Outline

Writing Functions In R

Necessity Really is a Mother

Paul E. Johnson[1][2]

[1]University of Kansas, Department of Political Science [2]Center for Research Methods and Data Analysis

2013

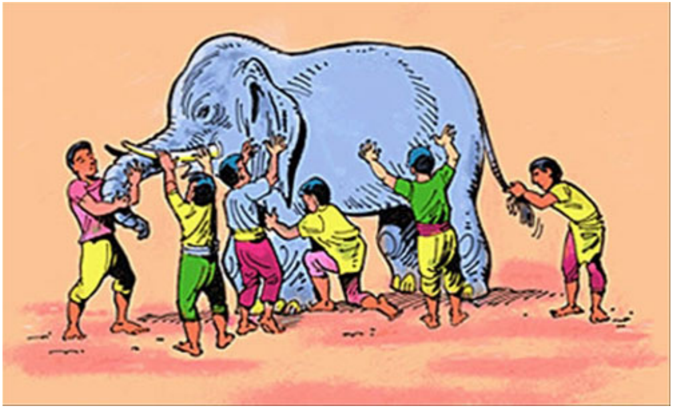
Outline

Outline

Did You Ever Write a Program?

- If Yes: R's different than that.
- If No: Its not like other things you've done.
- In either case, don't worry about it :)

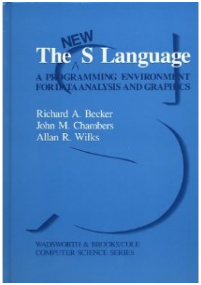
R is a little bit like an elephant



Its a tree trunk! Its a snake! Its a brush!

The R Language is like S, of course

- The S Language– John Chambers, et al. at Bell Labs, mid 1970s.
  See Richard Becker's "Brief History of S" about the AT&T years
- There have been 4 generations of the S language.
- Many packages now were written in S3, but S4 has been recommended for new packages for at least 5 years.
- A new framework known as "reference classes" is now being developed (was jokingly referred to as "R5" at one time)


NEW
The S Language
A PROGRAMMING ENVIRONMENT FOR DATA ANALYSIS AND GRAPHICS
Richard A. Becker
John M. Chambers
Allan R. Wilks
WADSWORTH & BROOKS/COLE COMPUTER SCIENCE SERIES

S3: *The New S Language* 1988

Is R a Branch from S?

- R can be seen as a competing implementation of S. Ross Ihaka and Robert Gentleman. 1996. "R: A language for data analysis and graphics." *Journal of Computational and Graphical Statistics*, 5(3):299-314.
- Open Source, Open Community, open repository CRAN

S pioneers now work to advance R.

S4: John Chambers,*Software for Data Analysis: Programming with R*, Springer, 2008

Outline

Overview: 3 reasons to write functions

- Preserve your sanity: isolate specific work.
  - Side benefit: preserve sanity of others who have to read your code
- Facilitate re-use of your ideas
- Co-operate with R functions like `lapply()` and `boot()` which REQUIRE functions from us.

Functions: Separate Calculations Meaningfully

- New programmers tempted to craft a giant sequence of 1000 commands
- Just Don't!

Problem	No other human can comprehend that mess
Solution	Write functions to calculate separate parts

- I don't feel comfortable with any function until I have a small "working example" to explore it (many available `http://pj.freefaculty.org/R`)

Re-use your work

- If you write a function, you can put it to use in many different contexts
- If you write a gigantic stream of 1000 commands, you can't.

Avoid for loops with lots of meat inside

Instead of this:

```
  for(i in 1:1000){
       1000s of lines here full of x[i], z[i], and so forth
       }
```

We want:

```
    fn1 <- function( arguments ) { ... }
    fn2 <- function( arguments ) { ... }
    for(i in 1: 1000){
       y <- fn1(x, ... )
       z <- fn2(y, ...)
       }
```

This is easier to read, understand, and more re-usable!

Personal confession

- My first attack at any problem is often a long string of commands that are not separable, not readable
- The revision process usually causes me to segregate code into separate pieces
- One hint that you need a function: constant cutting and pasting of code scraps from one place to another

When finished, I Wish Your R program would look like this

```
myfn1 <- function (argument1, argument2, ...){
   ## lines here using argument1, argument2
 }
myfn2 <- function (argument3, argument4){
   ## lines here
 }
## Try to perfect the above. Then use them
##
 a <- 7
 b <- c(4, 4, 4, 4, 2)
 great1 <- myfn1(a, b, parm3 = TRUE)
 great2 <- myfn2(b, great1)
```

How to Create a Function

- R allows us to create functions “on the fly”. This is the essential difference between a compiled language like C and an interpreted language like R. While an R session is running, we can add new capabilities to it.
- The artist Escher would like this one:

  The word `function` is a function that creates functions!

- A new function `somethingGood()` is defined by a stanza that begins like so:

```
somethingGood <- function(arguments){
```

- `somethingGood` is a name we choose

How to Create a Function ...

- The terms `arguments` and `parameters` are interchangeable. I often say `inputs`. In R, do not say “options”, that confuses people because R has a function called `options()` that governs the working session.
- `arguments` *may* be specified with default values, as in

```
somethingGood <- function(x1 = 0, x2 = NULL){
```

- After the squiggly brace, any valid R code can be used. We can even define functions inside the function!
- What happens in the function stays in the function. Things you create inside there do not escape the closure unless you really want them to.
- Return results: When when the function’s work is finished, a single object’s name is included on the last line.

How to Create a Function ...

```
somethingGood <- function(x1 = 0, x2 = NULL){
   ## suppose really interesting calculations create
       res, a result
   res
   }
```

- There are some little wrinkles about returns that will be discussed later. But, for now, I plant some seeds in your mind.
    - The return includes one object
    - The result will ordinarily print in the R terminal when the function runs, but we can prevent that by using the last line as

```
somethingGood <- function(x1 = 0, x2 = NULL){
## suppose really interesting calculations create \
    texttt{res}, a result
   invisible(res)
   }
```

How to Create a Function ...

- It is possible to exit sooner, to short-circuit the calculations before they have all run their course. That is what the `return()` function is for.

```
somethingGood <- function(x1 = 0, x2 = NULL){
## suppose you created res
 if (someLogicalCondition) return(invisible(res))
## otherwise, go on and revise res further.
 invisible(res)
 }
```

R Functions pass information "by value"

- The basic premise is that users should organize their information "here", in the current environment, and it is important that the function should not accidentally damage it.
- Thus, we send info "over there" to a function
- We get back a new something.
- The function **DOES NOT** change variables we give to the function
- The super assignment $<<-$ allows an exception to this, but R Core recommends we avoid it. Only experts should use it.

A simple example of a new function: doubleMe

```
doubleMe <- function(input = 0){
  newval <- 2 * input
}
```

The function's name is "doubleMe"
I choose a name for the incoming variable "input".
Other names would do (must start with a letter, etc.):

```
doubleMe <- function(x){
    out <- 2 * x
}
```

The last named thing is the one that comes back from the function. Note, explicit use of a return function is NOT REQUIRED. The last named thing comes back to us.

Key Elements of doubleMe

```
doubleMe <- function(input = 0){
  newval <- 2 * input
}
```

doubleMe	a name with which to access this function. Because of my background in "Objective C", I like this style of name. Don't put periods in function names unless you know about "classes" and are using them.
input	a name used INTERNALLY while making calculations
= 0	An *optional* default value.
newval	Last calculation is returned.

How to Call doubleMe

- What is 2 * 7?

```
(doubleMe(7))
```

```
[1] 14
```

- The caller may name the arguments explicitly:

```
(doubleMe(input = 8))
```

```
[1] 16
```

- Wonder why I use parentheses around everything? Its just a presentational trick. The default action is "print", and that's what happens when you put something in parentheses without a function name.
- The alternatives are:

```
print(doubleMe(input = 3))
```

How to Call doubleMe ...

```
[1] 6
```

- or

```
x <- doubleMe(input = 2)
x
```

```
[1] 4
```

Generally, I Prefer Clarity in the Call

- The "call" is the code statement that puts a function to use. The call includes the function's name and all arguments.
- This works

```
        doubleMe(10)
```

- But wouldn't you rather be clear?

```
doubleMe(input = 10)
```

- When there are many arguments, naming them often helps prevent accidental matching of input to arguments (R's positional matching can be fooled).

Function Calls

- But if you name the argument wrong, it breaks

```
> doubleMe(myInput = 7)
Error in doubleMe(myInput = 7) :
unused argument(s) (myInput = 7)
```

- What if you feed it something unsuitable?

```
> doubleMe(lm(rnorm(100) ~ rnorm(100)))
Error in 2 * input : non-numeric argument to binary
    operator
In addition: Warning messages:
1: In model.matrix.default(mt, mf, contrasts) :
  the response appeared on the right-hand side and was
      dropped
2: In model.matrix.default(mt, mf, contrasts) :
  problem with term 1 in model.matrix: no columns are
      assigned
```

Vectorization

This is not always true, but OFTEN:

- We get "free" "vectorization"

```
(doubleMe(c(1,2,3,4,5)))
```

```
[1]  2  4  6  8 10
```

- But it won't allow you to specify too many inputs:

```
> doubleMe(1,2,3,4,5)
  Error in doubleMe(1, 2, 3, 4, 5) :
      unused argument(s) (2, 3, 4, 5)
```

Vectorization: vector in $\Longrightarrow$ vector out

- Oops. I forgot the input

```
doubleMe()
```

Gives the default value.

print.function magic

- Oops. I forgot the parentheses

```
doubleMe
```

```
function(input = 0){
  newval <- 2 * input
}
```

- Similarly, type "lm" and hit return. Or "predict.glm". Don't add parentheses.
- When you type a function's name, R thinks you want to print that thing, and it invokes a "print method" for you, called `print.function()`. (That's `print` as applied to an object of class function.)
- Generally, `print.function()` will display the R internal functions in a "tidied up" format. Your functions–the ones you have created in your session–are generally not tidied up. That is discussed in the Rstyle vignette distributed with rockchalk.

predict.glm, for example

```
predict.glm
```

```
function (object, newdata = NULL, type = c("link", "response
    ",
    "terms"), se.fit = FALSE, dispersion = NULL, terms =
        NULL,
    na.action = na.pass, ...)
{
    type <- match.arg(type)
    na.act <- object$na.action
    object$na.action <- NULL
    if (!se.fit) {
        if (missing(newdata)) {
            pred <- switch(type, link = object$
                linear.predictors,
                response = object$fitted.values, terms =
                    predict.lm(object,
                  se.fit = se.fit, scale = 1, type = "terms"
                    ,
                  terms = terms))
            if (!is.null(na.act))
```

predict.glm, for example ...

```
                pred <- napredict(na.act, pred)
        }
        else {
            pred <- predict.lm(object, newdata, se.fit,
                scale = 1,
                type = ifelse(type == "link", "response",
                    type),
                terms = terms, na.action = na.action)
            switch(type, response = {
                pred <- family(object)$linkinv(pred)
            }, link = , terms = )
        }
    }
    else {
        if (inherits(object, "survreg"))
            dispersion <- 1
        if (is.null(dispersion) || dispersion == 0)
            dispersion <- summary(object, dispersion =
                dispersion)$dispersion
        residual.scale <- as.vector(sqrt(dispersion))
        pred <- predict.lm(object, newdata, se.fit, scale =
            residual.scale,
```

predict.glm, for example ...

```
                type = ifelse(type == "link", "response", type),
                terms = terms, na.action = na.action)
            fit <- pred$fit
            se.fit <- pred$se.fit
            switch(type, response = {
                se.fit <- se.fit * abs(family(object)$mu.eta(fit
                    ))
                fit <- family(object)$linkinv(fit)
            }, link = , terms = )
            if (missing(newdata) && !is.null(na.act)) {
                fit <- napredict(na.act, fit)
                se.fit <- napredict(na.act, se.fit)
            }
            pred <- list(fit = fit, se.fit = se.fit,
                residual.scale = residual.scale)
        }
        pred
}
<bytecode: 0x2bc3fb8>
<environment: namespace:stats>
```

Function Calls: Local versus Global

- The variables we create in “our” session are generally in the Global Environment.
- Local variables in functions.
  - Function arguments are local variables
  - Variables created inside are local variables
- Local variables cease to exist once R returns from the function
- After playing with `doubleMe()`, we note that the variable `input` does not exist in the current environment.

```
> ls()
[1] "doubleMe"
> input
Error: object 'input' not found
```

Check Point: write your own function

- Write a function “myGreatFunction” that takes a vector and returns the arithmetic average.
- Generate your own input data, x1, like so

```
set.seed(234234)
x1 <- rnorm(10000, m = 7, sd = 19)
```

- In `myGreatFunction()`, you can use any R functions you want, it is not necessary to re-create the `mean()` function (unless you really want to :))
- After you’ve written `myGreatFunction()`, use it:

```
x1mean <- myGreatFunction(x1)
x1mean
```

- Now stress test your function by changing x1

```
x1[c(13, 44, 99, 343, 555)] <- NA
myGreatFunction(x1)
```

Outline

Entropy can summarize diversity for a categorical variable

- Entropy in Physics means disorganization
- Sometimes called Shannon's Information Index
- Basic idea. List the possible outcomes and their probabilities
- The amount of diversity in a collection of observations depends on the equality of the proportions of cases observed within each type.

A Reasonable Person Would Agree . . .

- This distribution is “less diverse”

outcome name	t1	t2	t3	t4	t5
prob(outcome)	0.1	0.3	0.05	0.55	0.0

- than this distribution:

outcome name	t1	t2	t3	t4	t5
prob(outcome)	0.2	0.2	0.2	0.2	0.2

The Information Index

- For each type, calculate the following information (or can I say “diversity”?) value

$$-p_t * log_2(p_t) \tag{1}$$

- Note that if $p_t = 0$, the diversity value is 0
- If $p_t = 1$, then diversity is also 0
- Sum those values across the m categories

$$\sum_{t=1}^{m} -p_t * log_2(p_t) \tag{2}$$

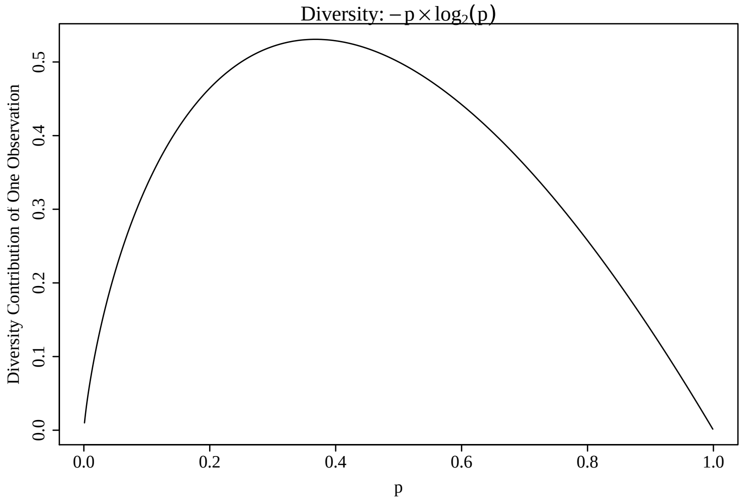
- Diversity is at a maximum when p_t are all equal

Calculate Diversity for One Type

```
divr <- function(p = 0){
  ifelse ( p > 0 & p < 1, -p * log2(p), 0)
}
```

Let's plot that

```
pseq <- seq(0.001, 0.999, length=999)
pseq.divr <- divr(pseq)
plot(pseq.divr ~ pseq, xlab = "p", ylab = "Diversity
    Contribution of One Observation",  main = expression(
    paste("Diversity: ", -p %*% log[2](p))), type = "l")
```


Diversity: $-p \times \log_2(p)$
Diversity Contribution of One Observation
0.0
0.1
0.2
0.3
0.4
0.5
0.0
0.2
0.4
0.6
0.8
1.0
p

Diversity Function

- Define an Entropy function that sums those values

```
entropy <- function(p){
     sum( divr(p) )
}
```

- Calculate some test cases

```
entropy( c(1/5, 1/5, 1/5, 1/5, 1/5) )
```

```
[1] 2.321928
```

```
entropy( c(3/5, 1/5, 1/5, 0/5, 0/5) )
```

```
[1] 1.370951
```

There's a Little Problem With This Approach

- Diversity is sensitive to the number of categories
  8 equally likely outcomes (rep(x,y): repeats x y times.)

```
entropy(rep(1/8, 8))
```

```
[1] 3
```

  14 equally likely outcomes

```
entropy(rep(1/14, 14))
```

```
[1] 3.807355
```

- Write it out for a 3 category case

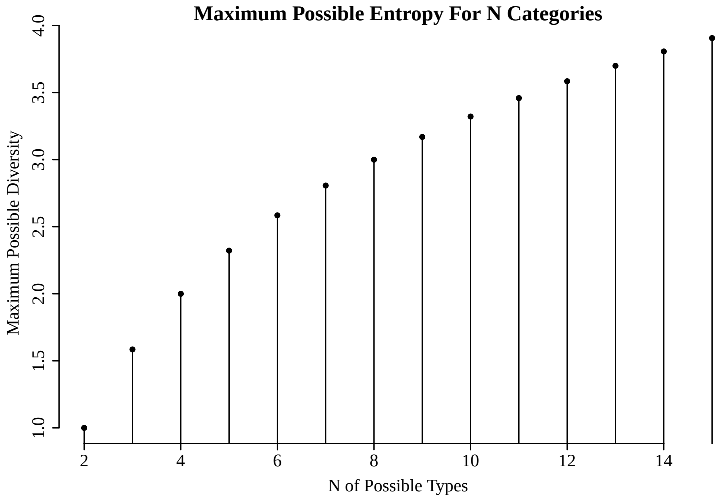
$$-\frac{1}{3}log_2(\frac{1}{3}) - \frac{1}{3}log_2(\frac{1}{3}) - \frac{1}{3}log_2(\frac{1}{3}) = -log_2(\frac{1}{3}) \qquad (3)$$

- The highest possible diversity with 3 types is $-log_2(\frac{1}{3})$
- The highest possible diversity for N types is $-log_2(\frac{1}{N})$

We Might As Well Plot That

```
maximumEntropy <- function(N) - log2(1/N)
Nmax <- 15
M <- 2:Nmax
plot(M, maximumEntropy(M), xlab = "N of Possible Types",
     ylab = "Maximum Possible Diversity", main = "Maximum
     Possible Entropy For N Categories", type = "h", axes =
     FALSE)
axis(1)
axis(2)
points(M, maximumEntropy(M), pch=19)
```

Maximum Entropy as a Function of the Number of Types



Final Result: Normed Entropy as a Diversity Summary

```
normedEntropy <- function(x) entropy(x)/ maximumEntropy(
    length(x))
```

Compare some cases with 4 possible outcomes

```
normedEntropy(c(1/4,1/4,1/4,1/4))
```

```
[1] 1
```

```
normedEntropy(c(1/2, 1/2, 0, 0))
```

```
[1] 0.5
```

```
normedEntropy(c(1, 0, 0, 0))
```

```
[1] 0
```

How about 7 types of outcomes:

```
normedEntropy(rep(1/7, 7))
```

```
[1] 1
```

```
normedEntropy((1:7)/(sum(1:7)))
```

```
[1] 0.9297027
```

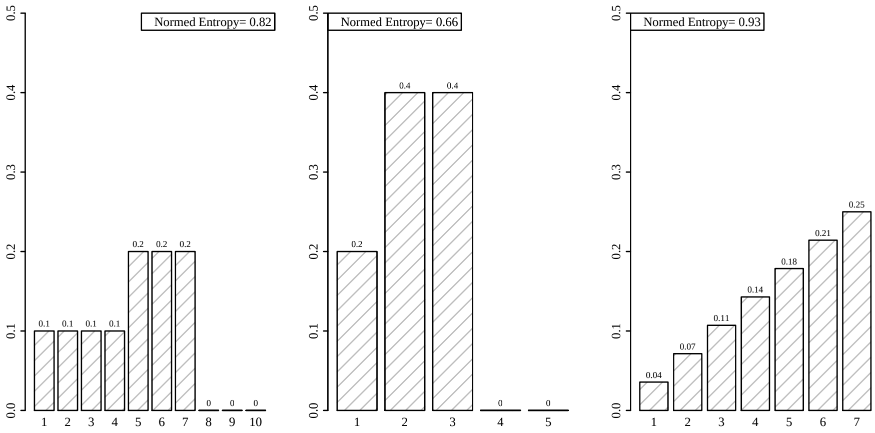
```
normedEntropy(c(2/7, 2/7, 3/7, 0, 0, 0, 0))
```

```
[1] 0.5544923
```

```
normedEntropy(c(5/7, 2/7, 0, 0, 0, 0, 0))
```

```
[1] 0.3074497
```

Compare 3 test cases



Subjectively, I wrestle with the question of whether comparison across variables with different M is meaningful.

Entropy is reported in summarize() and summarizeFactors() in the rockchalk package

Manufacture a variable to re-produce testcase3

```
round(testcase3,2)
```

```
[1] 0.04 0.07 0.11 0.14 0.18 0.21 0.25
```

```
library(rockchalk)
testcase3v <- factor(c(1,2,2,3,3,3, 4,4,4,4, 5,5,5,5,5,
    6,6,6,6,6,6, 7,7,7,7,7,7,7 ))
round((table(testcase3v)/length(testcase3v)), 2)
```

```
testcase3v
   1    2    3    4    5    6    7
0.04 0.07 0.11 0.14 0.18 0.21 0.25
```

```
dat <- data.frame(testcase3v)
summarizeFactors(dat)
```

Entropy is reported in `summarize()` and `summarizeFactors()` in the rockchalk package ...

```
           testcase3v
7             :  7.0000
6             :  6.0000
5             :  5.0000
4             :  4.0000
(All Others)  :  6.0000
NA's          :  0.0000
entropy       :  2.6100
normedEntropy :  0.9297
N             :28.0000
```

Outline

Function Anatomy

Arguments: The Arguments of a function

```
    someWork <- function(what1, what2, what3)
```

what1, what2, and what3 become “local variables” inside the function.

- named arguments must begin with letters, may include numbers, or “_” or “.”, but no “’ or “,” or “*” or ”-”

Arguments: named and unnamed R variables

```
someWork <- function(what1, what2, what3, ...)
```

- In R, everything is an “object”. what1, what2, what3 can be *ANYTHING!*
- That is a blessing and a curse
  - Blessing: Flexibility! Let an argument be a vector, matrix, list, whatever. The function declaration does not care.
  - Curse: Difficulty managing the infinite array of possible ways users might break your functions.
- It is your choice, whether your function should receive
  - 2 integers (x, y), or
  - one vector of 2 integers (x)
  - For examples of this, look at the arguments of plot.default, arrows, segments, lines, and matplot.

Peculiar spellings like . . .

Functions may have an argument ". . ."

- It seems funny, but "..." is actually a word made up of *three legal* characters
- If the caller includes any named arguments that are not `what1`, `what2`, or `what3`, then the R runtime will put them in a list and give them to "..."
- The "..." argument requires special effort.

Function Arguments: R is VERY Lenient

- R is lenient. Perhaps too lenient!
  - Arguments are not type-defined. In

    ```
    myF <- function(x1, x2){
    ```

    x1 could be an integer vector, a character string, a regression model, or anything
  - Function writer may declare default values, but need not. These are acceptable definitions

    ```
    someWork <- function(what1, what2, what3, what4,
        what5)

    someWork <- function(what1 = 0, what2 = NULL, what3
         = c(1,2,3), what4 = 3 * what1, what5)

    someWork <- function(what1 = 0, what2, what3, what4
         = 3 * what1, what5)
    ```

R is Lenient toward Users as Well

- R is lenient on format of function calls

```
someWork(1)
someWork(what=1, someObject)
someWork(what5 = fred, what4 = jim, what3 = joe)
```

  - Not all parameters must be provided
  - Partial argument matching

R is Lenient About Undefined Variables

This is a convenience and a curse

Variables inside functions might not be resolved the way you expect.

- If a variable is used, but not defined inside the function, R will "look outward" for it
- This is "lexical scope" in action. The runtime engine searches through a sequence of "frames", ending up at the user's workspace (which is the Global Environment).

Example of the Undefined Variable Problem

- Suppose “dat” exists in your workspace.
- Here is a function that will find “dat”, even if that’s not what you intend.

  ```
  myFn <- function(x, y, z){
      dat1 <- 2 * x + 3 * y
      res <- sqrt(dat)
  }
  ```

  Note my typographical error (dat where dat1 should be). The function should crash, but it will not.
- R will find the wrong “dat” and the result we get will be unhelpful
- In my experience, this is the single most important cause of hard-to-find flaws in user code.

Inside the function, Check Arguments

- Check and Re-format
  - “argument checking”: diagnose what arguments the user provided
  - Figure out if they are “correct” for what the function requires.
- You choose how sympathetic you want to be toward users. Do you want to accept incomplete input and re-format it? (In rockchalk/R/genCorrelatedData.R, find may “re-formatter” functions like `makeVec()`, `vech2mat()`).

Arguments: When to Use Defaults?

- I don't know, but ...
- As an R beginner, I took a very conservative strategy of putting defaults on everything!

```
function(what1 = NULL, what2 = NULL, what3 = 3, what4 =
     ``a'')
```

- That way, if a user forgets to provide "what3", then the system will not go looking for it.
- If the defaults usually work, this is concise, easy to read.
- Most functions in R base code don't set defaults for all, or even most, variables.
- Instead, we manage arguments with more delicacy. Insisting on NULL defaults is "throwing the baby out with the bath water."

Functions that Help while Checking Arguments

missing()

- Inside a function, `missing()` is a way to ask if the user supplied a value for an argument.

```
doSomething <- function(what1, what2, what3, what4){
   if (missing(what1)) stop(``you forgot to specify
       what1'')
}
```

- I've found this conservative approach to be an error-preventer. If x is not provided, or if the user gave us a NULL, we better adapt!

```
if (missing(x) || is.null(x)) ## do something
```

Functions that Help while Checking Arguments ...

Type Checks

There are many "is." functions. Ask if x is a certain type of thing:

is.vector()	TRUE or FALSE: are you a vector?
is.list()	TRUE or FALSE: are you a list?
is.numeric()	TRUE or FALSE: are you numeric?
	You get the general idea, I hope

Functions that Help while Checking Arguments ...

Look at all of these is. functions!

```
is.array            is.loaded                   is.object
is.atomic           is.logical                  is.ordered
is.call             is.matrix                   is.package_version
is.character        is.mts                      is.pairlist
is.complex          is.na                       is.primitive
is.data.frame       is.na<-                     is.qr
is.double           is.na.data.frame            is.R
is.element          is.na<-.default             is.raster
is.empty.model      is.na<-.factor              is.raw
is.environment      is.name                     is.recursive
is.expression       is.nan                      is.relistable
is.factor           is.na.numeric_version       is.single
is.finite           is.na.POSIXlt               is.stepfun
is.function         is.null                     is.symbol
is.infinite         is.numeric                  is.table
is.integer          is.numeric.Date             is.ts
is.language         is.numeric.difftime         is.tskernel
is.leaf             is.numeric.POSIXt           is.unsorted
is.list             is.numeric_version          is.vector
```

Functions that Help while Checking Arguments ...

Size Checks

length()	
nrow(), ncol()	number of rows (columns) from a matrix
NROW(), NCOL()	Works if input is matrix, but will treat a vector as a one column matrix.
dim()	Returns 2 dimensional vector, works with matrices and arrays

Stop, Warn

stop(), stopifnot()	Ways to abend the function
warning	Will return, but throws a message onto the R warning log. User can run warnings() to see messages.

The Return Has To Be Singular

- When you use a function, it is necessary to "catch" the output with a single object name, as in

```
newthing <- doubleMe(32)
newthing
```

```
[1] 64
```

```
is.numeric(newthing)
```

```
[1] TRUE
```

```
is.vector(newthing)
```

```
[1] TRUE
```

- We expect "doubleMe(32)" should return 64, and it does.

The Return Has To Be Singular ...

- The “vectorization for free” gives us a hint of what is to follow.

```
newthing <- doubleMe(c(1,2,3,4,5))
newthing
```

```
[1]  2  4  6  8 10
```

```
is.numeric(newthing)
```

```
[1] TRUE
```

```
is.vector(newthing)
```

```
[1] TRUE
```

- R allows us to return one “thing”, but “thing” can be a rich, informative thing!

Generalization. Return a list

- A list may include numbers, characters, vectors ,etc
- or data frames or other lists
- Read code for function "lm"
- Almost ALL interesting R functions return a list of elements.

Check Point: revise your function

- Create “myGreatFunction2” by revising “myGreatFunction”. Make it return a vector of 3 values: the maximum, the minimum, and the median.
- Generate your own input data, x1, like so

```
x1 <- rnorm(10000, m=7, sd=19)
```

- After you’ve written myGreatFunction, use it:

```
myGreatFunction(x1)
```

- Now stress test your function by changing x1

```
x1[c(13, 44, 99, 343, 555)] <- NA
myGreatFunction(x1)
```

Check Point: Run this example function

Admittedly, this is a dumb example, but . . .

- This function returns a regression object

```
aRegMod <- function(xin1, xin2, yout){ lm(yout ~ xin1
    + xin2) }
```

- Generate some data, run aRegMod()

```
dat <- rockchalk::genCorrelatedData(N = 100, rho = 0.3
    , beta = c(1, 1.0, -1.1, 0.0))
m1 <- with(dat, aRegMod(x1, x2, y))
```

- m1 is a “single object”
- Run “class(m1)”, “attributes(m1)”, “summary(m1)”, ‘str(m1)”,

Outline

The Unofficial Official R Style Guide

- This is discussed in rockchalk vignette Rstyle
- There is not much "official style guidance" from the R Core Team
- Don't mistake that as permission to write however you want.
- There ARE very widely accepted standards for the way that code should look

Inductive Style Guide

- To see that R really does have an implicitly stated Style, inspect the R source code.
- The code follows a uniform pattern of indentation, the use of white space, and so forth.
- Recall that `print.function()` will be called if you type the name of a function without parentheses. That is a tidied up view of R's internal structural represenation of a function.
- . This time, lets look at "lm" inside R:

```
lm
```

Inductive Style Guide ...

```
function (formula, data, subset, weights, na.action, method
    = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
        singular.ok = TRUE,
    contrasts = NULL, offset, ...)
{
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    mf <- match.call(expand.dots = FALSE)
    m <- match(c("formula", "data", "subset", "weights", "
        na.action",
        "offset"), names(mf), 0L)
    mf <- mf[c(1L, m)]
    mf$drop.unused.levels <- TRUE
    mf[[1L]] <- as.name("model.frame")
    mf <- eval(mf, parent.frame())
    if (method == "model.frame")
        return(mf)
    else if (method != "qr")
```

Inductive Style Guide ...

```
        warning(gettextf("method = '%s' is not supported.
            Using 'qr'",
            method), domain = NA)
    mt <- attr(mf, "terms")
    y <- model.response(mf, "numeric")
    w <- as.vector(model.weights(mf))
    if (!is.null(w) && !is.numeric(w))
        stop("'weights' must be a numeric vector")
    offset <- as.vector(model.offset(mf))
    if (!is.null(offset)) {
        if (length(offset) != NROW(y))
            stop(gettextf("number of offsets is %d, should
                equal %d (number of observations)",
                length(offset), NROW(y)), domain = NA)
    }
    if (is.empty.model(mt)) {
        x <- NULL
        z <- list(coefficients = if (is.matrix(y)) matrix(,
            0,
            3) else numeric(), residuals = y, fitted.values
                = 0 *
```

Inductive Style Guide ...

```
                  y, weights = w, rank = 0L, df.residual = if (!
                        is.null(w)) sum(w !=
                  0) else if (is.matrix(y)) nrow(y) else length(y)
                        )
            if (!is.null(offset)) {
                  z$fitted.values <- offset
                  z$residuals <- y - offset
            }
      }
      else {
            x <- model.matrix(mt, mf, contrasts)
            z <- if (is.null(w))
                  lm.fit(x, y, offset = offset, singular.ok =
                        singular.ok,
                        ...)
            else lm.wfit(x, y, w, offset = offset, singular.ok =
                  singular.ok,
                  ...)
      }
      class(z) <- c(if (is.matrix(y)) "mlm", "lm")
      z$na.action <- attr(mf, "na.action")
      z$offset <- offset
```

Inductive Style Guide ...

```
    z$contrasts <- attr(x, "contrasts")
    z$xlevels <- .getXlevels(mt, mf)
    z$call <- cl
    z$terms <- mt
    if (model)
        z$model <- mf
    if (ret.x)
        z$x <- x
    if (ret.y)
        z$y <- y
    if (!qr)
        z$qr <- NULL
    z
}
<bytecode: 0x2a512f8>
<environment: namespace:stats>
```

Why Neatness Counts

- You can write messy code, but you can't make anybody read it.
- Finding bugs in a giant undifferentiated mass of commands is difficult
- Chance of error rises as clarity decreases
- If you want people to help you, or use your code, you should write neatly!

Style Fundamentals

- WHITE SPACE:
  - indentation. R Core Team recommends 4 spaces
  - one space around operators like <- = *
- "$< -$" should be used for assignments. "$=$" was used by mistake so often by novices that the R interpreter was re-written to allow $=$. However, it may still fail in some cases.
- Use helpful variable names
- Separate calculations into functions. Sage advice from one of my programming mentors:

  Don't allow a calculation to grow longer than the space on one screen. Break it down into smaller, well defined pieces.

Be Careful about line endings

- Unlike C (or other languages), R does not require an “end of line character” like “;”.
- That’s convenient, but sometimes code can “fool” R into believing that a command is finished.
- From the help page for “if”

  > *Note that it is a common mistake to forget to put braces (‘{ .. }’) around your statements, e.g., after ‘if(..)’ or ‘for(....)’. In particular, you should not have a newline between ‘}’ and ‘else’ to avoid a syntax error in entering a ‘if ... else’ construct at the keyboard or via ‘source’. For that reason, one (somewhat extreme) attitude of defensive programming is to always use braces, e.g., for ‘if’ clauses.*

Be Careful about line endings ...

- The ■ else " Policy. I strongly recommend this format:

  ```
  if (a-logical-condition) {
    ## if TRUE, do this
    } else {
      ## if FALSE, the other thing
      }
  ```

  to close the previous if and begin else on same line.

- "if-else" is very troublesome. If R thinks the "if" is finished, it may not notice the else.

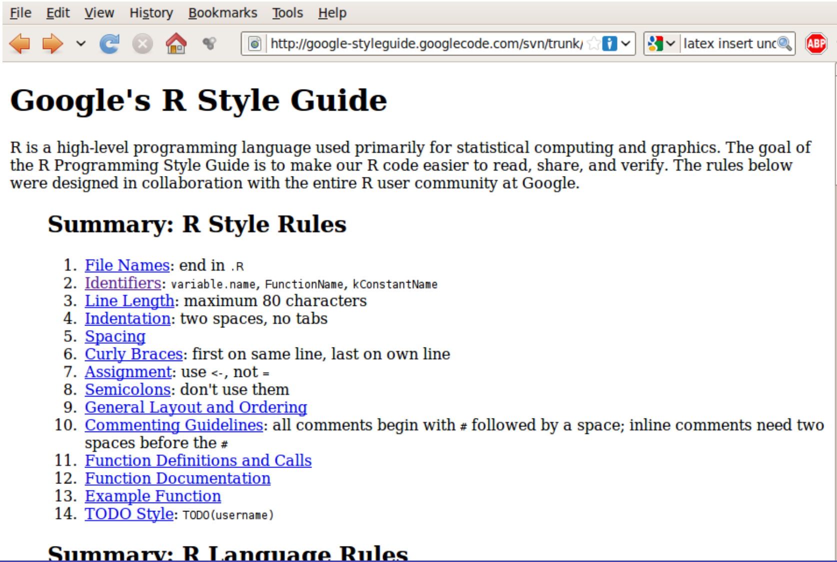
  ```
      if (x > 7) y <- 3
      else y <- 2
  ```

  Causes "Error: unexpected 'else' in "else"

- When running code line-by-line, the "naked else" always causes an error.

Google Doc on R Coding is Just “somebody’s” Opinion

- The Easily Googled Google R Standards


File Edit View History Bookmarks Tools Help
http://google-styleguide.googlecode.com/svn/trunk/
latex insert unc

Google's R Style Guide

R is a high-level programming language used primarily for statistical computing and graphics. The goal of the R Programming Style Guide is to make our R code easier to read, share, and verify. The rules below were designed in collaboration with the entire R user community at Google.

Summary: R Style Rules

1. File Names: end in `.R`
2. Identifiers: `variable.name`, `FunctionName`, `kConstantName`
3. Line Length: maximum 80 characters
4. Indentation: two spaces, no tabs
5. Spacing
6. Curly Braces: first on same line, last on own line
7. Assignment: use `<-`, not `=`
8. Semicolons: don't use them
9. General Layout and Ordering
10. Commenting Guidelines: all comments begin with `#` followed by a space; inline comments need two spaces before the `#`
11. Function Definitions and Calls
12. Function Documentation
13. Example Function
14. TODO Style: `TODO(username)`

Summary: R Language Rules

How To Name Functions

- Don't use names for functions that are already in widespread use, like lm, seq, rep, etc.
- I like Objective C style variable and function names that smash words together, as in `myRegression` `myCode`
- R uses periods in function names to represent "object orientation" or "subclassing", thus I avoid periods for simple punctuation.
  Ex: doSomething() is better than do.something
- Underscores are now allowed in function names.
  Ex: do_something() would be OK

How To Name Variables

- Use clear names always
- Use short names where possible
- Never use names of common functions for variables
- Never name a variable T or F (doing so tramples R symbols)
- “Name by suffix” strategy I’m using now:

```
m1 <- lm( y ~ x, data=dat)
m1sum <- summary(m1)
m1vif <- vif(m1)
m1inf <- influence.measures(m1)
```

Expect Some Variations in My Code

- I don't mind adding squiggly braces, even if not required
  `if (whatever == TRUE) {x <- y}`
- Sometimes I will use 3 lines where one would do

```
if (whatever == TRUE){
             x <- y
}
```

- When in doubt, I like to explicitly name arguments, but not always.
- I sometimes forget to write TRUE and FALSE when T and F will suffice
- Here's why that's a big deal. Suppose a some user mistakenly redefines T or F:

```
T <- 7
F <- myTerrificFunction
```

then my functions that use "T" and "F" will die.

Outline

What is this Section?

- Tips from the school of hard knocks
- Criticisms of R code I've found or written

Functions replace “cut and paste” editing

If you find yourself using “copy and paste” to repeat stanzas with slight variations, **you are almost certainly doing the wrong thing**.

- Re-conceptualize, write a function that does the right thing
- Use the function over and over

Why is this important: AVOIDING mistakes due to editing mistakes

We Learn By Criticizing

I keep a folder of R code that troubles me.
This example is a more-or-less literal translation from SAS into R that does not use R's special features.

```
#---------------SPECIFICATIONS------------------------#
iter=1000                         #how many iterations per
    condition
set.seed(7913025)                         # set random seed
#--------------END SPECIFICATIONS---------------------#
#n per cluster sample size
for (perclust in c(100)) {
#number of clusters - later for MLM application
  for (nclust in c(1) ){
#common correlation
    for (setcorr in c(1:8) ){
      if(setcorr==1){
        corr.10 <-1
        corr.20 <-0
        corr.30 <-0
        corr.40 <-0
```

We Learn By Criticizing ...

```
      corr.50 <-0
      corr.60 <-0
      corr.70 <-0
      corr.80 <-0
      }
    if(setcorr==2){
      corr.10 <-1
      corr.20 <-1
      corr.30 <-0
      corr.40 <-0
      corr.50 <-0
      corr.60 <-0
      corr.70 <-0
      corr.80 <-0
      }
    if(setcorr==3){
      corr.10 <-1
      corr.20 <-1
      corr.30 <-1
      corr.40 <-0
      corr.50 <-0
      corr.60 <-0
```

We Learn By Criticizing ...

```
    corr.70 <-0
    corr.80 <-0
    }
  if(setcorr==4){
    corr.10 <-1
    corr.20 <-1
    corr.30 <-1
    corr.40 <-1
    corr.50 <-0
    corr.60 <-0
    corr.70 <-0
    corr.80 <-0
    }
  if(setcorr==5){
    corr.10 <-1
    corr.20 <-1
    corr.30 <-1
    corr.40 <-1
    corr.50 <-1
    corr.60 <-0
    corr.70 <-0
    corr.80 <-0
```

We Learn By Criticizing ...

```
      }
    if(setcorr==6){
      corr.10 <-1
      corr.20 <-1
      corr.30 <-1
      corr.40 <-1
      corr.50 <-1
      corr.60 <-1
      corr.70 <-0
      corr.80 <-0
      }
    if(setcorr==7){
      corr.10 <-1
      corr.20 <-1
      corr.30 <-1
      corr.40 <-1
      corr.50 <-1
      corr.60 <-1
      corr.70 <-1
      corr.80 <-0
      }
    if(setcorr==8){
```

We Learn By Criticizing ...

```
        corr.10 <-1
        corr.20 <-1
        corr.30 <-1
        corr.40 <-1
        corr.50 <-1
        corr.60 <-1
        corr.70 <-1
        corr.80 <-1
    }
```

That project defined several variables in that way, consuming 100s of lines.

Quick Exercise

How would you convert that into a vector in R, without writing 300 lines.
Hint. You want to end up with a vector `setcorr` with 0 elements, all either 0 or 1.
Requirement: You need some easy, flexible way to adjust the number of 0's and 1's

Another “noisy code” example

Note in the following

1. the author does not declare functions
   Rather, treats comments ahead of blocks of code as if they were function declarations.
2. repeated use of cat() with same file argument is an example of cut-and-paste coding.

```
#########################################
#       WRITE GENERATION CODE           #
#########################################

pathGEN <- paste(dirroot,perclust,nclust,setcorr,sep="\\")
gen <- paste(pathGEN,"generatecorrdata.inp",sep="\\")
testdata1 <- paste(pathGEN,"data1.dat",sep="\\")

cat('MONTECARLO: \n', file=gen)
cat('NAMES ARE x y q1-q8; \n', file=gen, append=T)
cat('NOBSERVATIONS = 1000 ; \n', file=gen, append=T)
```

Another “noisy code” example ...

```
cat(' NREPS = ',iter,'; \n', file=gen, append=T)
cat(' SEED = ',round(runif(1)*10000000), '; \n', file=gen,
    append=T)
cat(' REPSAVE=ALL; \n', file=gen, append=T)
cat(' SAVE=\n', pathGEN,'\\data*.dat; \n', file=gen, append=
    T, sep="")
cat('MODEL POPULATION: \n', file=gen, append=T)
cat('[q1-q8*0 x*0 y*0]; \n', file=gen, append=T)
cat('q1-q8*1; x*1; y*1;  \n', file=gen, append=T)
cat('q1-q8 with q1-q8*.50;  \n', file=gen, append=T)
cat('x with y*.50; \n', file=gen, append=T)
cat('x with q1-q8*.50; \n', file=gen, append=T)
cat('y with q1-q8*', .0+(corr.10*(.10+corr.20*.10+corr.30*
    .10+corr.40*+corr.40*.10+corr.50*.10+corr.60*.10+corr.70
    *.10+corr.80*.10)), '; \n', file=gen, append=T, sep="")

if(file.exists(testdata1)){
} else {
shell (paste("mplus.exe",gen, paste(pathGEN,"save1.out", sep
    ="\\"), sep=" "))
        }
```

Another “noisy code” example ...

```
#
    ----------------------------------------------------------------------
    #

#########################################
#          GENERATE SAS CODE            #
#########################################
pathSAS <- paste(dirroot,perclust,nclust,setcorr,setpattern,
    percentmiss,aux,auxnumber,sep="\\")
pathSASdata <- paste(dirroot,perclust,nclust,setcorr,
    setpattern,percentmiss,sep="\\")
smcarmiss <- paste(pathSAS,"modifydata.sas",sep="\\")
testdata2 <- paste(pathSASdata,"data1.dat",sep="\\")

#-------------------------------------------------------#
#------------ import SIM data into SAS ----------------#
#-------------------------------------------------------#

if(file.exists(testdata2)){
} else {

cat('proc printto \n', file=smcarmiss, append=T)
```

Another “noisy code” example ...

```
cat('log="R:\\users\\username\\data\\simLOG2\\LOGLOG.log" \n
    ', file=smcarmiss, append=T)
cat('print="R:\\users\\username\\data\\simLOG2\\LSTLST.lst"
    \n', file=smcarmiss, append=T)
cat('new; \n', file=smcarmiss, append=T)
cat('run; \n', file=smcarmiss, append=T)
cat(' \n', file=smcarmiss, append=T)

cat('%macro importMPLUS; \n', file=smcarmiss, append=T)
cat('%do i=1 %to ', file=smcarmiss, append=T)
cat(paste(iter) , file=smcarmiss, append=T)
cat('; \n', file=smcarmiss, append=T)
cat('data work.data&i; \n', file=smcarmiss, append=T)
cat('infile ', file=smcarmiss, append=T)
cat('"', file=smcarmiss, append=T)
cat(paste(pathGEN,"data&i..dat",sep="\\") , file=smcarmiss,
    append=T)
cat('" ; \n', file=smcarmiss, append=T)
cat('INPUT x y q1 q2 q3 q4 q5 q6 q7 q8; /*<---insert
    variables here*/ \n', file=smcarmiss, append=T)
cat('RUN; \n', file=smcarmiss, append=T)
cat('%end; \n', file=smcarmiss, append=T)
```

Another "noisy code" example ...

```
cat('%mend; \n', file=smcarmiss, append=T)
cat('%importMPLUS \n', file=smcarmiss, append=T)
cat(' \n', file=smcarmiss, append=T)

########################################
#     Draw random sample of size N        #
########################################
cat('%macro samplesize; \n', file=smcarmiss, append=T)
cat('%do i=1 %to ', file=smcarmiss, append=T)
cat(paste(iter) , file=smcarmiss, append=T)
cat('; \n', file=smcarmiss, append=T)
cat('Proc surveyselect data=work.data&i out=work.sampledata&
    i method=SRS \n', file=smcarmiss, append=T)
if(perclust==50) {
cat('sampsize=50 \n', file=smcarmiss, append=T)
}
else if(perclust==75) {
cat('sampsize=75 \n', file=smcarmiss, append=T)
}
else if(perclust==100) {
cat('sampsize=100 \n', file=smcarmiss, append=T)
}
```

Another “noisy code” example ...

```
else if(perclust==200) {
cat('sampsize=200 \n', file=smcarmiss, append=T)
}
else if(perclust==400) {
cat('sampsize=400 \n', file=smcarmiss, append=T)
}
else if(perclust==800) {
cat('sampsize=800 \n', file=smcarmiss, append=T)
}
else if(perclust==1000) {
cat('sampsize=1000 \n', file=smcarmiss, append=T)
}
cat('SEED = ',round(runif(1)*10000000), '; \n', file=
    smcarmiss, append=T)
cat('RUN; \n', file=smcarmiss, append=T)
cat('%end; \n', file=smcarmiss, append=T)
cat('%mend; \n', file=smcarmiss, append=T)
cat('%samplesize \n', file=smcarmiss, append=T)
cat(' \n', file=smcarmiss, append=T)
```

This could be much better

- Weird indentation
- Use "/", not "backslash", even on Windows
- Use vectors
- Prolific copying and pasting of "cat" lines.
- Avoid system except where truly necessary. R has OS neutral functions to create directorys and such.

I found the prototype for that code in a previous project

```
gen <- paste(path,"generate.inp",sep="\\")

cat('MONTECARLO: \n', file=gen)
cat('NAMES ARE y1-y6; \n', file=gen, append=T)
cat(' NOBSERVATIONS = ',perclust*nclust, '; \n', file=gen,
    append=T)
if (perclust != 7.5) {
cat(' NCSIZES = 1; \n', file=gen, append=T)
cat(' CSIZES = ',nclust, '(', perclust, '); \n', file=gen,
    append=T)
}
if (perclust==7.5) {
cat(' NCSIZES = 2; \n', file=gen, append=T)
cat(' CSIZES = ',nclust/2, ' (7) ',nclust/2, ' (8); \n',
    file=gen, append=T)
}

#user-specified iterations
cat(' NREPS = ',iter,'; \n', file=gen, append=T)
cat(' SEED = 791305; \n', file=gen, append=T)
```

I found the prototype for that code in a previous project ...

```
cat(' REPSAVE=ALL; \n', file=gen, append=T)
cat(' SAVE=\n', path,'\\data*.dat; \n', file=gen, append=T,
    sep="")
cat(' ANALYSIS: TYPE = TWOLEVEL; \n', file=gen, append=T)

cat('MODEL POPULATION: \n', file=gen, append=T)

cat('%WITHIN% \n', file=gen, append=T)
# baseline loadings are all .3, add .4 if 'within' = 1, but
    change based on mod/strong
cat('FW BY y1-y2*', .3+(within*(.4+strong*.1)), ' \n', file=
    gen, append=T, sep="")
cat('y3-y4*', .3+(within*(.4-mod*.3)), ' \n', file=gen,
    append=T, sep="")
cat('y5-y6*', .3+(within*(.4-strong*.1)), '; \n', file=gen,
    append=T, sep="")
cat('FW@1;  \n', file=gen, append=T)
# residuals are just 1-loading^2
cat('y1-y2*', 1-(.3+(within*(.4+strong*.1)))^2, '; \n', file
    =gen, append=T, sep="")
cat('y3-y4*', 1-(.3+(within*(.4-mod*.3)))^2, '; \n', file=
    gen, append=T, sep="")
```

I found the prototype for that code in a previous project ...

```
cat('y5−y6*', 1−(.3+(within*(.4−strong*.1)))^2, '; \n', file
    =gen, append=T, sep="")
cat(' \n',file=gen, append=T, sep="")
cat('%BETWEEN% \n', file=gen, append=T)

# the ICC bit multiplies by .053 when ICC is low, by 1 when
    ICC is high
cat('FB BY y1−y2*', sqrt((.3+(between*(.4+strong*.1)))^2*
    (1+(icc*(.053−1)))), ' \n', file=gen, append=T, sep="")
cat('y3−y4*', sqrt((.3+(between*(.4−mod*.3)))^2 *(1+(icc*(
    .053−1)))), ' \n',file=gen, append=T, sep="")
cat('y5−y6*', sqrt((.3+(between*(.4−strong*.1)))^2 *(1+(icc*
    (.053−1)))), '; \n', file=gen, append=T, sep="")
cat('FB@1;  \n', file=gen, append=T)

#residuals are total variance (1 or .053, depending on  ICC)
     loading^2
cat('y1−y2*', 1+(icc*(.053−1))−sqrt((.3+(between*(.4+strong*
    .1)))^2*(1+(icc*(.053−1))))^2, '; \n', file=gen, append=
    T, sep="")
```

I found the prototype for that code in a previous project ...

```
cat('y3-y4*', 1+(icc*(.053-1))-sqrt((.3+(between*(.4-mod*.3)
    ))^2 *(1+(icc*(.053-1))))^2, '; \n', file=gen, append=T,
     sep="")
cat('y5-y6*', 1+(icc*(.053-1))-sqrt((.3+(between*(.4-strong*
    .1)))^2 *(1+(icc*(.053-1))))^2, '; \n', file=gen, append
    =T, sep="")
cat(' \n',file=gen, append=T, sep="")

#run the above syntax using Mplus

#shell (paste("cd", mplus, sep=" "))
shell (paste("mplus.exe",gen, paste(path,"save.out", sep="\\
    "), sep=" "))
```

Here was my Suggestion

Create separate functions to do separate parts of the work. Avoid so much "cut and paste" coding. The cat function can include MANY separate quoted strings or values, there is no reason to write separate cat statements for each line.
Here was my suggestion for part of the revision

```
##Create one MPlus Input file corresponding to following
    parameters.
createInpFile <- function(path="apath", gen="afilename.inp"
    , perclust=2, nclust=100, iter=1000, mod=1, strong=1,
    between=1, within=1){
    cat("MONTECARLO:
      NAMES ARE y1-y6;
      NOBSERVATIONS = ", perclust*nclust, "; \n",
        ifelse(perclust != 7.5 ,
                  paste("NCSIZES = 1; \n      CSIZES =", nclust,
                        "(", perclust, ");\n"),
                  paste("  NCSIZES = 2; \n    CSIZES = ",
                      nclust/2, " (7) ", nclust/2 , " (8); \n"
                        ) ), file=gen,    append=T,
```

Here was my Suggestion ...

```
        sep="")

  ##user-specified iterations
    cat( "NREPS = ", iter, ";
      SEED = 791305;
      REPSAVE=ALL;
      SAVE=", path, "\\data*.dat;
      ANALYSIS: TYPE = TWOLEVEL;
      MODEL POPULATION:
      %WITHIN% \n", file=gen, append=T, sep="")

    ## baseline loadings are all .3, add .4 if "within" =
        1, but change based on mod/strong

    cat("FW BY y1-y2*", .3+(within*(.4+strong*.1)),
        "y3-y4*", .3+(within*(.4-mod*.3)),
        "y5-y6*", .3+(within*(.4-strong*.1)),
        "FW@1; \n",
        "y1-y2*", 1-(.3+(within*(.4+strong*.1)))^2, "; \n",
        "y3-y4*", 1-(.3+(within*(.4-mod*.3)))^2, "; \n",
        "y5-y6*", 1-(.3+(within*(.4-strong*.1)))^2, "; \n",
        " %BETWEEN% ",
```

Here was my Suggestion ...

```
           "FB BY y1−y2∗", sqrt((.3+(between∗(.4+strong∗.1)))^
               2∗(1+(icc∗(.053−1)))),
           "y3−y4∗", sqrt((.3+(between∗(.4−mod∗.3)))^2 ∗(1+(
               icc∗(.053−1)))),
           "y5−y6∗", sqrt((.3+(between∗(.4−strong∗.1)))^2 ∗
               (1+(icc∗(.053−1)))),"; FB@1;  \n",
           "y1−y2∗", 1+(icc∗(.053−1))−sqrt((.3+(between∗(.4+
               strong∗.1)))^2∗(1+(icc∗(.053−1))))^2, ";",
           "y3−y4∗", 1+(icc∗(.053−1))−sqrt((.3+(between∗(
               .4−mod∗.3)))^2 ∗(1+(icc∗(.053−1))))^2, ";",
          "y5−y6∗", 1+(icc∗(.053−1))−sqrt((.3+(between∗(
              .4−strong∗.1)))^2 ∗(1+(icc∗(.053−1))))^2, ";",
          "\n", file=gen, append=T, sep="")
}
```

Critique that!

- Benefit of re-write is isolation of code writing into a separate function
- We need to work on cleaning up use of "cat" to write files.

Outline

Object Oriented Programming

- A re-conceptualization of programming to reduce programmer error
- OO includes a broad set of ideas, only a few of which are directly applicable to R programming
- The "rise" to pre-eminance of OO indicated by the
  - introduction of object frameworks in existing languages (C++, Objective-C)
  - growth of wholly new object-oriented languages (Java)

Decipher R OO by Intuition

Run the command

```
methods(print)
```

What do you see? I see 170 lines, like so:

```
  [1] print.acf*
  [2] print.anova
  [3] print.aov*
  [4] print.aovlist*
...
[167] print.warnings
[168] print.xgettext*
[169] print.xngettext*
[170] print.xtabs*

   Non-visible functions are asterisked
```

170 print.??? Methods.

- Yes: there are really 170 print “methods”
- No: the R team does not expect or want you to know all of them. Users just run

  ```
  print(x)
  ```

  Try not to worry about “how” the printing is achieved.
- Yes: R team wants package writers to create specialized print methods to control presentation of their output for the object they create.

The R Runtime System Handles the Details

- The user runs

```
print(x)
```

- The R runtime system
  1. notices that x is of a certain type, say "classOfx"
  2. and then the runtime system uses print.classOfX to handle the user's request

print is a “Generic Function”

Definition of “Generic Function”: the function that users call which causes an object-specific function to be called for the work to get done. Examples:“print”, “plot”, “summary”, “anova”

- Generic Function is terminology unique to R(AFAIK)
- In the standard case, a generic function does not do any work. It sends the work to the appropriate “implementation” in a method.

“A standard generic function does no computation other than dispatching a method, but R generic functions can do other coumputations as well before and/or after method dispatch”(Chambers, Software for Data Analysis, p. 398)

print is a “Generic Function” ...

- UseMethod() is the function that declares a function as generic: The R runtime system is warned to “be alert” to usage of the function.
- Example: the print generic function from R source (base package).

```
print <- function(x, ...) UseMethod("print")
```

- Example: the plot generic function from R source (graphics package).

```
plot <- function (x, y, ...)  UseMethod("plot")
```

Here's Where R Gains its Analytical Power

- The generic is just a place holder. User runs print(x), then R knows it is supposed to ask x for its class and then the appropriate thing is supposed to happen. No Big Deal.
- But the statisticians in the S & R projects saw enormous simplifying potential in developing a battery is standard generic accessor functions
  - summary()
  - anova()
  - predict()
  - plot()
  - aic()

Object

Object: self-contained "thing". A container for data.

- Operationally, in R: just about anything on the left hand side in an assignment "<-"
- Each "thing" in R carries with it enough information so that generic functions "know what to do.".
- If there is no function specific to an object, the work is sent to a default method (see print.default).

Class

Definition: As far as R with S3 is concerned, class is a characteristic label assigned to an object (an object can have a vector of classes, such as c("lm", "glm")).

- The class information is used by R do decide which method should be used to fulfill the request.
- Run class(x), ask x what class it inherits from.
- In R, the principal importance of the "class" of an object is that it is used to decide which function should be used to carry out the required work when a generic function is used.
- Classes called "numeric", "integer", "character" are all vector classes

Class ...

```
> y <- c(1, 10, 23)
> class(y)
[1] "numeric"
> x <- c("a", "b", "c")
> x
[1] "a" "b" "c"
> class(x)
[1] "character"
> x <- factor(x)
> class(x)
[1] "factor"
> m1 <- lm(y ~ x)
> class(m1)
[1] "lm"
> m2 <- glm( y ~ x, family=Gamma)
> class(m2)
[1] "glm" "lm"
```

Method, a.k.a, “Method Function”

Definition: The “implementation”: the function that does the work for an object of a particular type.

- When the user runs print(m1), and m1 is from class “lm”, the work is sent to a method print.lm()
- Methods are always named in the format “generic.class”, such as “print.default”, “print.lm”, etc.
- Note: Most methods do not “double-check” whether the object they are given is from the proper class. They count in R’s runtime system to check and then call print.whatever for obejcts of type whatever
- That’s why many methods are “hidden” (can only access via ::: notation)
- Accessing methods directly

Method, a.k.a, “Method Function” ...

- If a method is “exported”, can be called directly via “package::method.class()” format
- If a package is “attached” to the search path, then “method.class()” will suffice, but is not as clear
- If a method is NOT exported, then user can reach into the package and grab it by running “package:::method.class()”

Detour: attributes() Function and Confusing Output

The class is stored as an attribute in many object types. Run attributes()

```
> attributes(x)
$levels
[1] "a" "b" "c"

$class
[1] "factor"

> attributes(m1)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "
     df.residual"
 [9] "contrasts"     "xlevels"       "call"          "terms"
[13] "model"

$class
[1] "lm"

> attributes(y)
```

Detour: attributes() Function and Confusing Output ...

```
NULL
> is.object(y)
 [1]  FALSE
```

- puzzle: why has y no attribute? Why is it not an object?
- Honestly, I'm baffled, I thought "everything in R is an object." (And I still do.)

  If the object does not have a class attribute, it has an implicit class, "matrix", "array" or the result of "mode(x)" (except that integer vectors have the implicit class "integer"). (from ?class in R-2.15.1)

How Objects get “into” Classes

- In older S3 terminology, user is allowed to simply claim that x is from one or more classes

```
class(x) <- c(``lm'', ``glm'', class(x))
```

- That would say x’s class includes “lm” and “glm” as new classes, and also would keep x’s old classes as well.
- The class is an attribute, can be set thusly

```
attr(x, ``class'') <- c(``lm'', ``whateverISay'')
```

- When a generic method “run” is called with x, the R runtime will first try to use run.lm. If run.lm is not found, then run.whateverISay will be tried, and if that fails, it falls back to run.default.

How Objects get “into” Classes: S4

- S4 has more structure, makes classes & methods work more like truly object oriented programs.
- S4 classes are defined with a list of variables BEFORE objects are created.
- Variables are typed!
- Example imitates Matloff, p. 223

```
 setClass("pjfriend", representation(
name="character",
gender="factor",
food="factor",
age="integer"))
```

- Create an instance of class pjfriend (Note: to declare an integer, add letter “L” to end of number).

How Objects get “into” Classes: S4 ...

```
william <- new("pjfriend", name = "william", gender =
  factor("male"), food=factor("pizza"), age=33L)
william
```

```
An object of class "pjfriend"
Slot "name":
[1] "william"

Slot "gender":
[1] male
Levels: male

Slot "food":
[1] pizza
Levels: pizza

Slot "age":
[1] 33
```

How Objects get “into” Classes: S4 ...

```
jane <- new("pjfriend", name="pumpkin", gender =
    factor("female"), food=factor("hamburger"), age=21L
    )
jane
```

```
An object of class "pjfriend"
Slot "name":
[1] "pumpkin"

Slot "gender":
[1] female
Levels: female

Slot "food":
[1] hamburger
Levels: hamburger

Slot "age":
[1] 21
```

- jane and william are *instances* of class “pjfriend”

How Objects get “into” Classes: S4 ...

- The variables inside an S4 object are called *“slots”* in R
- “slot” would be called “instance variable” in most OO-languages)
- values in slots can be retrieved with symbol `@`, not $

Implement an S4 method

- Step 1. Write a function that can receive a function of type "pjfriend" and do something with it.
- Step 2. Use setMethod to tell the R system that the function implements the method that is called for.
- setMethod "wraps" a function.

```
setMethod("some-generic-function-name", "pjfriend",
          function(x) {
              #do something with x
              }
)
```

Difficult to Account For Changes between S3 and S4

I think it is difficult to explain some of the notational and terminological changes between S3 and S4.

- If you type an S4 object's name on the command line

  ```
  > x
  ```

  the R runtime looks for a method "show.class" (where class is the class of x).
- Why change from "print" to "show"? (IDK)
- Why change the "accessor" symbol from $ to @ ?
- Why call things accessed with @ "slots" rather than instance variables?