential issues is recorded in the *l*og

Errors, Warnings, and Messages

- Creating (also called 'raising' or 'throwing' in various languages) conditions is easy in R:
  - The `stop` function interrupts the currently running code and generates an error message.
  - The `warning` function generates a warning message, either immediately, or after the current block of code is finished running, depending on the value of `options("warn")`
  - The `message` function generages a message which is printed to the screen.
- In R, all of these are known as 'conditions'
- Take a look at the code for the `lm` function to see how these can be used

Handling Conditions

- The method for programmatically reacting to conditions in R is a function called 'tryCatch'

```
result <- tryCatch({
           f()
         },
         message = function(m){
           print("Got a message")
         },
         warning = function(w){
           print("Got a warning")
         },
         error = function(e){
           print("Got an error")
         },
         finally = {
            print("Finally...")
         })
```

Handling Conditions

- Any combination of the `message`, `warning`, `error`, and `finally`, arguments can be passed to tryCatch
- If such a 'handler' is present, then any condition matching the handler type causes the code in the `expr` argument (the first argument) to stop running, and the appropriate handler is executed.
- tryCatch returns the value of the handler expression
- This is how 'warning' conditions are treated as errors in the course challenges: there is a warning handler in a tryCatch call, and it records the warning as the final result.

How do you actually... handle stuff?

- What you do in response to various conditions depends on what problem you're trying to solve.
- Sometimes, you need to re-start a process. Putting a tryCatch block inside a 'while' loop is a useful way to accomplish this, when you know a certain task can fail
- Other times, you need to simply record diagnostic data for later perusal (error messages, current values of relevant variables, etc.)
- It is often also useful to initiate some notification process, like an email or text message. (see example)