R Programming - Notes

Tanner Prestegard

Course taken from 3/2/2015 - 3/28/2015

Writing code / setting your working directory

- In R, you can use `getwd()` to get the current directory you're working in.
- dir() gives a list of files that are in your current directory.
- If you try to call files that aren't in your working directory, R will not be able to find them. So you can either copy the file into your working directory or change directories.
- For this course, it may be useful to put these files into a single directory to help keep things organized.
- To load code into R, use source("mycode.R").

History of R

- R is a dialect of the S language.
  - S was initiated in 1976 as an internal statistical analysis environment - originally implemented as a series of Fortran libraries.
- Features of R
  - Quite lean in terms of software, functionality is divided into modular packages.
  - Sophisticated graphics capabilities.
  - Userful for interactive work but also contains a powerful programming language for developing new tools.
  - Active and vibrant user community - R-help and R-devel mailing lists and Stack Overflow.
- Drawbacks of R
  - Essentially based on 40 year old technology.
  - Little built-in support for dynamic or 3D graphics.
  - Functionality is based on consumer demand and user contributions.
  - Objects must generally be stored in physical memory - can be a problem due to increasing size of datasets.
- The R system is divided into two conceptual parts.
  - The "base" R system that you download from CRAN.
    - Base package

    - Packages like: utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4.
    - There are also recommended packages: boot, class, cluster, codetools, foreign, KernSmooth, lattice, mgcv, nlme, rpart, survival, MASS, spatial, nnet, Matrix.
  - Everything else.
    - ~4000 packages on CRAN developed by users. Note: CRAN has quality standards for their packages.
    - Packages available on people's personal websites.

- Some R resources (all available on CRAN:
  - An Introduction to R
  - Writing R Extensions (for developing R packages)
  - R Data Import/Export (useful for getting data into and out of R)
  - R Installation and Administration (useful for building R from source)
  - R Internals (very low level)

R Console Input and Evaluation

- Assignment operator: <-
  - Example: x <- 1 assigns the value 1 to x.
  - Example: msg <- "hello"
- Comments: #
- Auto-printing: assign a value to x, then just enter x. The value will be auto-printed.
- Sequence: x <- 1:20

R Objects and Attributes

- R has five basic or "atomic" classes of objects.
  - Character
  - Numeric (real numbers) (decimals)
  - Integer
  - Complex
  - Logical (true/false)
- The most basic object is a vector.
  - A vector can only contain objects of the same class.
  - A list is represented as a vector but can contain objects of different classes.
- Empty vectors can be created with the vector() function.
  - First argument: class of objects in vector.
  - Second argument: length of vector.
- Numbers
  - Generally treated as double precision real numbers.

  - If you explicitly want an integer, you need to specify the L suffix: 1L.
  - There is also a special number Inf which represents infinity (1/0).
  - The value NaN represents an undefined value.

- R objects can have attributes.
  - Names, dimnames
  - Dimensions
  - Class: numeric, character, etc.
  - Length
  - Other user-defined attributes/metadata.
  - Attributes of an object can be accessed using the attributes() function.

R Vectors and Lists

- The c() function can be used to create vectors of objects.
  - Ex: x <- c(0.5, 0.6)
  - Ex: x <- c(TRUE, FALSE)
  - Ex: x <- c("a","b","c")
- Using the vector() function:
  - Example: x <- vector("numeric", length = 10) outputs a vector of 10 zeros (0 is default value).
- When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.
  - Ex: y <- c(1.7, "a") is a character vector, so the first element will be a string: "1.7".
  - Ex: y <- c(TRUE, 2) is numeric because True/False can be represented as 1/0.
  - Ex: y <- c("a", TRUE) is a character vector.
- Explicit coercion - use the as.* functions:
  - Example: x <- 0:6 is an integer sequence, so we can do as.numeric(x), as.character(x), as.logical(x), etc.
  - Coercion doesn't always work - nonsensical coercion results in NAs and a warning.
- Lists - a special type of vector that can contain elements of different classes.
  - They are a VERY important data type in R.
  - Example: x <- list(1, "a", TRUE, 1 + 4i)

Matrices

- Matrices are vectors with a dimension attribute.
- The dimension attribute is itself an integer vector of length 2 (nrow, ncol).
- Example: m <- matrix(nrow = 2, ncol = 3)
  - Can do dim(m), attributes(m) to get attributes of the matric.
- Matrices are inserted column-wise: 1st column is filled, then 2nd column, and so on.

  - Example; m <- matrix(1:6, nrow = 2, ncol = 3) has column 1 = 1, 2; column 2 = 3, 4; column 3 = 5, 6

- Matrices can be created directly from vectors by adding a dimension attribute.
  - m <- 1:0
  - dim(m) <- c(2,5)
- Matrices can be created by column-binding or row-binding with cbind() and rbind()
  - Example:
    * x <- 1:3
    * y <- 10:12
    * cbind(x,y) gives a matrix with column 1 = 1,2,3 and column 2 = 10,11,12
    * rbind(x,y) gives a matrix with column 1 = 1,10; column 2 = 2,11; column 3 = 3,12

Factors

- Factors are a special type of data that are used to represent categorical data.
- Factors can be ordered or unordered. You can think of a factor as an integer vector where each integer has a label.
- Factors are treated specially by modeling functions like lm() and glm()
- Using factors with labels is better than using integers because factors are self-describing; having a variable that has values "Male" and "Female" is more descriptive than a variable that has values 1 and 2.
- Factors are created with factor(). Example:
  - x <- factor(c("yes","yes","no","yes","no"))
  - When you auto-print this vector, the levels are shown - levels are possible values in the factor.
  - table(x) gives the frequency of each value.
  - unclass(x) shows what is underlying the vectors ("yes" is represented as 2, "no" is represented as 1)
- The order of the levels can be set using the levels argument to factor().
  - x <- factor(c("yes","yes","no","yes","no"), levels=c("yes","no"))
  - This has "yes" represented by 1 and "no" represented by 2.

Missing Values

- Missing values are denoted by NA or NaN.
- NaN is used for undefined mathematical operations, NA is used for everything else.
- is.na() tests objects to see if they are NA.
- is.nan() is used to test for NaN.
- NA values have a class (integer, character, etc.)
- NaN values are also NA, but the reverse is not necessarily true.

Data Frames

- Data frames are used to store tabular data.
- They are represented as a special type of list where every element of the list has to have the same length.
- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.
- Unlike matrices, data frames can store different classes of objects in each column.
- Data frames also have a special attribute called row.names.
- Data frames are usually created by calling read.table() or read.csv().
- Can be converted to a matrix by calling data.matrix().
  - This may cause coercion if the data types are not the same!
- Data frames can also be created using data.frame()
  - x <- data.frame(foo = 1:4, bar = c(T,T,T,F))
  - nrow(x) gives number of rows.
  - ncol(x) gives number of columns.

Data Types - The Names Attribute

- R objects can also have names, which is very useful for writing readable code and self-describing objects.
  - x <- 1:3
  - names(x) is NULL at this point.
  - names(x) <- c("foo", "bar", "norf")
  - Now auto-printing x shows the names above each element (foo, bar, norf)
- List names
  - x <- list(a = 1, b = 2, c = 3)
- Matrix names (dimnames)
  - m <- matrix(1:4, nrow = 2, ncol = 2)
  - dimnames(m) < list(c("a","b"), c("c","d")) (this labels the rows as "a" and "b" and the columns as "c" and "d")

Reading Data Into R

- read.table(), read.csv() for reading tabular data.
- readLines() for reading lines of a text file.
- source() for reading in R code files (inverse of dump()).
- dget() for reading in R code files (inverse of dput()).
- load() for reading in saved workspaces.
- read.table() arguments:

  - file - name of file, or a connection.
  - header - logical indicating if the first line of the file is a header line.
  - sep - a string indicated how the columns are separated.
  - colClasses - a character vector indicating the class of each column in the dataset.
  - nrows - the number of rows in the dataset.
  - comment.char - a character string indicating the comment character.
  - skip - the number of lines to skip from the beginning.
  - stringsAsFactors - should character variables be coded as factors?

- For small to moderate sized datasets, you can usually call read.table() without specifying any other arguments:
  - data <- read.table("foo.txt")
  - R will automatically skip lines that begin with #, figure out how many rows there and how much memory to allocate, and figure out variable types.
  - It may run faster if you provide other information.
  - read.csv() is the same except it assumes the separator value is a comma (instead of a space).

Reading Large Tables

- Some things you can do to make your life easier and prevent R from choking when you use large datasets.
  - Read the help page for read.table, which contains many hints.
  - Make a rough calculation of the memory required to store your dataset.
  - Set comment.char = "" if there are no commented lines in your file.
  - Use the colClasses argument.
    - Specifying this option instead of using the default can make read.table run twice as fast!
    - If you only specify a single value for colClasses, it assumes all columns are this class.
    - A quick and dirty way to figure out column classes:
      - initial <- read.table("datatable.txt", nrows = 100)
      - classes <- sapply(initial,class)
      - tabAll <- read.table("datatable.txt", colClasses = classes)
  - Setting nrows doesn't make it run faster but does help with memory usage.

- Should know the following about your system:
  - RAM in your computer.
  - What other applications are in use?
  - What OS?
  - Is the OS 32 or 64-bit?
  - Are there other users logged into the same system?

- Roughly calculating memory requirements
  - Example: data frame with 1500000 rows and 120 columns, all of which are numeric data.
  - 1500000*120*8 bytes/numeric = 1440000000 bytes = 1440000000/2^20 bytes/MB = 1373.29 MB
  - Will need some overhead for reading in this data - probably about a factor of 2.

Textual Data Formats

- dump and dput are useful because the resulting textual format is editable, and may be possible to recover if corrupted.
- Unlike writing out a table or csv file, dump and dput preserve the metadata so that another user won't have to specify it all over again.
- Textual formats can work much better with version control programs which can only track changes meaningfully in text files.
- Downside: textual files are not very space-efficient. One option is to compress them.
- dput-ing R objects: provides a way to write R code that can be used to reconstruct an R object.
  - y <- data.frame(a = 1, b = "a")
  - dput(y) gives: structure(list(a=1, b=structure(1L, .Label = "a", class = "factor")), .Names = c("a","b"), row.names = c(NA, -1L), class = "data.frame")
  - dput(y, file = "y.r")
  - new.y <- dget("y.R")
- dump can be used on multiple R objects, unlike dput.
  - `y <- data.frame(a=1,b="a")`
  - `dump(c("x","y"), file = "data.R")`
  - `rm(x,y)`
  - `source("data.R")` to get x and y from the file.

Connections - Interfaces to the Outside World

- Data are read in using connection interfaces.
- Connections can be made to files (most common) or to other things.
  - `file` - opens a connection to a file.
    - `con <- file("foo.txt","r")`
    - `data <- read.csv(con)`
    - `close(con)`
  - `gzfile` - opens a connection to a file compressed with gzip.
    - `con <- gzfiles("words.gz")`
    - `x <- readLines(con, 10)`
  - `bzfile` - opens a connection to a file compressed with bzip2.
  - `url` - opens a connection to a webpage.
    - `con <- url("http://ww.jhsph.edu","r")`
    - `x <- readLines(con)`

Subsetting R Objects

- Several operators for extracting subsets in R.
  - [- returns a subset of the same class as the original; can be used to select more than one element (with one exception).
  - [[- is used to extract elements of a list or data frame and can only be used to extract a single element. The class of the returned object will not necessarily be a list or data frame.
  - $ - is used to extract elements of a list or data frame by name; semantics are similar to that of [[.
- Example:
  - `x <- c("a","b","c","c","d","a")`
  - `x[1]` returns "a"
  - `x[1:4]` returns "a" "b" "c" "c"
  - `x[x > "a"]` returns "b" "c" "c" "d"
  - `u <- x > "a"` returns `FALSE TRUE TRUE TRUE TRUE FALSE`
- Subsetting lists is a little different because you can use any of the 3 operators.
  - `x <- list(foo = 1:4, bar = 0.6)`
  - `x[1]` returns `$foo` and `1 2 3 4` (i.e., a list)
  - `x[[1]]` returns `1 2 3 4` (i.e., just a sequence)
  - `x$bar` or `x["bar"]]` returns `0.6`
  - `x["bar"]` returns `$bar` and `0.6`
- To extract multiple elements of a list, use the single bracket operator
  - `x <- list(foo = 1:4, bar = 0.6, baz = "hello")`
  - `x[c(1,3)]` return `$foo 1 2 3 4` and `$baz "hello"`
- The [[operator can be used with string variables where the name of the element is computed.
  - `name <- "foo"`
  - `x[[name]]` returns `1 2 3 4`
- Subsetting nested elements of a list
  - `x <- list(a = list(10,12,14), b = c(3.14,2.81))`
  - `x[[c(1,3)]]` returns `14`
  - `x[[1]][[3]]` returns `14`
  - `x[[c(2,1)]]` returns `3.14`
- Subsetting matrices - can use (i,j) type indices
  - `x <- matrix(1:6, 2, 3)`
  - `x[1, 2]` returns `3`
  - `x[2, 1]` returns `2`
- Matrix indices can also be missing:
  - `x[1, ]` returns first row: `1 3 5`

- By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1 x 1 matrix. Use `drop = FALSE` to turn this behavior off.
  - `x[1, 2, drop = FALSE]` returns `3` as a matrix.
- Subsetting a single column or row gives a vector by default, too. Can use `drop = FALSE` again here.
- Partial matching of names is allowed with [[and $.
  - `x <- list (aardvark = 1:5)`
  - `x$a` returns `1 2 3 4 5`
  - `x[[“a”]] returns NULL`
  - `x[[“a”, exact = FALSE]]` returns `1 2 3 4 5`

Removing NA Values

- Example:
  - `x <- c(1,2,NA,4,NA,5)`
  - `bad <- is.na(x)`
  - `x[!bad]` returns `1 2 4 5`
  - Note: exclamation point is NOT operator.
- Finding subset of multiple things with no missing values:
  - `x <- c(1,2,NA,4,NA,5)`
  - `y <- c(“a”,”b”,NA,”d”,NA,”f”)`
  - `good <- complete.cases(x,y)`
  - `good` returns `TRUE TRUE FALSE TRUE FALSE TRUE`
  - `x[good]` gives `1 2 4 5`
  - `y[good]` gives “a” “b” “d” “f”
  - Can use this to get only good rows or columns in a data file.

Vectorized Operations

- Can do operations on array elements in parallel (just like in Matlab).
- Can do this on matrices as well.
  - `x * y` is element-wise multiplication.
  - `x %*% y` is true matrix multiplication.

Control Structures - If, Else

- Basic structure:

```
if (<condition1>) {
        ## do something
} else if (<condition2>){
        ## do something
} else {
        ## do something
}
```

- Can assign value:

```
y <- if (<condition1>) {
        10
} else {
        0
}
```

Control Structures - For Loops

- Example:

```
for (i in 1:10){
        print(i)
}
```

- Example 2: all of these for loops are equivalent.

```
x <- c("a","b","c","d")
for (i in 1:4){
        print(x[i])
}
for (i in seq_along(x)){
        print(x[i])
}
for (letter in x){
        print(letter)
}
```

- Can nest for loops - example: loop over elements in a matrix.

Control Structures - While Loops

- Example:

```
count <- 0
while (count < 10 {
        print(count)
        cout <- count + 1
}
```

Control Structures - Repeat, Next, Break

- `repeat` initiates an infinite loop - these are not common but do have their uses. The only way to exit is to call `break`.

```
x0 <- 1
tol <- 1e-8

repeat {
        x1 <- computeEstimate()
```

```
        if (abs(x1-x0) < tol) {
                break
        } else {
                x0 <- x1
        }
}
```

- This is dangerous because there is no guarantee that the loop will converge.
- It's better to set a hard limit on the number of iterations and then report whether convergence was achieved or not.
- `next` is used to skip an iteration of a loop.
- `return` signals that a function should exit and return a given value. This can also be used to interrupt the flow of a program.

R Functions

- Example of function syntax:

```
add2 <- function(x,y=5) {
        x + y
}
```

- `y=5` assigns a default value to `y` in case the user doesn't specify `y`.
- Don't need a return statement - all functions return the value of the last expression.
- Functions are R objects of class "function."
- Use the function directive to define a function: `f <- function(<arguments) { ## do stuff }`
- Functions can be passed as arguments to other functions.
- Functions can be nested (defined inside of another function).
- Named arguments:
  - May potentially have default values.
  - *Formal* arguments are the arguments included in the function definition.
  - `formals` returns a list of all formal arguments of a function.
- R function arguments can be matched positionally, or by name:
  - `sd(x = mydata, na.rm = FALSE)` is the same as `sd(na.rm = FALSE, x = mydata)`
  - You can mix positional matching and matching by name. When are argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.
  - Named arguments are useful on the command line when you have a long argument list and you want to mostly use defaults for everything except an argument near the end of the list, or if you can remember the name of the argument but not its position.
- Function arguments can also be partially matched (useful if you don't remember the argument's exact name).

Defining a Function

- Function arguments can be set to `NULL` by default.

```
f <- function(a, b=1, c=2, d=NULL) {
}
```

- Arguments to function are evaluated *lazily*, so they are only evaluated when they are needed:
- Example: you can call this function with `f(2)`; b is not needed since it's never used and the 2 gets positionally matched to a.

```
f <- function(a, b) {
        a^2
}
```

- This also means that if an argument is missing, the code actually executes every line up until that argument is needed.
- The "`...`" argument is used to indicate a variable number of arguments.
  - Useful when you extend another function and you don't want to copy the entire argument list of the original function.
  - Example:

```
f <- function(x, y, type = "1", ...) {
        plot(x, y, type = type, ...)
}
```

  - This argument is also necessary when the number of arguments passed to the function cannot be known in advance.
    - For example, the `paste` function can take as many arguments as needed. If you were to extend this function, the "`...`" operator would be useful.
  - One catch: any arguments in the argument list must be named explicitly and cannot be partially matched.

Symbol Binding

- When R tries to bind a value to a symbol, it looks through the global environment and the packages in the search list.
- The order of packages in the search list matters!
- Users can configure which packages get loaded on startup.
- When a user loads a package with `library`, the namespace of that package gets put in position 2 on the search list and everything else gets shifted down.
- R has separate namespaces for functions and non-functions, so it's possible to have an object and function with the same name.
- Scoping rules determine how a value is associated with a free variable in a function.
  - R uses *lexical* or *static* scoping.
    - Example:

```
f <- function(x, y) {
        x^2 + y/z
}
```

  - Here z is not a formal argument or a local variable - it's called a *free variable*.
  - Lexical scoping means that the values of free variables are searched for in the environment in which the function was defined.
  - If the value of a symbol is not found in this environment, the search is continued in the parent environment.
  - The search continues down the sequence of parent environments until we hit the top-level environment; this is usually the global environment (workspace) or the namespace of a package.
  - After the top-level environment, the search continues down the search list until we hit the empty environment.
  - If a value for a given symbol cannot be found once the empty environment is arrived at, an error is thrown.

- Typically a function is defined in the global environment so the values of free variables are just found in the user's workspace.
- But in R, you can have functions defined inside other functions - in this case the environment in which the function is defined is the body of another function.
  - You can use this to define a function which constructs other functions.
- What's in a function's environment?
  - Can use `ls(environment(function))` to get this.
- In R, all objects must be stored in memory.
  - All functions must carry a pointer to their respective defining environments, which could be anywhere.

Optimization

- Some optimization routines: `optim, nlm, optimize`.
- May be useful to hold some parameters fixed and allow other to vary during the optimization.
- Useful solution - write a "constructor" function which contains all of the necessary data for evaluating the function.

Coding Standards in R

- Write code in a text file with a text editor.
- Indent your code to keep the flow of the program and different control blocks easily readable.
  - Suggested to set TAB to be 4 - 8 spaces.
- Limit the width of your code (80 columns?)
- Limit the length of your functions.

Dates and Times in R

- R has a special representation for dates and times.
  - Dates are represented by the `Date` class.
  - Times are represented by the `POSIXct` or the `POSIXlt` class.
    - `POSIXct` is just a very large integer under the hood
    - `POSIXlt` is a list underneath and stores a bunch of other useful information like the day of the week, day of the year, month, day of the month, etc.
  - Dates are stored internally as the number of days since 1/1/1970.
  - Times are stored internally as the number of seconds since 1/1/1970.
- Date example:
  - `x <- as.Date("1970-01-01")`
  - `unclass(x)` returns 0.
- There are a number of generic functions that work on dates and times.
  - `weekdays`: gives day of the week
  - `months`: gives the month name
- Time example:
  - `x <- Sys.time()`
  - `p <- as.POSIXlt(x)`
  - `names(unclass(p))` returns "sec" "min" "hour" "mday" "mon" "year" "wday" "yday" "isdst"
  - `c <- as.POSIXct(x)` gives "Already in 'POSIXct format"
- `strptime` can convert from string to a Date or Time object (just like in Python).
  - Check `?strptime` to get the formatting strings.
- You can use + and - on dates and times.
- You can use comparison operators, too.
- Can't always mix different objects (need to convert between classes before performing operations).
- Even keeps track of leap years, leap seconds, daylight savings, and time zones.

Loop functions - lapply

- `lapply` loops over a list and evaluates a function on each element.
- `sapply` is the same as lapply, but tries to simplify the result.
- lapply takes three arguments:
  - A list `x.`
  - A function or the name of a function.
  - Other arguments using the ... argument.
- The looping is done internally in C code.
- `lapply` always returns a list, regardless of the input class

- Example:

```
x <- list(a=1:5, b=rnorm(10))
lapply(x,mean)
$a
[1] 3

$b
[1] 0.0296824
```

- Example 2:

```
x <- 1:3
lapply(x,runif,min=0,max=10)
[[1]]
[1] 3.302142

[[2]]
[1] 6.848960 7.195282

[[3]]
[1] 3.5031416 0.8465707 9.7421014
```

- `lapply` and friends make heavy use of *anonymous* functions.
  - Example:

```
x <- 1:4
lapply(x, function(elt) elt[,1])
$a
[1] 1 2

$b
[1] 1 2 3
```

- `sapply` simplifications.
  - If the result is a list where every element is length 1, it returns a vector.
  - If the result is a list where every element is a vector of the same length (>1), a matrix is returned.
  - If it can't figure things out, a list is returned.

Loop functions - apply

- `apply` applies a function over the margins of an array.
- It is most ofter used to apply a function to the rows of columns of a matrix.
- It can be used with general arrays (e.g., taking the average of an array of matrices).
- It is not really faster than writing a loop, but works in a single line.
- Arguments:
  - An array `x` (recall that an array is a vector with dimensions attached to it).
  - An integer vector `margin` indicating which margins should be "retained".
    - Using `margin` = 2 will keep dimension 2 (i.e. it will apply the function over each column).

  - A function `fun` to be applied.
  - ... other arguments.

- Shortcut functions - highly optimized to do special operations quickly. Faster than actually calling apply.
  - `rowSums = apply(x,1,sum)`
  - `rowMeans = apply(x,1,mean)`
  - `colSums = apply(x,2,sum)`
  - `colMeans = apply(x,2,mean)`

Loop functions - mapply

- `mapply` is the multivariate version of `lapply`.
- Arguments:
  - `fun` - function to apply.
  - ... - arguments to apply over.
  - `MoreArgs` - list of other arguments to `fun`.
  - `Simplify` - indicates whether the result should be simplified.
- Example: the following two lines are equivalent.

```
list(rep(1,4), rep(2,3), rep(3,2), rep(4,1))

mapply(rep, 1:4, 4:1)
```

- `mapply` can be used to vectorize a function (i.e., allow vector inputs to functions that don't normally allow it).

Loop functions - tapply

- `tapply` ("table apply") applies a function over subsets of a vector.
- Arguments:
  - `x` is a vector.
  - `Index` is a factor or a list of factors.
  - `fun` is a function to be applied.
  - ... contains other arguments to be passed to `fun`.
  - `simplify` specifies if we should simplify the result.
- Uses factor variables to split the input vector into groups.

Loop functions - split

- `split` is not technically a loop function, but rather an auxiliary function that is typically used in conjunction with loop functions.
- It takes a vector or other obejct and splits it into gropus determined by a factor or list of factor.
- Arguments:
  - `x` is a vector or list or data frame.
  - `f` is a factor (or coerced to one) or a list of factors.
  - `drop` indicates whether empty levels should be dropped.
- Can use `gl` function to define factors.
- In some cases, using `split` then `lapply` can be equivalent to doing `tapply`. But the first method is more flexible in general.
- Example with data from Programming Assignment #1:

```
s <- split(airquality, airquality$Month
## above we use the month as a factor variable
## even though it isn't one technically
lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")],
           na.rm = TRUE))
```

- Splitting on more than one level:

```
x <- rnorm(10)
f1 <- gl(2,5)
f2 <- gl(5,2)
interaction(f1,f2)
[1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
10 Levels: [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
str(split(x, list(f1,f2), drop = True))
```

  - This can be useful (for example) if you want to divide people based on multiple pieces of demographic information.

Debugging tools in R

- First step: diagnosing the problem.
- Indications that there is a problem:
  - `message`: a generic notification/diagnostic message produced by the `message` function; execution continues.
  - `warning`: indicates that something is wrong but not necessarily fatal; execution of the function continues; generated by the `warning` function.
  - `error`: a fatal problem has occurred; execution stop; produced by the `stop` function.
  - `condition`: a generic concept for indicating that something unexpected can occur; programmers can create their own conditions.
- Things to think about:
  - What was your input? How did you call the function?

  - What were you expecting? Output, messages, other results?
  - What did you get? How does this differ from your expectations?
  - Were your expectations correct in the first place?
  - Can you reproduce the problem exactly? (can be hard if random numbers are involved)

- Primary debugging tools in R:

  - `traceback`: prints out the function call stack after an error occurs; does nothing if there's no error.
  - `debug`: flags a function for "debug" mode which allows you to step through the execution of a function one line at a time.
  - `browser`: suspends the execution of a function wherever it is called and puts the function in debug mode.
  - `trace`: allows you to insert debugging code into a function at specific places.
  - `recover`: allows you to modify the error behavior so that you can browse the function call stack.
    * To use in the command line, enter `options(error = recover)`.

- These tools are all *interactive*, which is very useful!

The str function

- Displays the internal structure of an R object.
- It's a diagnostic function and an alternative to 'summary'.
- It is especially well suited to compactly display the abbreviated contents of lists (which may be nested).
- Roughly one line per basic object.
- Basic goal: answer the question "what's in this object?"

Simulation - generating random numbers

- Functions for probability distributions in R:

  - `rnorm`: generate random normal variates with a given mean and standard deviation.
  - `dnorm`: evaluate the normal probability density with a given mean and standard deviation at a point or vector of points.
  - `pnorm`: evaluate the cumulative distribution function for a normal distribution.
  - `rpois`: generate random Poisson variates with a given rate.

- Probability distribution functions usually have four functions associated with them. These functions are prefixed with the following letters:

  - `d` for density.
  - `r` for random number generation.
  - `p` for cumulative distribution.
  - `q` for quantile function.

- Setting the random number seed with `set.seed(x)` ensures reproducibility.

Simulation - simulating a linear model

- Example model: $y = \beta_0 + \beta_1 x + \epsilon$
- First, generate x and ϵ as random variables based on whatever distributions they have.
- Then add the β_0 and β_1 constants to get y.

Simulation - random sampling

- The `sample` function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distribution.
- Examples:

```
sample(1:10,4)
[1] 3 4 5 7

sample(letters,5)
[1] "q" "b" "e" "x" "p"

sample(1:10) ## permutation
[1] 4 7 10 6 9 2 8 3 1 5

sample(1:10, replace = TRUE) ## sample with replacement
[1] 2 9 7 8 2 8 5 9 7 8
```

R profiler

- Profiling is a systematic way to examine how much time is spent in different parts of a progrm.
- Useful when trying to optimize your code.
- Often code runs fine once, but what if you have to put it in a loop for 1000 iterations? Is it still fast enough?
- Profiling is better than guessing!
- Optimizing your code
  - Getting the biggest impact depends on knowing where the code spends most of its time.
  - This cannot be done without performance analysis or profiling.
- General principles of optimization:
  - Design first, then optimize.
  - "Premature optimization is the root of all evil."
  - Measure (collect data), don't guess.
- Using `system.time()`
  - Takes an arbitrary R expression as input and returns the amount of time to evaluate the expression.
  - If there's an error, it gives the time until the error occurred.
  - Returns an object of class `proc_time`
    - User time: time charged to the CPU for this expression.
    - Elapsed time: "wall clock" time.

    * Usually these two times are relatively close for straightforward computing tasks.
    * Elapsed time may be greater than user time if the CPU spends a lot of time waiting around.
    * Elapsed time may be smaller than the user time if your machine has multiple cores and is capable of using them.
        - Ex: multi-threaded BLAS libraries.
        - Ex: parallel processing via the `parallel` package.
  - `system.time()` is very useful, but not if you don't know where to start!

- Using the R profiler `Rprof()`
  - R must be compiled with profiler support (it usually is).
  - The `summaryRprof()` funciton summarizes the profiler output (otherwise it's not readable).
  - DO NOT use `system.time()` and `Rprof()` together or you will be sad!.
  - The profiler keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent in each function.
  - The default sampling interval is 0.02 seconds.
  - If your code runs very quickly, the profiler is not very useful (but then you probably don't need it anyway).

- Using `summaryRprof()`
  - This function tabulates the R profiler output and calculates how much time is spent in each function.
  - There are two methods for normalizing the data:
    * `by.total` divides the time spent in each function by the total run time.
    * `by.self` does the same but first subtracts out lower-level function calls.
  - It does not profile C or Fortran code!