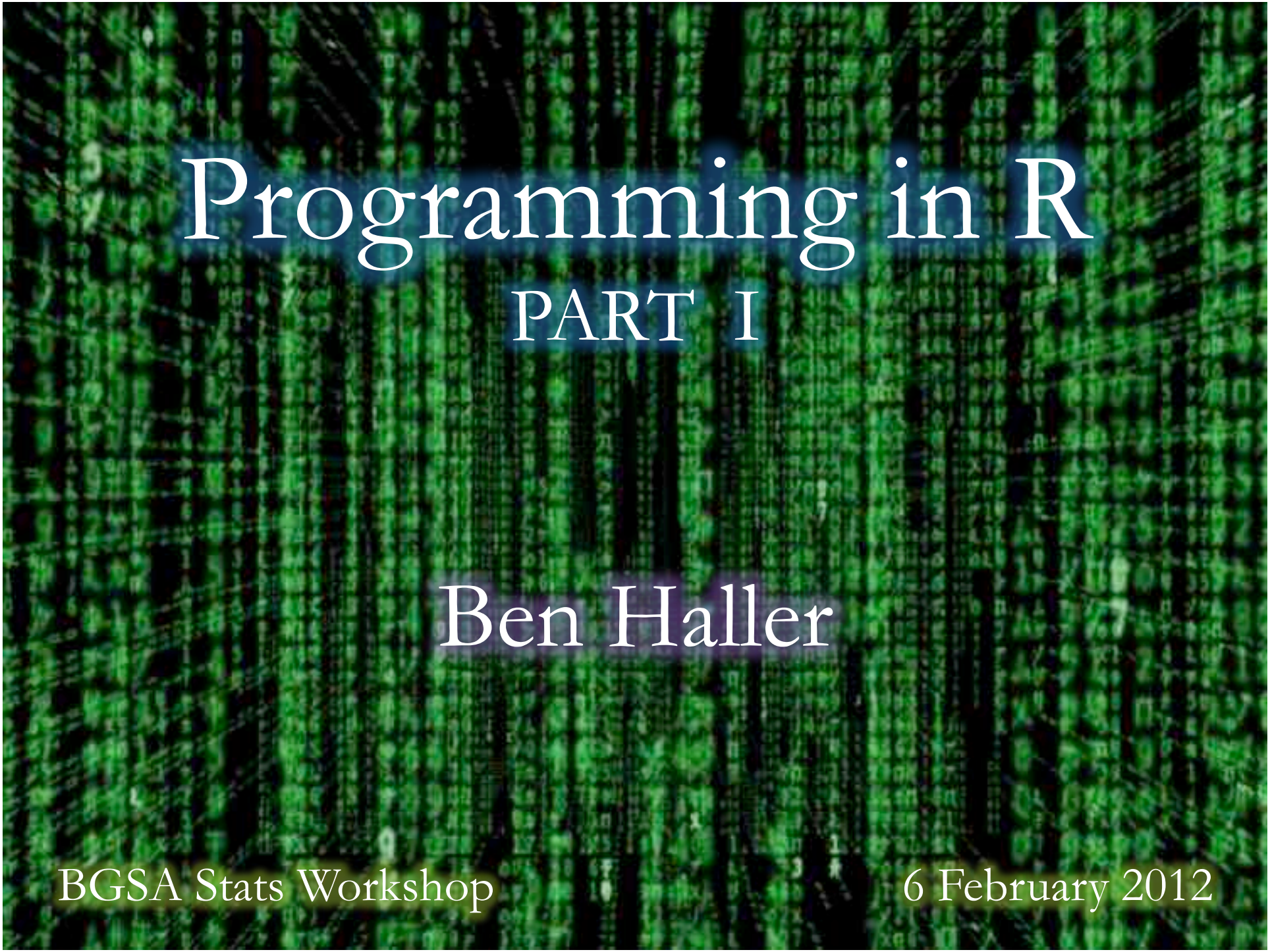
Pre-workshop:

- Install R (or use it on the computers here):
  - http://cran.r-project.org/
- Install an R environment, such as R Studio:
  - http://rstudio.org/
- Download the slides and the .R script:
  - http://bit.ly/yRjShO
- Twiddle your thumbs impatiently


Programming in R
PART I
Ben Haller
BGSA Stats Workshop
6 February 2012

Outline

- Flow control
- Subsetting and sorting
- Writing functions

- Coding for speed
- Debugging in R
- Caveats of R

Format

- Input to R:

```
> say.hello <- function(target)
+ {
+     cat("Hello, ", target,
+         "!", sep="")
+ }
> say.hello("world")
```

- Output from R:

```
Hello, world!
```

- R stuff you don't need to type:

```
> say.goodbye()
```

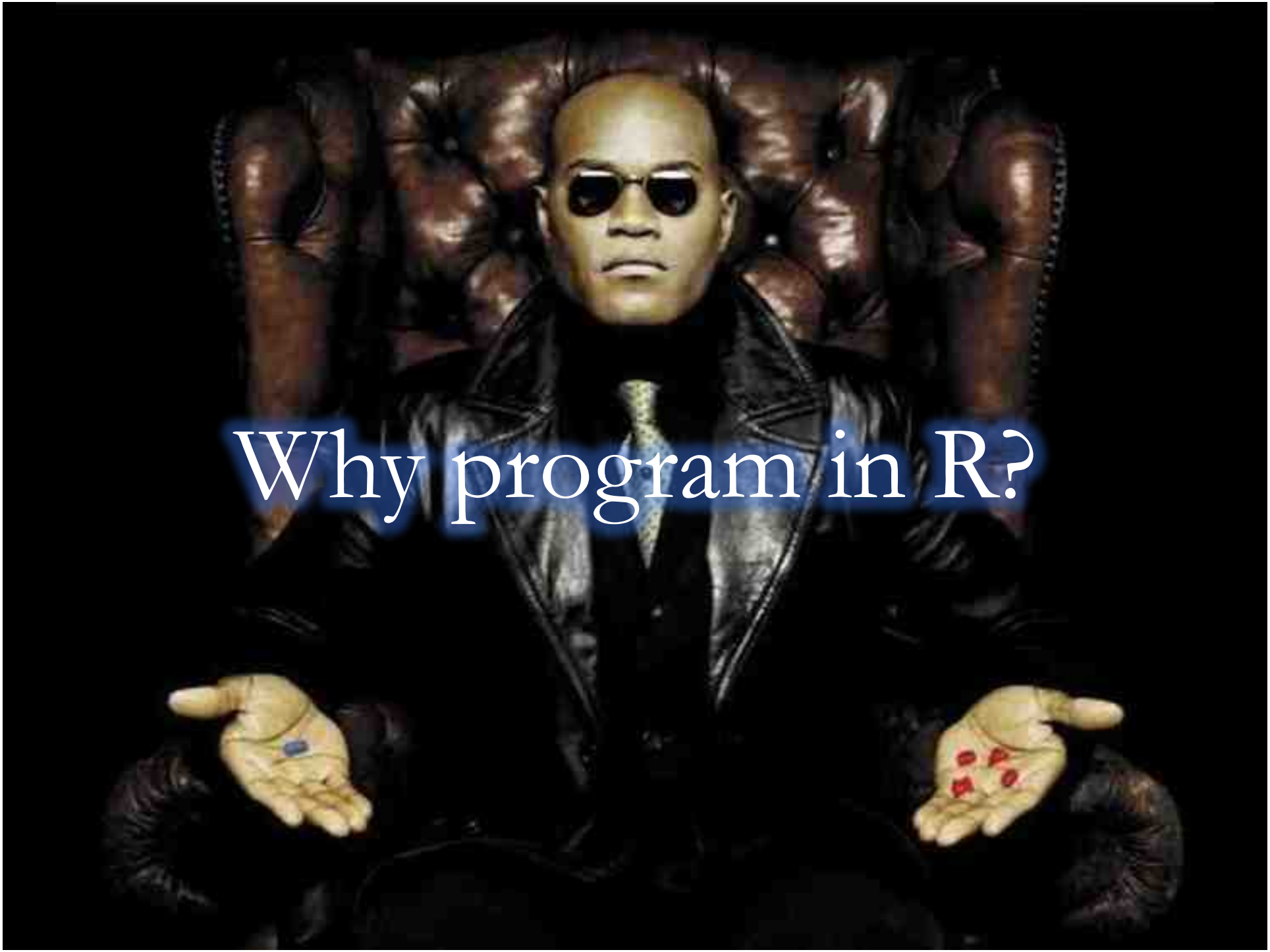
Help

- Don't forget you can get help using R's built-in help function:

```
> ?cat
> ?"["
```

help:
`?cat`

- Raise your hand at any time to get help; don't be shy!
- I recommend Tom Short's R Reference Card:
  - http://cran.r-project.org/doc/contrib/Short-refcard.pdf


Why program in R?

Why Program in R?

- Reuse and share your code:
  - Achieve greater consistency
  - Avoid copy/paste errors
  - Avoid reinventing the wheel
  - Redo your analysis quickly and easily
- Use R to do repetitive tasks for you
- Understand what R is doing to your data
- Do analyses that nobody has prepackaged
- It's fun! (no, really!)

Flow Control

`?Control`

Flow Control

- Execute statements conditionally with:
  - `if` and `if` / `else` statements
- Execute statements multiple times with:
  - `for` loops
  - `while` loops
  - `repeat` loops
- Modify loop execution with:
  - `break` statements
  - `next` statements

Flow Control: `if` / `else`

`?Control`

- `if` and `if` / `else` statements are good for:
  - checking for problems or violations of assumptions
  - treating different rows of your dataframe differently
  - testing for the existence of a file or a variable
- Syntax:

```
> if (condition) <expression>
```

```
> if (condition) <expression> else <expression>
```

- Example:

```
> if (2 + 2 == 4) print("Arithmetic works.")
```

```
[1] "Arithmetic works."
```

- Beware of R's expression parsing! This:

```
> if (2 + 2 == 4) print("Arithmetic works.")
> else print("Houston, we have a problem.")
```

does not work! You get this:

```
> if (2 + 2 == 4) print("Arithmetic works.")
[1] "Arithmetic works."
> else print("Houston, we have a problem.")
Error: unexpected 'else' in "else"
```

2

So do this:

```
> if (2 + 2 == 4) {
+     print("Arithmetic works.")
+ } else {
+     print("Houston, we have a problem.")
+ }
```

3

Flow Control: Loops

```
?Control
```

- Loops are good for:
  - doing something for every element of an object
  - doing something until the processed data runs out
  - doing something for every file in a folder
  - doing something that can fail, until it succeeds
  - iterating a calculation until it converges
- Syntax:

```
> for (variable in sequence) <expression>
```

```
> while (condition) <expression>
```

```
> repeat <expression>
```

?cat

- The most common type is the `for` loop:

```
> for (variable in sequence) <expression>
```

- For example:

```
> for (i in 1:5) cat(i, " ")
```

```
1  2  3  4  5
```

4

- The "sequence" can be almost anything:

```
> for (i in c("green","egg","ham")) cat(i, " ")
```

```
green  egg  ham
```

5

- Very often you loop over a dataset:

```
> for (i in row.names(USArrests)) cat(i, " ")
```

```
Alabama  Alaska  Arizona  Arkansas...
```

6

?NROW

- The "expression" part of the loop:

```
> for (variable in sequence) <expression>
```

can also be anything, and is often a "compound statement":

```
> for (i in 1:NROW(USArrests))
+ {
+     name <- row.names(USArrests)[i]
+     if (substr(name, 1, 1) == "C")
+         print(name)
+ }
```

```
[1] "California"
[1] "Colorado"
[1] "Connecticut"
```


- Instead of just printing row names, we could be doing calculations on each row of the dataset

- The examples I show you are not always best practice in R! Take the last example:

```
> for (i in 1:NROW(USArrests))
+ {
+      name <- row.names(USArrests)[i]
+      if (substr(name, 1, 1) == "C")
+          print(name)
+ }
```


- This is much better done as:

```
> s.names <- row.names(USArrests)
> s.names[substr(s.names, 1, 1) == "C"]
```

- Or even better:

```
> grep("^C", row.names(USArrests), value=TRUE)
```

- But this wouldn't teach you about loops!

- Normally, loops iterate over and over until they finish. To change this behavior, you can use:

```
break
```

which breaks out of the loop's execution completely, or:

```
next
```

which stops executing the current iteration, and jumps to executing the next iteration.

- Example:

```
> i <- 0
> repeat {
+     i <- i + 1
+     if (i %% 2 == 1) next  # skip this loop
+     print(i)
+     if (i == 10) break     # stop looping
+ }
```

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

- Note that `%%` is the "modulo" operator: it gives the remainder after integer division.
- For example, `9%%3 == 0`, but `10%%3 == 1`.

- The same loop can be coded in different ways:

```
> i <- 1
> repeat {
+     i <- i + 1
+     if (i %% 2 == 1) next
+     print(i)
+     if (i == 10) break
+ }
```

8

```
> i <- 0
> while (i <= 10)
+ {
+     i <- i + 1
+     if (i %% 2 == 0) print(i)
+ }
```

9

```
> for (i in 1:10)
+     if (i %% 2 == 0) print(i)
```

10

```
> for (i in seq(2, 10, by=2)) print(i)
```

11

- A (sort of) practical example:

```
> set.seed(3)
> d <- data.frame(x=rnorm(20), y=rnorm(20),
    t=sample.int(3, size=20, replace=TRUE))
> head(d)
```

```
            x          y t
1 -0.96193342 -0.5784837 3
2 -0.29252572 -0.9423007 2
3  0.25878822 -0.2037282 2
4 -1.15213189 -1.6664748 1
5  0.19578283 -0.4844551 3
6  0.03012394 -0.7410727 3
```

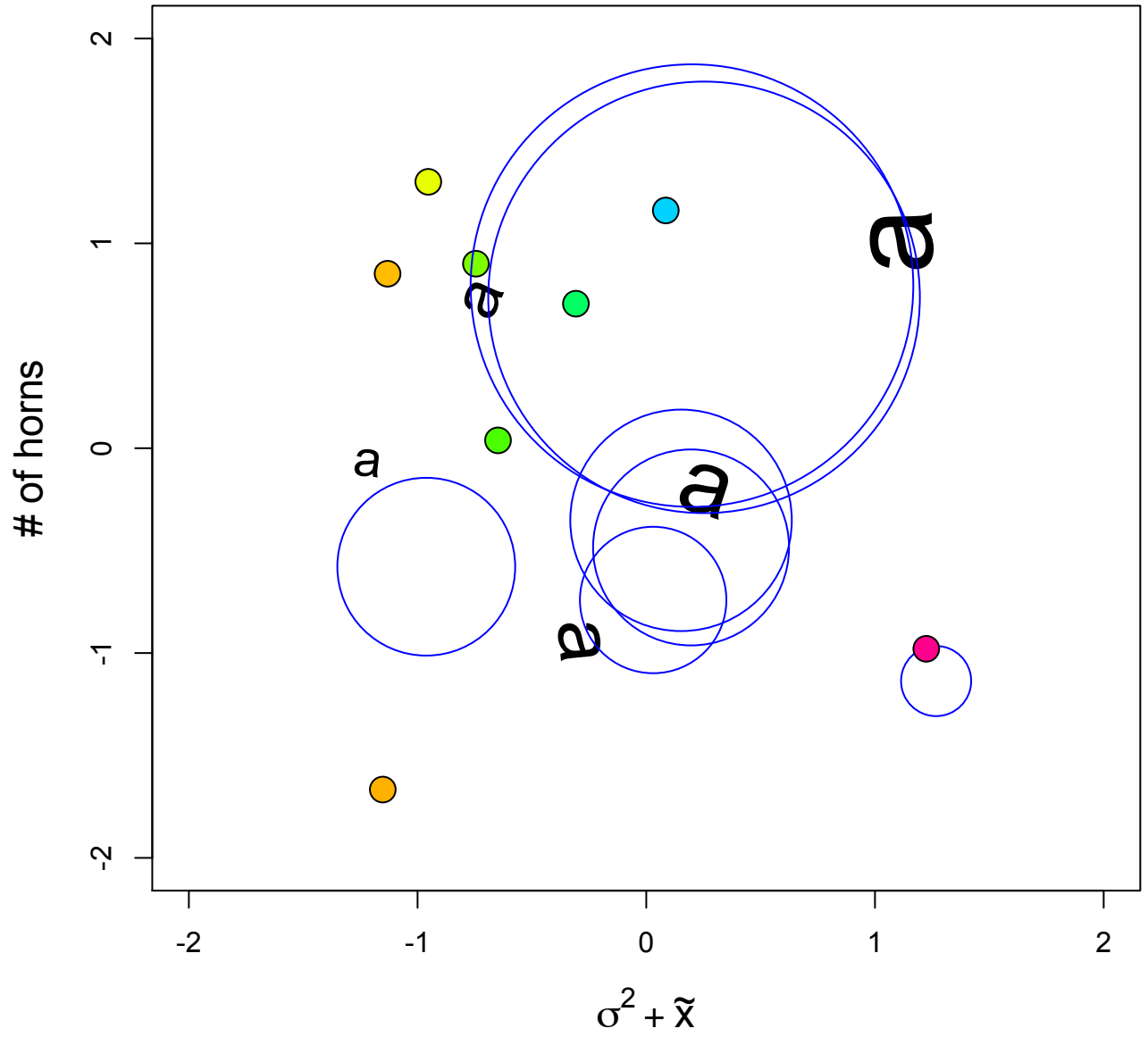
- And now we want to plot this data, in different ways depending upon the value of t!

- First we make an empty plot area:

```
> plot(x=d$x, y=d$y, xlim=c(-2,2), ylim=c(-2,2),
+     xlab=expression(sigma^2+tilde(x)),
+     ylab="# of horns", type="n", cex.lab=1.4)
```

- Then we loop over the dataset:

```
> for (i in 1:20)
+ {
+     if (d$t[i] == 1)
+         points(d$x[i], d$y[i], bg=hsv((d$x[i]+2)/4),
+             cex=2.0, pch=21)
+     else if (d$t[i] == 2)
+         text(d$x[i], d$y[i], "a", cex=d$x[i]+3.0,
+             srt=d$y[i]*90)
+     else
+         points(d$x[i], d$y[i], col="blue",
+             cex=(d$y[i]+1.5)*15, pch=1)
+ }
```


a
a
a
a
a
# of horns
2
1
0
-1
-2
-2
-1
0
1
2
σ² + x̃

`?Control`

Exercise 1

- Use the built-in dataset called `USArrests`
- Write a **simple** piece of code with a loop, plus an `if` statement if you wish, to do something
- Note there are ideas and hints in the .R file
- Show your code to the person sitting next to you, and have them guess what your code does

Flow Control: `ifelse`

?ifelse

- `if` and `if` / `else` test only a single condition
- `ifelse` can be used to test a vector of conditions and produce a vector of results:

```
> ifelse(test, yes, no)
```

- This can be particularly useful for making complex plots that encode information:

```
plot(..., pch = ifelse(test, 3, 19))
```

```
plot(..., col = ifelse(test, "black", "red"))
```

```
plot(..., lwd = ifelse(test, 1, 3))
```

```
plot(..., lty = ifelse(test, "solid", "dashed"))
```

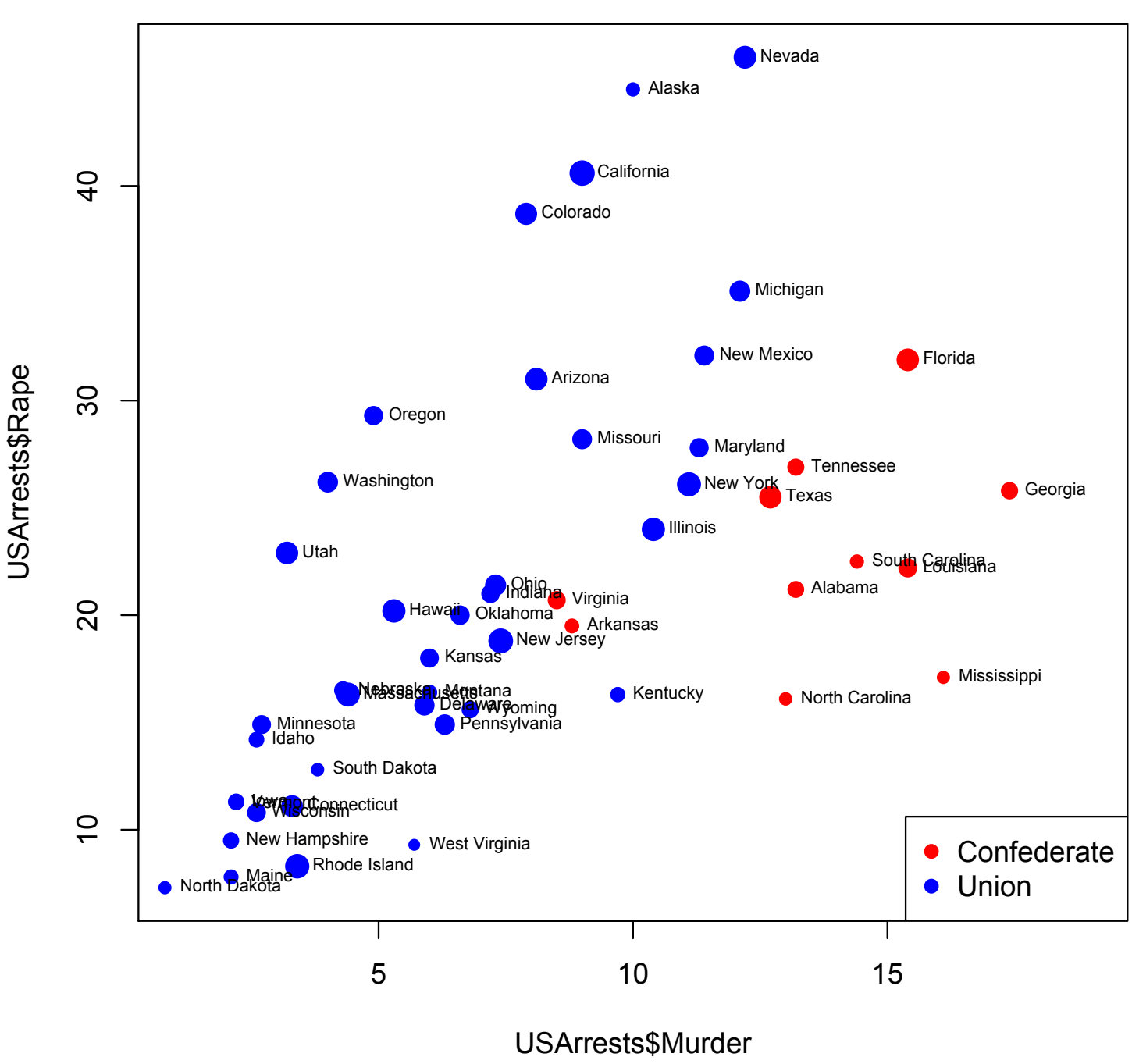
Flow Control: `ifelse`


```
> confederateStates <- c("Florida", "Georgia",
+     "North Carolina", "South Carolina", "Virginia",
+     "Alabama", "Mississippi", "Tennessee",
+     "Arkansas", "Louisiana", "Texas")
>
> USArrests$Confederate <- row.names(USArrests) %in%
+     confederateStates
>
> head(USArrests)
```

	Murder	Assault	UrbanPop	Rape	Confederate
Alabama	13.2	236	58	21.2	TRUE
Alaska	10.0	263	48	44.5	FALSE
Arizona	8.1	294	80	31.0	FALSE
Arkansas	8.8	190	50	19.5	TRUE
California	9.0	276	91	40.6	FALSE
Colorado	7.9	204	78	38.7	FALSE

Flow Control: `ifelse`


```
> plot(USArrests$Murder,
+     USArrests$Rape,
+     pch=19,
+     cex=(USArrests$UrbanPop / 50),
+     xlim=c(0,19),
+     col=ifelse(USArrests$Confederate,"red","blue"))
>
> text(USArrests$Murder,
+     USArrests$Rape,
+     pos=4,
+     offset=(USArrests$UrbanPop / 130),
+     labels=row.names(USArrests),
+     cex=0.6)
>
> legend(x="bottomright", pch=19, col=c("red",
+     "blue"), legend=c("Confederate", "Union"))
```


Nevada
Alaska
California
Colorado
Michigan
New Mexico
Florida
Arizona
Oregon
Missouri
Maryland
Tennessee
Washington
New York
Texas
Georgia
Illinois
Utah
South Carolina
Louisiana
Ohio
Indiana
Alabama
Virginia
Hawaii
Oklahoma
Arkansas
New Jersey
Kansas
Mississippi
Kentucky
North Carolina
Delaware
Wyoming
Minnesota
Pennsylvania
Idaho
South Dakota
Connecticut
Wisconsin
New Hampshire
West Virginia
Rhode Island
Maine
North Dakota
Confederate
Union
USArrests$Rape
USArrests$Murder
10
20
30
40
5
10
15

Subsetting and Sorting

?Extract

Subsetting and Sorting

- The `[` operator in R is very powerful for:
  - selecting a subset of rows, columns or elements
  - changing the order of rows or columns (sorting)
- Syntax:

```
> x[i]
```

- Example:

```
> squares <- ((1:10)^2); squares
 [1]   1   4   9  16  25  36  49  64  81 100
> squares[7]
[1] 49
```

- One way of using `[` is with indices:

```
> squares[c(2, 4, 6, 8, 10)]
```

```
[1]   4  16  36  64 100
```

15

- That gave us a new vector, with only the elements that we requested by index
- The order of the indices given determines the order of the values in the new vector:

```
> squares[c(6, 4, 2, 10, 8)]
```

```
[1]  36  16   4 100  64
```

16

- Why is that interesting? This is how you sort things in R!

- For example, suppose we have a vector:

```
> set.seed(0); x<- rnorm(5); print(x, digits=3)
```

```
[1]  1.263 -0.326  1.330  1.272  0.415
```

- If we want to sort it, first we get a vector of indices from `order`:

```
> order(x)
```

```
[1] 2 5 1 4 3
```


- Then we use `[` with that "order vector":

```
> print(x[order(x)], digits=3)
```

```
[1] -0.326  0.415  1.263  1.272  1.330
```

- Another way of using `[` is with a vector of logical (`TRUE` or `FALSE`) values:

```
> squares[c(TRUE, FALSE, TRUE, FALSE, TRUE,
+     FALSE, TRUE, FALSE, TRUE, FALSE)]
```

```
[1]    1    9   25   49   81
```

- R will repeat the logical vector to make it the same length as the data:


```
> squares[c(TRUE, FALSE)]    # gets repeated!
```

```
[1]    1    9   25   49   81
```

- Sometimes this is useful; more often, it is a source of very nasty bugs, so beware!

- Often logical vectors are based on data:

```
> squares > 50
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
TRUE  TRUE  TRUE
```

```
> squares[squares > 50]
```

```
[1]  64  81 100
```



```
> squares %% 10 == 1
```

```
[1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
FALSE  TRUE FALSE
```

```
> squares[squares %% 10 == 1]
```

```
[1]  1 81
```

?Logic

- Logical vectors combine with `&` and `|`:

```
> (squares > 50) & (squares %% 10 == 1)
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
FALSE  TRUE FALSE
```

```
> squares[(squares >50) & (squares %% 10 == 1)]
```

```
[1] 81
```



```
> (squares > 50) | (squares %% 10 == 1)
```

```
[1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
TRUE  TRUE  TRUE
```

```
> squares[(squares >50) | (squares %% 10 == 1)]
```

```
[1]   1  64  81 100
```

?Logic

- Note that `&&` and `||` are different!

```
> (squares > 50) && (squares %% 10 == 1)
```

```
[1] FALSE
```

21

- Basically:
  - The `&` and `|` operators act on entire vectors, elementwise, and they return a vector
  - Use `&` and `|` when manipulating logical vectors
  - The `&&` and `||` operators act on single values, and they always return just a single value
  - Use `&&` and `||` when evaluating a Boolean condition, such as in an `if` statement

?which

- `which` converts a logical vector to indices:

```
> which(squares > 50)
```

```
[1]  8  9 10
```

22

- Then you can use those indices to subset:

```
> squares[which(squares > 50)]
```

```
[1]  64  81 100
```

23

- Note what is really happening here:
  1. `squares > 50` returns a logical vector
  2. `which(~)` returns indices for `TRUE` elements
  3. `squares[~]` returns the values at those indices

```
?intersect
```

- Often `which` is useful with functions such as `intersect`, `union`, and `%in%`:

```
> which(squares %% 10 == 1)
```

```
[1] 1 9
```

```
> which(squares > 50)
```

```
[1]  8  9 10
```

```
> intersect(which(squares %% 10 == 1),
+     which(squares > 50))
```

```
[1] 9
```

```
> squares[intersect(which(squares %% 10 == 1),
+     which(squares > 50))]
```

```
[1] 81
```

- There is a sort of parallelism between:
  - using indices with `which`, and combining sets of indices using operators like `intersect`, `union`, `setdiff`, and `%in%`
  - using logical vectors, and manipulating and combining those vectors using operators like `&`, `|`, and `!`
- Which strategy is best for a given problem may be a matter of taste in some cases; in other cases, one strategy will work much better than the other. Be flexible!

- A (sort of) practical example with a constructed dataset:

```
> set.seed(5)
> df <- data.frame(
+     species=sample(letters, 20, rep=TRUE),
+     type=sample(c(1, 2), 20, rep=TRUE))
> df
```

```
   species type
1        f    2
2        r    2
3        x    1
4        h    1
5        c    1
6        s    1
7        n    1
8        v    2
9        y    1
10       c    2
...
```

```
   species type
1        f    2
2        r    2
3        x    1
4        h    1
5        c    1
6        s    1
7        n    1
8        v    2
9        y    1
10       c    2
...
```

- Problem: we want to select only rows for species that have both a type 1 and a type 2 entry!

- Start by getting a species list:

```
> sp_list <- unique(df$species)
> sp_list
```

```
 [1] f r x h c s n v y m i o g k
Levels: c f g h i k m n o r s v x y
```

- Note `species` is a factor according to R, but this won't matter to us
- Now we can loop over this species list, and check each species, one by one, for inclusion

- So let's do that:

```
> has_1 <- logical(0)
>
> for (species in sp_list)
+ {
+     type1rows <- df[(df$species == species)
+         & (df$type == 1), ]
+     has_1 <- c(has_1, NROW(type1rows) >= 1)
+ }
>
> has_1
```

```
[1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

- The `has_1` vector now has `TRUE` for species that have a type 1 entry

- And let's do the same for type 2:

```
> has_2 <- logical(0)
>
> for (species in sp_list)
+ {
+     type2rows <- df[(df$species == species)
+         & (df$type == 2), ]
+     has_2 <- c(has_2, NROW(type2rows) >= 1)
+ }
>
> has_2
```

```
 [1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

- The `has_2` vector now has `TRUE` for species that have a type 2 entry

- We can use `&` to find which species have both types:

```
> has_1 & has_2
```

```
[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
> sp_int <- sp_list[has_1 & has_2]
> sp_int                  # species of interest
```

```
[1] f x c v
Levels: c f g h i k m n o r s v x y
```

- So we want species f, x, c, and v
- How do we select just those dataset rows?

- We can use `%in%` to do this:

```
> df$species %in% sp_int
```

```
[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
TRUE FALSE  TRUE FALSE TRUE
```

```
> df_12 <- df[df$species %in% sp_int, ]
> df_12
```

```
   species type
1        f    2
3        x    1
5        c    1
8        v    2
10       c    2
16       f    1
18       x    2
20       v    1
```

- It would be nice if that were sorted...

```
> df_12_o <- order(df_12$species, df_12$type)
> df_12_o
```

```
[1] 3 5 6 1 8 4 2 7
```

```
> df_12[df_12_o, ]
```

```
   species type
5        c    1
10       c    2
16       f    1
1        f    2
20       v    1
8        v    2
3        x    1
18       x    2
```

- Problem solved. We have a dataframe that contains only the rows for species that have entries of type 1 and type 2, and they are sorted.

`?Extract`

Exercise 2

- Continue using the `USArrests` dataset
- Write a **simple** piece of code that uses subsetting to calculate some statistics about subsets of it, and/or to sort it in interesting ways
- Show your code to the person sitting next to you, and have them guess what your code does

Writing Functions

```
?"function"
```

Writing Functions

- Much of the heavy lifting in R is done by functions. They are useful for:
  - performing a task repeatedly, but configurably
  - making your code more readable
  - sharing code between different analyses
  - sharing code with other people
  - modifying R's built-in functionality
- Syntax:

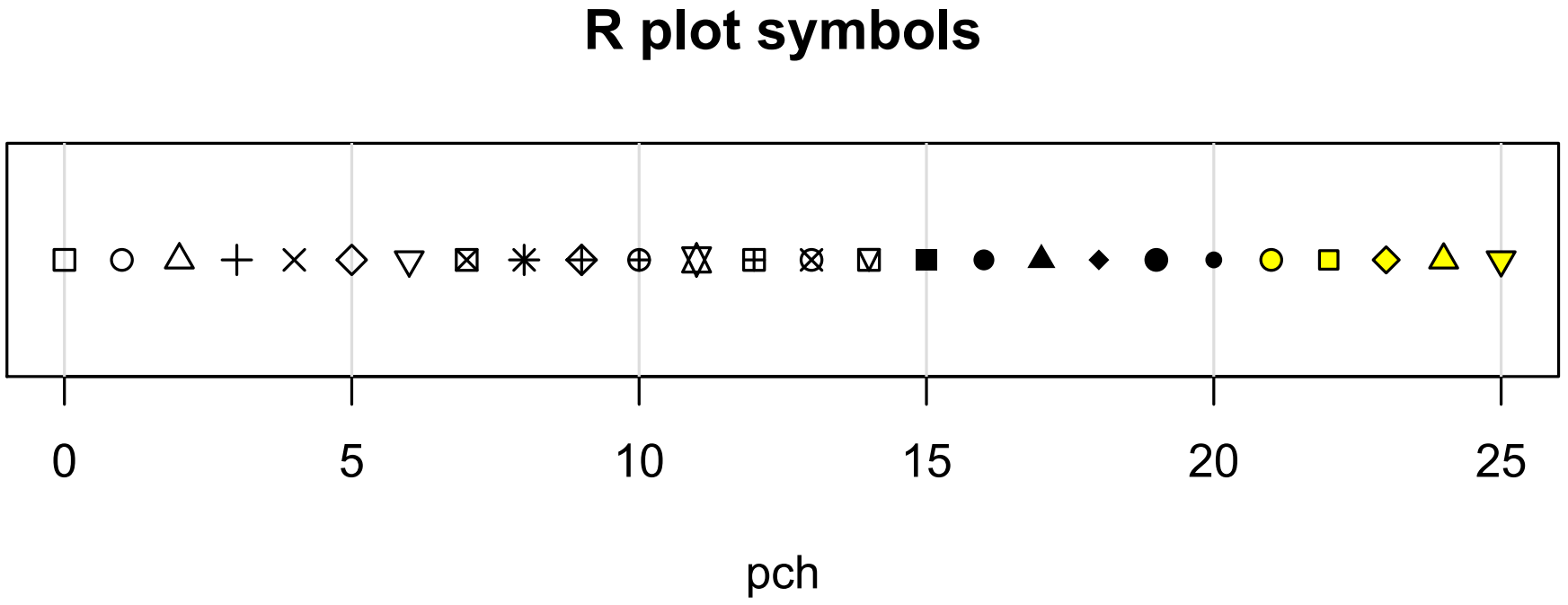
```
> my_function <- function(...) <expression>
```

- Example:

```
> plotSymbols <- function()
+ {
+     yvec <- rep(1.0, length.out=26)
+
+     plot(x=0:25, y=yvec, type="n",
+         yaxt="n", ylab="", xlab="pch",
+         main="R plot symbols")
+
+     for (i in seq(from=0, to=25, by=5))
+         abline(v=i, col="#DDDDDD")
+
+     points(x=0:25, y=yvec, pch=0:25,
+         bg="yellow")
+ }
```

- This function is very simple; it always does exactly the same thing. But it's a useful thing!

```
> plotSymbols()
```



- Another example:
  - This function takes an **argument** (also called a **parameter**), `x`, which it operates upon:

```
> stderr <- function(x) sqrt(var(x)/length(x))
> stderr(rnorm(100))
```

```
[1] 0.09499494
```

```
> stderr(rnorm(1000))
```

```
[1] 0.03200371
```


- Since R doesn't include a standard error function, it's useful to be able to define one!

- You can look at the built-in functions in R:

```
> xor
```

```
function (x, y)
{
    (x | y) & !(x & y)
}
<environment: namespace:base>
```


- This calculates "exclusive or":

```
> xor(FALSE, FALSE)
```

```
[1] FALSE
```

```
> xor(FALSE, TRUE)
```

```
[1] TRUE
```

```
> xor(TRUE, FALSE)
```

```
[1] TRUE
```

```
> xor(TRUE, TRUE)
```

```
[1] FALSE
```

- But the standard `xor` function can't handle "fuzzy" truth values between 0 and 1:

```
> xor(1.0, 0.0)    # right
```

```
[1] TRUE
```

28

```
> xor(1.0, 0.001)  # note lack of fuzziness
```

```
[1] FALSE
```

- So we can modify it!

```
> xor <- function(x, y, fuzzy=FALSE)
+ {
+   if (fuzzy)
+       return(pmin(pmax(x, y), pmax(1-x, 1-y)))
+   else
+       return((x | y) & !(x & y))
+ }
```

29

- Let's test our new `xor`:

```
> xor(1.0, 0.0, fuzzy=TRUE)
```

```
[1] 1
```

```
> xor(1.0, 0.001, fuzzy=TRUE)
```

```
[1] 0.999
```


- Importantly, `xor` is unchanged when `fuzzy==FALSE`, which is the default:

```
> xor(TRUE, FALSE)
```

```
[1] TRUE
```

```
> xor(TRUE, TRUE)
```

```
[1] FALSE
```

- This ability to modify built-in functions can be very powerful! But be careful!

- If we screw up, we can restore `xor`:

```
> rm(xor)
> xor
```

```
function (x, y)
{
    (x | y) & !(x & y)
}
<environment: namespace:base>
```



```
> xor(1.0, 0.0, fuzzy=TRUE)
```

```
Error in xor(1, 0, fuzzy = TRUE) : unused
argument(s) (fuzzy = TRUE)
```

- Still, use caution when modifying R:
  - unexpected side effects may occur
  - the code for R functions may change

?sapply

- Functions can be "anonymous":

```
> sapply(1:10, FUN=function(x) {
+      x + 3
+ } )
```

31

```
[1]  4  5  6  7  8  9 10 11 12 13
```

- Anonymous functions:
  - have no name (uh, yeah)
  - are typically "called back" to repeatedly
  - cease to exist once they have been used
- There are a raft of "apply"-type functions that take anonymous functions as parameters:
  - `apply`, `sapply`, `lapply`, `mapply`, `rapply`, …

- The `apply` family of functions are good for processing data. Remember the example:

```
> has_1 <- logical(0)
> for (species in sp_list) {
+     type1rows <- df[(df$species == species)
+         & (df$type == 1), ]
+     has_1 <- c(has_1, NROW(type1rows) >= 1)
+ }
```

25

- This created a vector with `TRUE` for each species in `sp_list` having a `type==1` entry. We can now do this more easily:

```
> has_1 <- sapply(sp_list, FUN=function(x) {
+   any((df$species == x) & (df$type == 1))
+ } )
```

32

Exercise 3: Mystery function!

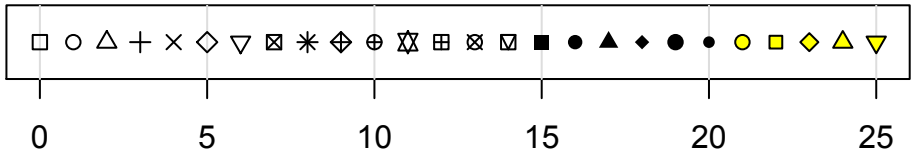
```
> sol <- function()
+ {
+     objs <- objects(name=.GlobalEnv)
+     osizes <- sapply(objs, function(x)
          { object.size(get(x)) } )
+     oclasses <- sapply(objs, function(x)
          { class(get(x))[1] } )
+     o <- data.frame(obj=as.character(objs),
          osize=as.numeric(osizes),
          oclass=as.factor(oclasses),
          stringsAsFactors=FALSE)
+     o <- o[o$oclass != "function",]
+     top_objs <- head(o[order(o$osize,
          decreasing=TRUE),], 20)
+     row.names(top_objs) <- NULL
+
+     top_objs
+ }
```

- Your functions can take multiple arguments:

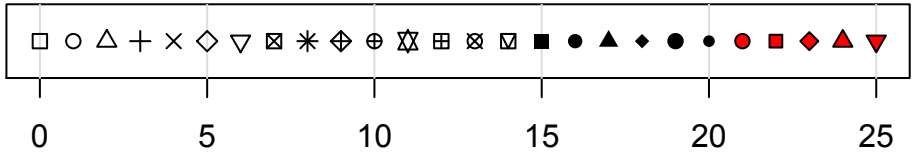
```
> plotSymbols <- function(cex=1.0, col="black",
+     bg="yellow")
+ {
+     yvec <- rep(1.0, length.out=26)
+
+     plot(x=0:25, y=yvec, type="n",
+         yaxt="n", ylab="", xlab="pch",
+         main="R plot symbols")
+
+     for (i in seq(from=0, to=25, by=5))
+         abline(v=i, col="#DDDDDD")
+
+     points(x=0:25, y=yvec, pch=0:25,
+         cex=cex, col=col, bg=bg)
+ }
```

- Now we can use those arguments to make different plots:

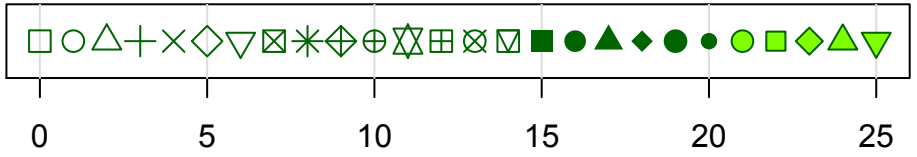
```
> plotSymbols()
```




```
> plotSymbols(bg="red")
```




```
> plotSymbols(cex=1.5, col="darkgreen",
+     bg="chartreuse")
```



- Let's look more closely at our parameters:

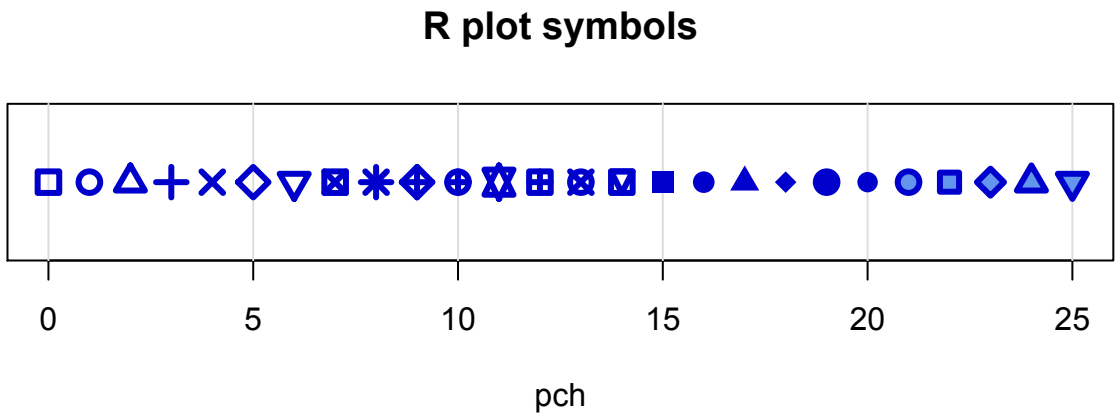
```
> plotSymbols <- function(cex=1.0, col="black",
+     bg="yellow")
+ {
+     # other stuff is left out here...
+     points(x=0:25, y=yvec, pch=0:25,
+         cex=cex, col=col, bg=bg)
+ }
```


- Notice that we're just passing them on; we are "wrapping" the call to `points`:
  - this is bad enough for `cex`, `col`, and `bg`, but `points` can take many more arguments!
  - our default values override those of `points`
  - arguments can't be left out, and conflicting arguments present a big problem

- When you want to "wrap" another function, use `...` to pass arguments through
- For example:

```
> plotSymbols <- function(...)
+ {
+     yvec <- rep(1.0, length.out=26)
+
+     plot(x=0:25, y=yvec, type="n",
+         yaxt="n", ylab="", xlab="pch",
+         main="R plot symbols")
+
+     for (i in seq(from=0, to=25, by=5))
+         abline(v=i, col="#DDDDDD")
+
+     points(x=0:25, y=yvec, pch=0:25, ...)
+ }
```

```
> plotSymbols(cex=1.5, col="blue3",
+     bg="cornflowerblue", lwd=3.0)
```



- Notice that now we can pass a parameter, `lwd`, that was not supported by the old code!
  - the old way: we have to add `lwd` to our function
  - the new way: we just use `lwd` and it works!

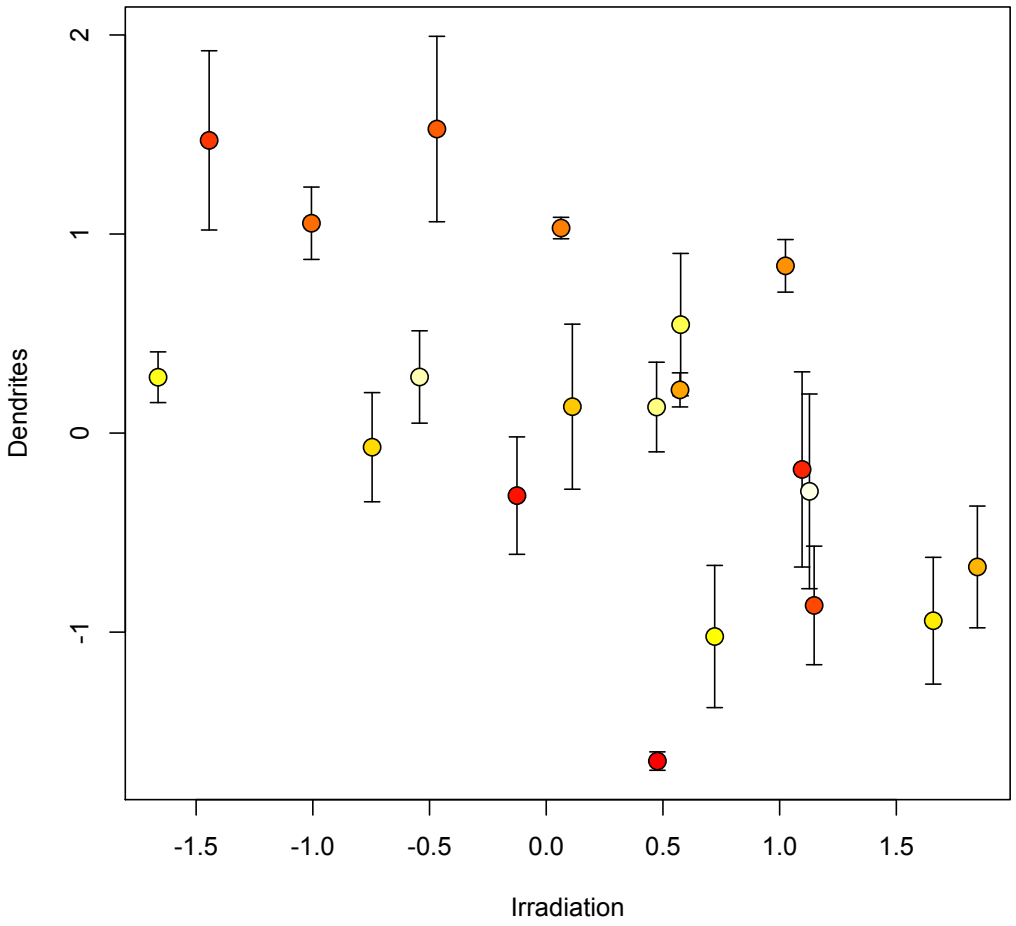
- Wrapping calls to `plot` and plotting commands like `points`, `lines`, and `arrows` is very common, so `...` comes in useful!
- You can wrap multiple calls, sometimes:

```
> error_bar_plot <- function(x, y, err, ...)
+ {
+     yl <- range(y-err, y+err)
+     plot(x, y, ylim=yl, type="n", ...)
+     arrows(x0=x, y0=y-err, x1=x, y1=y+err,
+         length=0.05, angle=90, code=3)
+     points(x, y, ...)
+ }
```

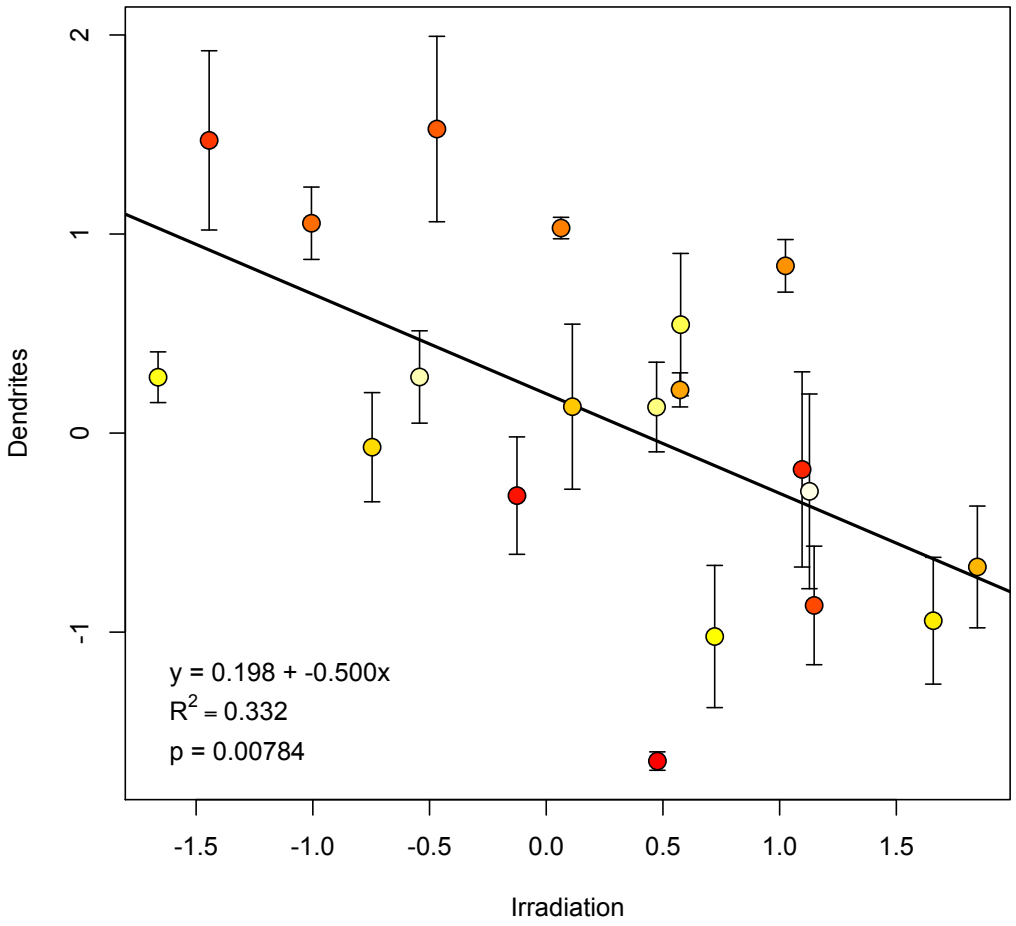

- All arguments beyond `x`, `y`, and `err` will be passed on to both `plot` and `points`.

```
> set.seed(16)
> x <- rnorm(20)
> y <- rnorm(20)
> err <- runif(20, max=0.5)
>
> error_bar_plot(
+     x=x,
+     y=y,
+     err=err,
+     pch=21,
+     col="black",
+     bg=heat.colors(20),
+     cex=1.5,
+     xlab="Irradiation",
+     ylab="Dendrites")
```


- Note that some arguments go to `plot`, and some to `points`. R is smart – sometimes.



- Nice correlation, considering it's random data!



- A bit of extra-credit code is given to make a spiffier plot that shows the correlation

```
?"function"
```

Exercise 4

- Continue using the `USArrests` dataset
- Write a **simple** function to analyze or plot one column of this dataframe; then call your function for each of the columns
- Show your code to the person sitting next to you, and have them guess what your code does

Getting help

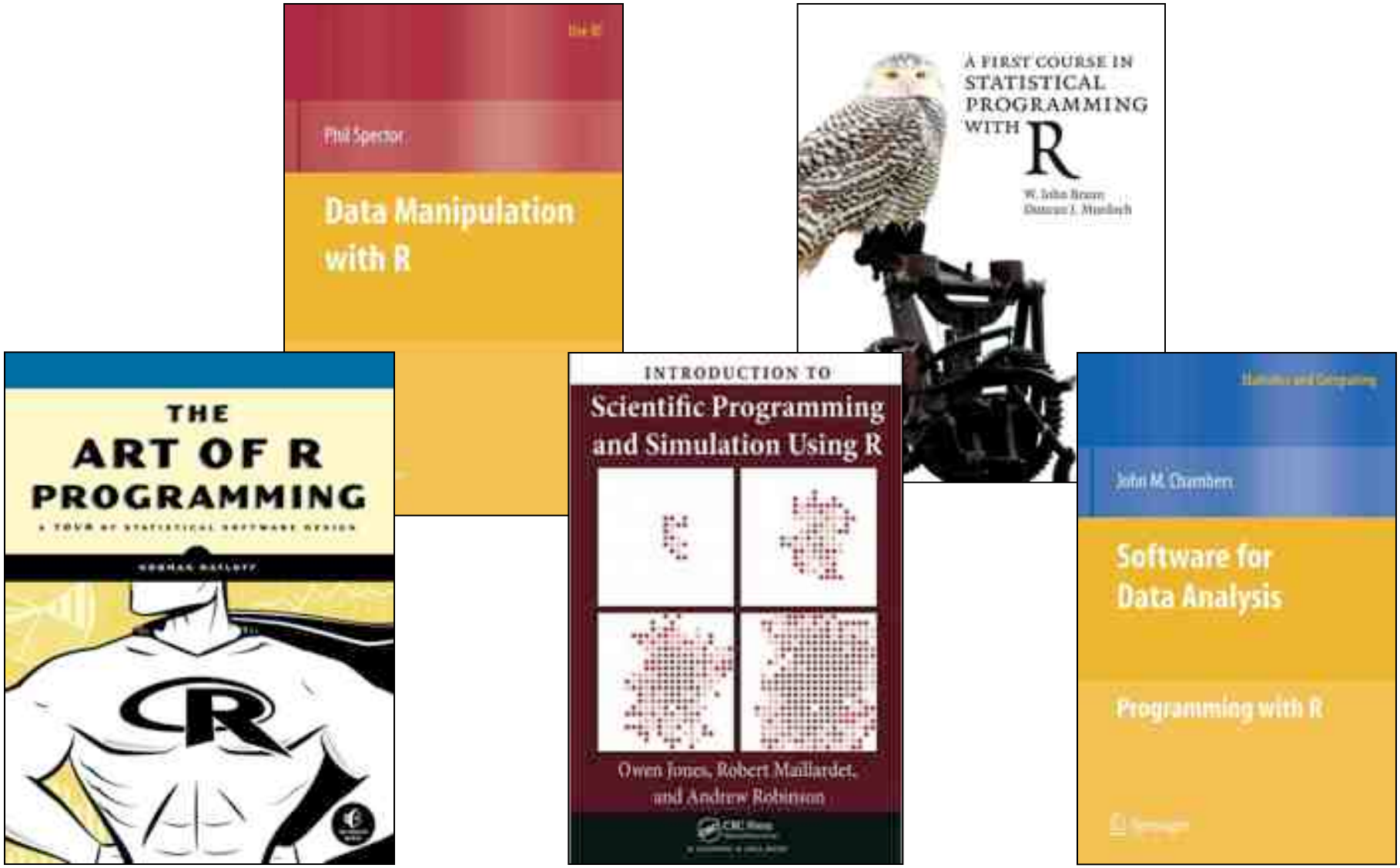
Getting Help: Standard Resources

- Built-in help:
  - `help.start` for loads of searchable help pages
  - `?` or `help` to find help on commands
  - `??` or `help.search` to search for help on topics
- The R Project:
  - http://www.r-project.org/
  - http://cran.r-project.org/doc/manuals/R-lang.html
- Mailing lists:
  - http://www.r-project.org/mail.html
  - R-help, R-SIG-Mac, and R-sig-ecology may be useful

Getting Help: Web Resources

- R reference cards:
  - http://devcheatsheet.com/tag/r/
  - http://cran.r-project.org/doc/contrib/refcard.pdf
  - http://cran.r-project.org/doc/contrib/Short-refcard.pdf
- Websites:
  - R Inferno: http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
  - R-bloggers: http://www.r-bloggers.com/
  - Calling C and Fortran: http://users.stat.umn.edu/~geyer//rc/
  - Color chart: http://research.stowers-institute.org/efg/R/Color/Chart/
  - Zoonekynd: http://zoonek2.free.fr/UNIX/48_R/02.html
  - R programming for those coming from other languages: http://www.johndcook.com/R_language_for_programmers.html

Getting Help: Books



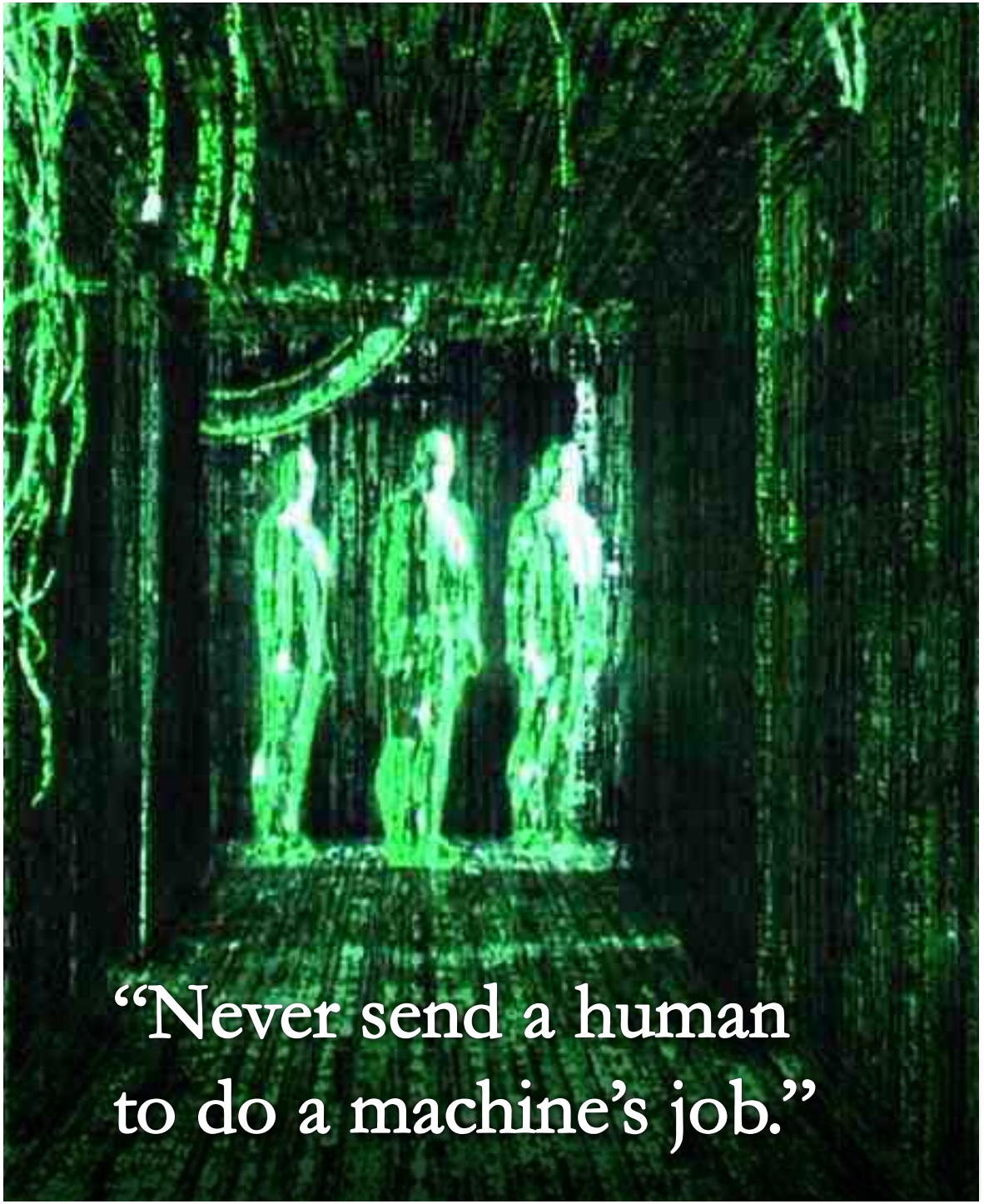
Getting Help: R Style Guides

- You should write code in a consistent style:
  - for readability
  - for sharing with others
- There are many opinions as to the best style:
  - Google's style guide:
    http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html
  - Aroma's R Coding Conventions:
    https://docs.google.com/Doc?docid=0ATuHjzgbQ0B8ZGRkenFkNTNfMjY0NTR0azd4enhj&hl=en
  - 4D Pie Charts:
    http://4dpiecharts.com/r-code-style-guide/
- Pick a readable style and stick to it

- Morpheus says:
  - Take the red pill! Don't be afraid to explore!

Exercises now online!

- Sample solutions to the exercises are now posted online:
  - http://bit.ly/yRjShO
  - https://sites.google.com/site/mcgillbgsa/workshops/programming_in_r
- I'll stick around now to work on them
- You can also email me:
  - ben dot haller [at] mail dot mcgill dot ca
- Remember, there are always many ways to solve any problem in R!



- Thanks to:
  - Jonathan Whiteley
  - Kiyoko Gotanda
  - Etienne Low-Décarie
  - Zofia Taranu
  - Corey Chivers
  - Morpheus

National Science Foundation
WHERE DISCOVERIES BEGIN

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1038597.