Overview and History of R

Computing for Data Analysis

What is R?

What is R?

What is R?

R is a dialect of the S language.

What is S?

- S is a language that was developed by John Chambers and others at Bell Labs.
- S was initiated in 1976 as an internal statistical analysis environment—originally implemented as Fortran libraries.
- Early versions of the language did not contain functions for statistical modeling.
- In 1988 the system was rewritten in C and began to resemble the system that we have today (this was Version 3 of the language). The book *Statistical Models in S* by Chambers and Hastie (the white book) documents the statistical analysis functionality.
- Version 4 of the S language was released in 1998 and is the version we use today. The book *Programming with Data* by John Chambers (the green book) documents this version of the language.

Historical Notes

- In 1993 Bell Labs gave StatSci (now Insightful Corp.) an exclusive license to develop and sell the S language.
- In 2004 Insightful purchased the S language from Lucent for $2 million and is the current owner.
- In 2006, Alcatel purchased Lucent Technologies and is now called Alcatel-Lucent.
- Insightful sells its implementation of the S language under the product name S-PLUS and has built a number of fancy features (GUIs, mostly) on top of it—hence the "PLUS".
- In 2008 Insightful is acquired by TIBCO for $25 million
- The fundamentals of the S language itself has not changed dramatically since 1998.
- In 1998, S won the Association for Computing Machinery's Software System Award.

S Philosophy

In "Stages in the Evolution of S", John Chambers writes:

> *"[W]e wanted users to be able to begin in an interactive environment, where they did not consciously think of themselves as programming. Then as their needs became clearer and their sophistication increased, they should be able to slide gradually into programming, when the language and system aspects would become more important."*

http://www.stat.bell-labs.com/S/history.html

Back to R

- 1991: Created in New Zealand by Ross Ihaka and Robert Gentleman. Their experience developing R is documented in a 1996 *JCGS* paper.
- 1993: First announcement of R to the public.
- 1995: Martin Mächler convinces Ross and Robert to use the GNU General Public License to make R free software.
- 1996: A public mailing list is created (R-help and R-devel)
- 1997: The R Core Group is formed (containing some people associated with S-PLUS). The core group controls the source code for R.
- 2000: R version 1.0.0 is released.
- 2012: R version 2.15.1 is released on June 22, 2012.

Features of R

- Syntax is very similar to S, making it easy for S-PLUS users to switch over.
- Semantics are superficially similar to S, but in reality are quite different (more on that later).
- Runs on almost any standard computing platform/OS (even on the PlayStation 3)
- Frequent releases (annual + bugfix releases); active development.

Features of R (cont'd)

- Quite lean, as far as software goes; functionality is divided into modular packages
- Graphics capabilities very sophisticated and better than most stat packages.
- Useful for interactive work, but contains a powerful programming language for developing new tools (user $\longrightarrow$ programmer)
- Very active and vibrant user community; R-help and R-devel mailing lists and Stack Overflow

Features of R (cont'd)

It's free!
(Both in the sense of beer and in the sense of speech.)

Free Software

With *free software*, you are granted

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

http://www.fsf.org

Drawbacks of R

- Essentially based on 40 year old technology.
- Little built in support for dynamic or 3-D graphics (but things have improved greatly since the "old days").
- Functionality is based on consumer demand and user contributions. If no one feels like implementing your favorite method, then it's *your* job!
  - (Or you need to pay someone to do it)
- Objects must generally be stored in physical memory; but there have been advancements to deal with this too
- Not ideal for all possible situations (but this is a drawback of all software packages).

Design of the R System

The R system is divided into 2 conceptual parts:

1. The "base" R system that you download from CRAN
2. Everything else.

R functionality is divided into a number of *packages*.

- The "base" R system contains, among other things, the **base** package which is required to run R and contains the most fundamental functions.
- The other packages contained in the "base" system include **utils**, **stats**, **datasets**, **graphics**, **grDevices**, **grid**, **methods**, **tools**, **parallel**, **compiler**, **splines**, **tcltk**, **stats4**.
- There are also "Recommend" packages: **boot**, **class**, **cluster**, **codetools**, **foreign**, **KernSmooth**, **lattice**, **mgcv**, **nlme**, **rpart**, **survival**, **MASS**, **spatial**, **nnet**, **Matrix**.

And there are many other packages available:

- There are about 4000 packages on CRAN that have been developed by users and programmers around the world.
- There are also many packages associated with the Bioconductor project (http://bioconductor.org).
- People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion.

Some R Resources

Available from CRAN (http://cran.r-project.org)

- An Introduction to R
- Writing R Extensions
- R Data Import/Export
- R Installation and Administration (mostly for building R from sources)
- R Internals (not for the faint of heart)

Some Useful Books on S/R

Standard texts

- Chambers (2008). *Software for Data Analysis*, Springer. (your textbook)
- Chambers (1998). *Programming with Data*, Springer.
- Venables & Ripley (2002). *Modern Applied Statistics with S*, Springer.
- Venables & Ripley (2000). *S Programming*, Springer.
- Pinheiro & Bates (2000). *Mixed-Effects Models in S and S-PLUS*, Springer.
- Murrell (2005). *R Graphics*, Chapman & Hall/CRC Press.

Other resources

- Springer has a series of books called *Use R!*.
- A longer list of books is at http://www.r-project.org/doc/bib/R-books.html

Introduction to the R Language

Data Types and Basic Operations

Computing for Data Analysis

Objects

R has five basic or "atomic" classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

The most basic object is a vector

- A vector can only contain objects of the same class
- BUT: The one exception is a *list*, which is represented as a vector but can contain objects of different classes (indeed, that's usually why we use them)

Empty vectors can be created with the `vector()` function.

Numbers

- Numbers in R a generally treated as numeric objects (i.e. double precision real numbers)
- If you explicitly want an integer, you need to specify the `L` suffix
- Ex: Entering `1` gives you a numeric object; entering `1L` explicitly gives you an integer.
- There is also a special number `Inf` which represents infinity; e.g. `1 / 0`; `Inf` can be used in ordinary calculations; e.g. `1 / Inf` is 0
- The value `NaN` represents an undefined value ("not a number"); e.g. 0 / 0; `NaN` can also be thought of as a missing value (more on that later)

Attributes

R objects can have attributes

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class
- length
- other user-defined attributes/metadata

Attributes of an object can be accessed using the `attributes()` function.

Entering Input

At the R prompt we type *expressions*. The <- symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
> x <-  ## Incomplete expression
```

The # character indicates a *comment*. Anything to the right of the # (including the # itself) is ignored.

Evaluation

When a complete expression is entered at the prompt, it is *evaluated* and the result of the evaluated expression is returned. The result may be *auto-printed*.

```
> x <- 5  ## nothing printed
> x       ## auto-printing occurs
[1] 5
> print(x)  ## explicit printing
[1] 5
```

The `[1]` indicates that `x` is a vector and 5 is the first element.

Printing

```
> x <- 1:20
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 16 17 18 19 20
```

The `:` operator is used to create integer sequences.

Creating Vectors

The `c()` function can be used to create vectors of objects.

```
> x <- c(0.5, 0.6)        ## numeric
> x <- c(TRUE, FALSE)     ## logical
> x <- c(T, F)            ## logical
> x <- c("a", "b", "c")   ## character
> x <- 9:29               ## integer
> x <- c(1+0i, 2+4i)      ## complex
```

Using the `vector()` function

```
> x <- vector("numeric", length = 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
```

Mixing Objects

What about the following?

```
> y <- c(1.7, "a")   ## character
> y <- c(TRUE, 2)    ## numeric
> y <- c("a", TRUE)  ## character
```

When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

Explicit Coercion

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

Explicit Coercion

Nonsensical coercion results in NAs.

```
> x <- c("a", "b", "c")
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
> as.logical(x)
[1] NA NA NA
```

Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices (cont'd)

Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices (cont'd)

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
 [1]  1  2  3  4  5  6  7  8  9 10
> dim(m) <- c(2, 5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

cbind-ing and rbind-ing

Matrices can be created by *column-binding* or *row-binding* with `cbind()` and `rbind()`.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

Factors

Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*.

- Factors are treated specially by modelling 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 better than a variable that has values 1 and 2.

Factors

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no  yes no
Levels: no yes
> table(x)
x
 no yes
  2   3
> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no"  "yes"
```

Factors

The order of the levels can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"),
              levels = c("yes", "no"))
> x
[1] yes yes no  yes no
Levels: yes no
```

Missing Values

Missing values are denoted by `NA` or `NaN` for undefined mathematical operations.

- `is.na()` is used to test objects if they are `NA`
- `is.nan()` is used to test for `NaN`
- `NA` values have a class also, so there are integer `NA`, character `NA`, etc.
- A `NaN` value is also `NA` but the converse is not true

Missing Values

```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
```

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 (just like lists); matrices must have every element be the same class
- 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()`

Data Frames

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

Names

R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("foo", "bar", "norf")
> x
 foo  bar norf
   1    2    3
> names(x)
[1] "foo"  "bar"  "norf"
```

Names

Lists can also have names.

```
> x <- list(a = 1, b = 2, c = 3)
> x
$a
[1] 1

$b
[1] 2

$c
[1] 3
```

Names

And matrices.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

Summary

Data Types

- atomic classes: numeric, logical, character, integer, complex
- vectors, lists
- factors
- missing values
- data frames
- names

Introduction to the R Language

Data Types and Basic Operations

Computing for Data Analysis

Subsetting

There are a number of operators that can be used to extract subsets of R objects.

- `[` always returns an object of the same class as the original; can be used to select more than one element (there is one exception)
- `[[` is used to extract elements of a list or a data frame; it can only be used to extract a single element and 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 hat of `[[`.

Subsetting

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]
[1] "a"
> x[2]
[1] "b"
> x[1:4]
[1] "a" "b" "c" "c"
> x[x > "a"]
[1] "b" "c" "c" "d"
> u <- x > "a"
> u
[1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"
```

Subsetting a Matrix

Matrices can be subsetted in the usual way with (i, j) type indices.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

Indices can also be missing.

```
> x[1, ]
[1] 1 3 5
> x[, 2]
[1] 3 4
```

Subsetting a Matrix

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. This behavior can be turned off by setting `drop = FALSE`.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3

> x[1, 2, drop = FALSE]
     [,1]
[1,]    3
```

Subsetting a Matrix

Similarly, subsetting a single column or a single row will give you a vector, not a matrix (by default).

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5
> x[1, , drop = FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

Subsetting Lists

```
> x <- list(foo = 1:4, bar = 0.6)
> x[1]
$foo
[1] 1 2 3 4

> x[[1]]
[1] 1 2 3 4

> x$bar
[1] 0.6
> x[["bar"]]
[1] 0.6
> x["bar"]
$bar
[1] 0.6
```

Subsetting Lists

Extracting multiple elements of a list.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> x[c(1, 3)]
$foo
[1] 1 2 3 4

$baz
[1] "hello"
```

Subsetting Lists

The [[operator can be used with *computed* indices; $ can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
> x[[name]]  ## computed index for 'foo'
[1] 1 2 3 4
> x$name     ## element 'name' doesn't exist!
NULL
> x$foo
[1] 1 2 3 4  ## element 'foo' does exist
```

Subsetting Nested Elements of a List

The [[can take an integer sequence.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
> x[[c(1, 3)]]
[1] 14
> x[[1]][[3]]
[1] 14

> x[[c(2, 1)]]
[1] 3.14
```

Partial Matching

Partial matching of names is allowed with `[[` and `$`.

```
> x <- list(aardvark = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]]
[1] 1 2 3 4 5
```

Removing NA Values

A common task is to remove missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> x[!bad]
[1] 1 2 4 5
```

Removing NA Values

What if there are multiple things and you want to take the subset 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
[1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE
> x[good]
[1] 1 2 4 5
> y[good]
[1] "a" "b" "d" "f"
```

Removing NA Values

```
> airquality[1:6, ]
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> good <- complete.cases(airquality)
> airquality[good, ][1:6, ]
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
7    23     299  8.6   65     5   7
```

Introduction to the R Language

Vectorized Operations

Computing for Data Analysis

Vectorized Operations

Many operations in R are *vectorized* making code more efficient, concise, and easier to read.

```
> x <- 1:4; y <- 6:9
> x + y
[1]  7  9 11 13
> x > 2
[1] FALSE FALSE  TRUE  TRUE
> x >= 2
[1] FALSE  TRUE  TRUE  TRUE
> y == 8
[1] FALSE FALSE  TRUE FALSE
> x * y
[1]  6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Vectorized Matrix Operations

```
> x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
> x * y       ## element-wise multiplication
     [,1] [,2]
[1,]   10   30
[2,]   20   40
> x / y
     [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
> x %*% y     ## true matrix multiplication
     [,1] [,2]
[1,]   40   40
[2,]   60   60
```

Introduction to the R Language

Reading and Writing Data

Computing for Data Analysis

Reading Data

There are a few principal functions 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
- `unserialize`, for reading single R objects in binary form

Writing Data

There are analogous functions for writing data to files

- `write.table`
- `writeLines`
- `dump`
- `dput`
- `save`
- `serialize`

Reading Data Files with read.table

The `read.table` function is one of the most commonly used functions for reading data. It has a few important arguments:

- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line
- `sep`, a string indicating 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?

read.table

For small to moderately 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 a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table

Telling R all these things directly makes R run faster and more efficiently.

- `read.csv` is identical to `read.table` except that the default separator is a comma.

Reading in Larger Datasets with read.table

With much larger datasets, doing the following things will make your life easier and will prevent R from choking.

- Read the help page for `read.table`, which contains many hints
- Make a rough calculation of the memory required to store your dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
- Set `comment.char = ""` if there are no commented lines in your file.

Reading in Larger Datasets with read.table

- Use the `colClasses` argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set `colClasses = "numeric"`. A quick an dirty way to figure out the classes of each column is the following:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt",
                     colClasses = classes)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool `wc` to calculate the number of lines in a file.

Know Thy System

In general, when using R with larger datasets, it's useful to know a few things about your system.

- How much memory is available?
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system?
- Is the OS 32 or 64 bit?

Calculating Memory Requirements

I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame?

$$
\begin{aligned}
1,500,000 \times 120 \times 8 \text{ bytes/numeric} &= 1440000000 \text{ bytes} \\
&= 1440000000/2^{20} \text{ bytes/MB} \\
&= 1,373.29 \text{ MB} \\
&= 1.34 \text{ GB}
\end{aligned}
$$

Textual Formats

- `dumping` and `dputing` are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- Unlike writing out a table or csv file, `dump` and `dput` preserve the *metadata* (sacrificing some readability), so that another user doesn't have to specify it all over again.
- Textual formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem
- Textual formats adhere to the "Unix philosophy"
- Downside: The format is not very space-efficient

dput-ting R Objects

Another way to pass data around is by deparsing the R object with dput and reading it back in using dget.

```
> y <- data.frame(a = 1, b = "a")
> dput(y)
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")
> new.y
  a b
1 1 a
```

Dumping R Objects

Multiple objects can be deparsed using the `dump` function and read back in using `source`.

```
> x <- "foo"
> y <- data.frame(a = 1, b = "a")
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
> source("data.R")
> y
  a b
1 1 a
> x
[1] "foo"
```

Interfaces to the Outside World

Data are read in using *connection* interfaces. Connections can be made to files (most common) or to other more exotic things.

- `file`, opens a connection to a file
- `gzfile`, opens a connection to a file compressed with gzip
- `bzfile`, opens a connection to a file compressed with bzip2
- `url`, opens a connection to a webpage

File Connections

```
> str(file)
function (description = "", open = "", blocking = TRUE,
          encoding = getOption("encoding"))
```

- `description` is the name of the file
- open is a code indicating
  - "r" read only
  - "w" writing (and initializing a new file)
  - "a" appending
  - "rb", "wb", "ab" reading, writing, or appending in binary mode (Windows)

Connections

In general, connections are powerful tools that let you navigate files or other external objects. In practice, we often don't need to deal with the connection interface directly.

```
con <- file("foo.txt", "r")
data <- read.csv(con)
close(con)
```

is the same as

```
data <- read.csv("foo.txt")
```

Reading Lines of a Text File

The `readLines` function can be used to simply read lines of a text file and store them in a character vector.

```
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
 [1] "1080"      "10-point" "10th"      "11-point"
 [5] "12-point" "16-point" "18-point" "1st"
 [9] "2"         "20-point"
```

`writeLines` takes a character vector and writes each element one line at a time to a text file.

Reading Lines of a Text File

readLines can be useful for reading in lines of webpages

```
## This might take time
con <- url("http://www.jhsph.edu", "r")
x <- readLines(con)
> head(x)
[1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">"
[2] ""
[3] "<html>"
[4] "<head>"
[5] "\t<meta http-equiv=\"Content-Type\" content=\"text/html;charset=utf-8
```

Introduction to the R Language

Control Structures

Computing for Data Analysis

Control Structures

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are

- `if`, `else`: testing a condition
- `for`: execute a loop a fixed number of times
- `while`: execute a loop *while* a condition is true
- `repeat`: execute an infinite loop
- `break`: break the execution of a loop
- `next`: skip an interation of a loop
- `return`: exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expresisons.

Control Structures: if

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

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

if

This is a valid if/else structure.

```
if(x > 3) {
        y <- 10
} else {
        y <- 0
}
```

So is this one.

```
y <- if(x > 3) {
        10
} else {
        0
}
```

Of course, the `else` clause is not necessary.

```
if(<condition1>) {

}

if(<condition2>) {

}
```

for

`for` loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

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

This loop takes the `i` variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, and then exits.

for

These three loops have the same behavior.

```
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)
}

for(i in 1:4) print(x[i])
```

Nested for loops

`for` loops can be nested.

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

Be careful with nesting though. Nesting beyond 2–3 levels is often very difficult to read/understand.

while

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

```
count <- 0

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

While loops can potentially result in infinite loops if not written properly. Use with care!

while

Sometimes there will be more than one condition in the test.

```
z <- 5

while(z >= 3 && z <= 10) {
        print(z)
        coin <- rbinom(1, 1, 0.5)

        if(coin == 1) {  ## random walk
                z <- z + 1
        } else {
                z <- z - 1
        }
}
```

Conditions are always evaluated from left to right.

repeat

Repeat initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a `repeat` loop is to call `break`.

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

repeat {
        x1 <- computeEstimate()

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

The loop in the previous slide is a bit dangerous because there's no guarantee it will stop. Better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence was achieved or not.

next, return

`next` is used to skip an iteration of a loop

```
for(i in 1:100) {
        if(i <= 20) {
                ## Skip the first 20 iterations
                next
        }
        ## Do something here
}
```

`return` signals that a function should exit and return a given value

Summary

- Control structures like `if`, `while`, and `for` allow you to control the flow of an R program
- Infinite loops should generally be avoided, even if they are theoretically correct.
- Control structures mentiond here are primarily useful for writing programs; for command-line interactive work, the *apply functions are more useful.

Introduction to the R Language

Functions

Computing for Data Analysis

Functions

Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```
f <- function(<arguments>) {
        ## Do something interesting
}
```

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function

The return value of a function is the last expression in the function body to be evaluated.

Function Arguments

Functions have *named arguments* which potentially have *default values*.

- The *formal arguments* are the arguments included in the function definition
- The `formals` function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be *missing* or might have default values

Argument Matching

R functions arguments can be matched positionally or by name. So the following calls to sd are all equivalent

```
> mydata <- rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

Argument Matching

You can mix positional matching with matching by name. When an 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.

```
> args(lm)
function (formula, data, subset, weights, na.action,
          method = "qr", model = TRUE, x = FALSE,
          y = FALSE, qr = TRUE, singular.ok = TRUE,
          contrasts = NULL, offset, ...)
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

Argument Matching

- Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
- Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Argument Matching

Function arguments can also be *partially* matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Defining a Function

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

}
```

In addition to not specifying a default value, you can also set an argument value to `NULL`.

Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed.

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

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the 2 gets positionally matched to `a`.

Lazy Evaluation

Another example

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

```
> f(45)
[1] 45
Error in print(b) : argument "b" is missing, with no default
>
```

Notice that "45" got printed first before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` it had to throw an error.

The "..." Argument

The `...` argument indicate a variable number of arguments that are usually passed on to other functions.

- `...` is often used when extending another function and you don't want to copy the entire argument list of the original function

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

- Generic functions use `...` so that extra arguments can be passed to methods (more on this later).

```
> mean
function (x, ...)
UseMethod("mean")
```

The "..." Argument

The `...` argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> args(cat)
function (..., file = "", sep = " ", fill = FALSE,
    labels = NULL, append = FALSE)
```

Arguments Coming After the "..." Argument

One catch with `...` is that any arguments that appear *after* `...` on the argument list must be named explicitly and cannot be partially matched.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> paste("a", "b", sep = ":")
[1] "a:b"

> paste("a", "b", se = ":")
[1] "a b :"
```

A Diversion on Binding Values to Symbol

How does R know which value to assign to which symbol? When I type

```
> lm <- function(x) { x * x }
> lm
function(x) { x * x }
```

how does R know what value to assign to the symbol `lm`? Why doesn't it give it the value of `lm` that is in the **stats** package?

A Diversion on Binding Values to Symbol

When R tries to bind a value to a symbol, it searches through a series of `environments` to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly

1. Search the global environment for a symbol name matching the one requested.
2. Search the namespaces of each of the packages on the search list

The search list can be found by using the `search` function.

```
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Binding Values to Symbol

- The *global environment* or the user's workspace is always the first element of the search list and the **base** package is always the last.
- The order of the packages on the search list matters!
- User's can configure which packages get loaded on startup so you cannot assume that there will be a set list of packages available.
- When a user loads a package with `library` the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.
- Note that R has separate namespaces for functions and non-functions so it's possible to have an object named `c` and a function named `c`.

Scoping Rules

The scoping rules for R are the main feature that make it different from the original S language.

- The scoping rules determine how a value is associated with a free variable in a function
- R uses *lexical scoping* or *static scoping*. A common alternative is *dynamic scoping*.
- Related to the scoping rules is how R uses the *search list* to bind a value to a symbol
- Lexical scoping turns out to be particularly useful for simplifying statistical computations

Lexical Scoping

Consider the following function.

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

This function has 2 formal arguments `x` and `y`. In the body of the function there is another symbol `z`. In this case `z` is called a *free variable*.
The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned insided the function body).

Lexical Scoping

Lexical scoping in R means that

> *the values of free variables are searched for in the environment in which the function was defined.*

What is an environment?

- An *environment* is a collection of (symbol, value) pairs, i.e. `x` is a symbol and `3.14` might be its value.
- Every environment has a parent environment; it is possible for an environment to have multiple "children"
- the only environment without a parent is the empty environment
- A function + an environment = a *closure* or *function closure*.

Lexical Scoping

Searching for the value for a free variable:

- If the value of a symbol is not found in the environment in which a function was defined, then 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 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, then an error is thrown.

Lexical Scoping

Why does all this matter?

- Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace
- This behavior is logical for most people and is usually the "right thing" to do
- However, in R you can have functions defined *inside other functions*
  - Languages like C don't let you do this
- Now things get interesting — In this case the environment in which a function is defined is the body of another function!

Lexical Scoping

```
make.power <- function(n) {
        pow <- function(x) {
                x^n
        }
        pow
}
```

This function returns another function as its value.

```
> cube <- make.power(3)
> square <- make.power(2)
> cube(3)
[1] 27
> square(3)
[1] 9
```

Exploring a Function Closure

What's in a function's environment?

```
> ls(environment(cube))
[1] "n"   "pow"
> get("n", environment(cube))
[1] 3

> ls(environment(square))
[1] "n"   "pow"
> get("n", environment(square))
[1] 2
```

Lexical vs. Dynamic Scoping

```
y <- 10

f <- function(x) {
        y <- 2
        y^2 + g(x)
}

g <- function(x) {
        x * y
}
```

What is the value of

```
f(3)
```

Lexical vs. Dynamic Scoping

- With lexical scoping the value of `y` in the function `g` is looked up in the environment in which the function was defined, in this case the global environment, so the value of `y` is 10.
- With dynamic scoping, the value of `y` is looked up in the environment from which the function was *called* (sometimes referred to as the *calling environment*).
  - In R the calling environment is known as the *parent frame*

  So the value of `y` would be 2.

Lexical vs. Dynamic Scoping

When a function is *defined* in the global environment and is subsequently *called* from the global environment, then the defining environment and the calling environment are the same. This can sometimes give the appearance of dynamic scoping.

```
> g <- function(x) {
+         a <- 3
+         x + a + y
+ }
> g(2)
Error in g(2) : object "y" not found
> y <- 3
> g(2)
[1] 8
```

Other Languages

Other languages that support lexical scoping

- Scheme
- Perl
- Python
- Common Lisp (all languages converge to Lisp)

Consequences of Lexical Scoping

- In R, all objects must be stored in memory
- All functions must carry a pointer to their respective defining environments, which could be anywhere
- In S-PLUS, free variables are always looked up in the global workspace, so everything can be stored on the disk because the "defining environment" of all functions is the same.

Application: Optimization

Why is any of this information useful?

- Optimization routines in R like `optim`, `nlm`, and `optimize` require you to pass a function whose argument is a vector of parameters (e.g. a log-likelihood)
- However, an object function might depend on a host of other things besides its parameters (like *data*)
- When writing software which does optimization, it may be desirable to allow the user to hold certain parameters fixed

Maximizing a Normal Likelihood

Write a "constructor" function

```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {
        params <- fixed
        function(p) {
                params[!fixed] <- p
                mu <- params[1]
                sigma <- params[2]
                a <- -0.5*length(data)*log(2*pi*sigma^2)
                b <- -0.5*sum((data-mu)^2) / (sigma^2)
                -(a + b)
        }
}
```

Note: Optimization functions in R *minimize* functions, so you need to use the negative log-likelihood.

Maximizing a Normal Likelihood

```
> set.seed(1); normals <- rnorm(100, 1, 2)
> nLL <- make.NegLogLik(normals)
> nLL
function(p) {
                params[!fixed] <- p
                mu <- params[1]
                sigma <- params[2]
                a <- -0.5*length(data)*log(2*pi*sigma^2)
                b <- -0.5*sum((data-mu)^2) / (sigma^2)
                -(a + b)
        }
<environment: 0x165b1a4>
> ls(environment(nLL))
[1] "data"   "fixed"  "params"
```

Estimating Parameters

```
> optim(c(mu = 0, sigma = 1), nLL)$par
      mu    sigma
1.218239 1.787343
```

Fixing $\sigma = 2$

```
> nLL <- make.NegLogLik(normals, c(FALSE, 2))
> optimize(nLL, c(-1, 3))$minimum
[1] 1.217775
```

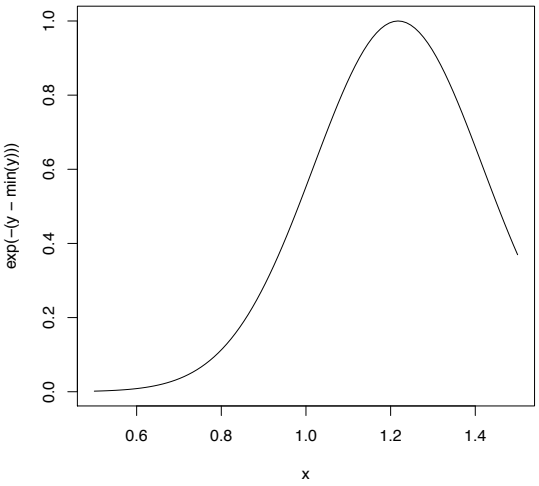
Fixing $\mu = 1$

```
> nLL <- make.NegLogLik(normals, c(1, FALSE))
> optimize(nLL, c(1e-6, 10))$minimum
[1] 1.800596
```

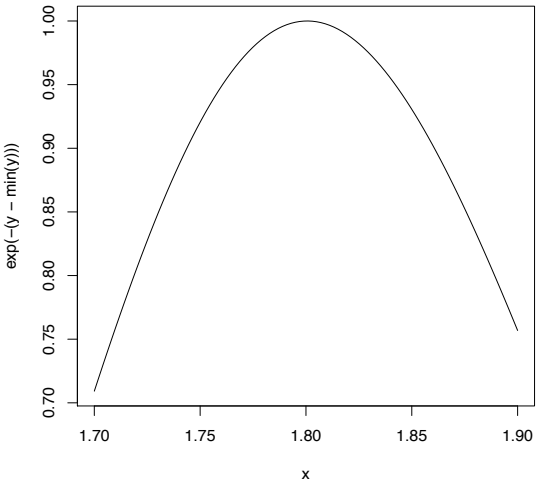
Plotting the Likelihood

```
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")

nLL <- make.NegLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")
```

Plotting the Likelihood



Plotting the Likelihood



Lexical Scoping Summary

- Objective functions can be "built" which contain all of the necessary data for evaluating the function
- No need to carry around long argument lists — useful for interactive and exploratory work.
- Code can be simplified and cleand up
- Reference: Robert Gentleman and Ross Ihaka (2000). "Lexical Scope and Statistical Computing," *JCGS*, 9, 491–508.

Application: Optimization

Why is any of this information useful?

- Optimization routines in R like `optim`, `nlm`, and `optimize` require you to pass a function whose argument is a vector of parameters (e.g. a log-likelihood)
- However, an object function might depend on a host of other things besides its parameters (like *data*)
- When writing software which does optimization, it may be desirable to allow the user to hold certain parameters fixed

Maximizing a Normal Likelihood

Write a "constructor" function

```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {
        params <- fixed
        function(p) {
                params[!fixed] <- p
                mu <- params[1]
                sigma <- params[2]
                a <- -0.5*length(data)*log(2*pi*sigma^2)
                b <- -0.5*sum((data-mu)^2) / (sigma^2)
                -(a + b)
        }
}
```

Note: Optimization functions in R *minimize* functions, so you need to use the negative log-likelihood.

Maximizing a Normal Likelihood

```
> set.seed(1); normals <- rnorm(100, 1, 2)
> nLL <- make.NegLogLik(normals)
> nLL
function(p) {
                params[!fixed] <- p
                mu <- params[1]
                sigma <- params[2]
                a <- -0.5*length(data)*log(2*pi*sigma^2)
                b <- -0.5*sum((data-mu)^2) / (sigma^2)
                -(a + b)
        }
<environment: 0x165b1a4>
> ls(environment(nLL))
[1] "data"   "fixed"  "params"
```

Estimating Parameters

```
> optim(c(mu = 0, sigma = 1), nLL)$par
      mu    sigma
1.218239 1.787343
```

Fixing $\sigma = 2$

```
> nLL <- make.NegLogLik(normals, c(FALSE, 2))
> optimize(nLL, c(-1, 3))$minimum
[1] 1.217775
```

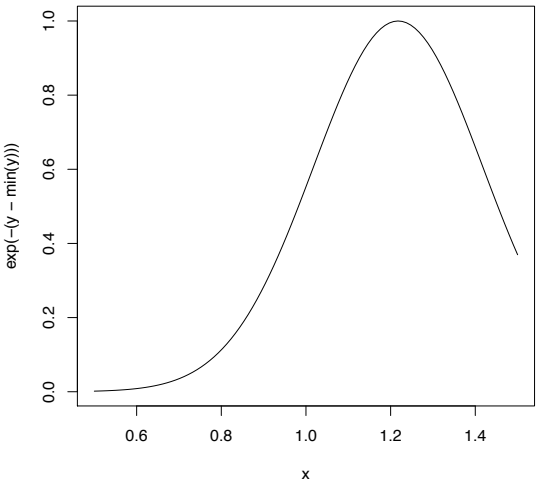
Fixing $\mu = 1$

```
> nLL <- make.NegLogLik(normals, c(1, FALSE))
> optimize(nLL, c(1e-6, 10))$minimum
[1] 1.800596
```

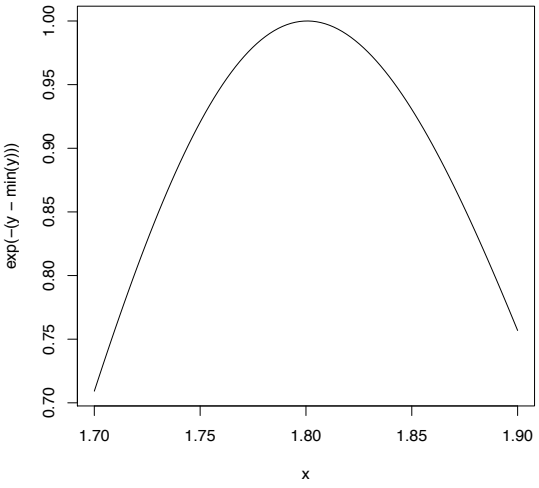
Plotting the Likelihood

```
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")

nLL <- make.NegLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")
```

Plotting the Likelihood



Plotting the Likelihood



Lexical Scoping Summary

- Objective functions can be "built" which contain all of the necessary data for evaluating the function
- No need to carry around long argument lists — useful for interactive and exploratory work.
- Code can be simplified and cleand up
- Reference: Robert Gentleman and Ross Ihaka (2000). "Lexical Scope and Statistical Computing," *JCGS*, 9, 491–508.

Introduction to the R Language

Loop Functions

Computing for Data Analysis

Looping on the Command Line

Writing for, while loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier.

- `lapply`: Loop over a list and evaluate a function on each element
- `sapply`: Same as `lapply` but try to simplify the result
- `apply`: Apply a function over the margins of an array
- `tapply`: Apply a function over subsets of a vector
- `mapply`: Multivariate version of `lapply`

An auxiliary function `split` is also useful, particularly in conjunction with `lapply`.

lapply

`lapply` takes three arguments: a list `X`, a function (or the name of a function) `FUN`, and other arguments via its `...` argument. If `X` is not a list, it will be coerced to a list using `as.list`.

```
> lapply
function (X, FUN, ...)
{
    FUN <- match.fun(FUN)
    if (!is.vector(X) || is.object(X))
        X <- as.list(X)
    .Internal(lapply(X, FUN))
}
```

The actual looping is done internally in C code.

lapply

`lapply` always returns a list, regardless of the class of the input.

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

$b
[1] 0.0296824
```

lapply

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.06082667

$c
[1] 1.467083

$d
[1] 5.074749
```

lapply

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.2675082

[[2]]
[1] 0.2186453 0.5167968

[[3]]
[1] 0.2689506 0.1811683 0.5185761

[[4]]
[1] 0.5627829 0.1291569 0.2563676 0.7179353
```

lapply

```
> x <- 1:4
> 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

[[4]]
[1] 1.195114 3.594027 2.930794 2.766946
```

lapply

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

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

lapply

An anonymous function for extracting the first column of each matrix.

```
> lapply(x, function(elt) elt[,1])
$a
[1] 1 2

$b
[1] 1 2 3
```

sapply

`sapply` will try to simplify the result of `lapply` if possible.

- If the result is a list where every element is length 1, then a vector is returned
- 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

sapply

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.06082667

$c
[1] 1.467083

$d
[1] 5.074749
```

sapply

```
> sapply(x, mean)
         a          b          c          d
2.50000000 0.06082667 1.46708277 5.07474950
```

```
> mean(x)
[1] NA
Warning message:
In mean.default(x) : argument is not numeric or logical: returning NA
```

apply

`apply` is used to a evaluate a function (often an anonymous one) over the margins of an array.

- It is most often used to apply a function to the rows or 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 it works in one line!

apply

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- `X` is an array
- `MARGIN` is an integer vector indicating which margins should be "retained".
- `FUN` is a function to be applied
- `...` is for other arguments to be passed to `FUN`

apply

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
 [1]  0.04868268  0.35743615 -0.09104379
 [4] -0.05381370 -0.16552070 -0.18192493
 [7]  0.10285727  0.36519270  0.14898850
[10]  0.26767260

> apply(x, 1, sum)
 [1] -1.94843314  2.60601195  1.51772391
 [4] -2.80386816  3.73728682 -1.69371360
 [7]  0.02359932  3.91874808 -2.39902859
[10]  0.48685925 -1.77576824 -3.34016277
[13]  4.04101009  0.46515429  1.83687755
[16]  4.36744690  2.21993789  2.60983764
[19] -1.48607630  3.58709251
```

col/row sums and means

For sums and means of matrix dimensions, we have some shortcuts.

- `rowSums` = apply(x, 1, sum)
- `rowMeans` = apply(x, 1, mean)
- `colSums` = apply(x, 2, sum)
- `colMeans` = apply(x, 2, mean)

The shortcut functions are *much* faster, but you won't notice unless you're using a large matrix.

Other Ways to Apply

Quantiles of the rows of a matrix.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
           [,1]        [,2]       [,3]        [,4]
25% -0.3304284 -0.99812467 -0.9186279 -0.49711686
75%  0.9258157  0.07065724  0.3050407 -0.06585436
           [,5]       [,6]      [,7]       [,8]
25% -0.05999553 -0.6588380 -0.653250 0.01749997
75%  0.52928743  0.3727449  1.255089 0.72318419
          [,9]      [,10]      [,11]      [,12]
25% -1.2467955 -0.8378429 -1.0488430 -0.7054902
75%  0.3352377  0.7297176  0.3113434  0.4581150
         [,13]      [,14]      [,15]      [,16]
25% -0.1895108 -0.5729407 -0.5968578 -0.9517069
75%  0.5326299  0.5064267  0.4933852  0.8868922
         [,17]      [,18]      [,19]      [,20]
```

Average matrix in an array

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
           [,1]        [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908

> rowMeans(a, dims = 2)
           [,1]        [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908
```

tapply

`tapply` is used to apply a function over subsets of a vector. I don't know why it's called `tapply`.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

- `X` is a vector
- `INDEX` is a factor or a list of factors (or else they are coerced to factors)
- `FUN` is a function to be applied
- `...` contains other arguments to be passed `FUN`
- `simplify`, should we simplify the result?

tapply

Take group means.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3
[24] 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
        1         2         3
0.1144464 0.5163468 1.2463678
```

tapply

Take group means without simplification.

```
> tapply(x, f, mean, simplify = FALSE)
$`1`
[1] 0.1144464

$`2`
[1] 0.5163468

$`3`
[1] 1.246368
```

tapply

Find group ranges.

```
> tapply(x, f, range)
$`1`
[1] -1.097309  2.694970

$`2`
[1] 0.09479023 0.79107293

$`3`
[1] 0.4717443 2.5887025
```

split

`split` takes a vector or other objects and splits it into groups determined by a factor or list of factors.

```
> str(split)
function (x, f, drop = FALSE, ...)
```

- `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 factors levels should be dropped

split

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$`1`
 [1] -0.8493038 -0.5699717 -0.8385255 -0.8842019
 [5]  0.2849881  0.9383361 -1.0973089  2.6949703
 [9]  1.5976789 -0.1321970

$`2`
 [1] 0.09479023 0.79107293 0.45857419 0.74849293
 [5] 0.34936491 0.35842084 0.78541705 0.57732081
 [9] 0.46817559 0.53183823

$`3`
 [1] 0.6795651 0.9293171 1.0318103 0.4717443
 [5] 2.5887025 1.5975774 1.3246333 1.4372701
```

split

A common idiom is `split` followed by an `lapply`.

```
> lapply(split(x, f), mean)
$`1`
[1] 0.1144464

$`2`
[1] 0.5163468

$`3`
[1] 1.246368
```

Splitting a Data Frame

```
> library(datasets)
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

Splitting a Data Frame

```
> s <- split(airquality, airquality$Month)
> lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
$`5`
   Ozone  Solar.R     Wind
      NA       NA 11.62258

$`6`
    Ozone   Solar.R      Wind
       NA 190.16667  10.26667

$`7`
     Ozone    Solar.R       Wind
        NA 216.483871   8.941935

$`8`
   Ozone  Solar.R     Wind
```

Splitting a Data Frame

```
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
               5         6          7        8        9
Ozone         NA        NA         NA       NA       NA
Solar.R       NA 190.16667 216.483871       NA 167.4333
Wind    11.62258  10.26667   8.941935 8.793548  10.1800
```

```
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")],
                                 na.rm = TRUE))
               5         6          7          8         9
Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

Splitting on More than One Level

```
> x <- rnorm(10)
> f1 <- gl(2, 5)
> f2 <- gl(5, 2)
> f1
 [1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
> f2
 [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> 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 2.1 1.2 2.2 1.3 2.3 1.4 ... 2.5
```

Splitting on More than One Level

Interactions can create empty levels.

```
> str(split(x, list(f1, f2)))
List of 10
 $ 1.1: num [1:2] -0.378  0.445
 $ 2.1: num(0)
 $ 1.2: num [1:2] 1.4066 0.0166
 $ 2.2: num(0)
 $ 1.3: num -0.355
 $ 2.3: num 0.315
 $ 1.4: num(0)
 $ 2.4: num [1:2] -0.907  0.723
 $ 1.5: num(0)
 $ 2.5: num [1:2] 0.732 0.360
```

Empty levels can be dropped.

```
> str(split(x, list(f1, f2), drop = TRUE))
List of 6
 $ 1.1: num [1:2] -0.378  0.445
 $ 1.2: num [1:2] 1.4066 0.0166
 $ 1.3: num -0.355
 $ 2.3: num 0.315
 $ 2.4: num [1:2] -0.907  0.723
 $ 2.5: num [1:2] 0.732 0.360
```

mapply

`mapply` is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
          USE.NAMES = TRUE)
```

- `FUN` is a function to apply
- `...` contains arguments to apply over
- `MoreArgs` is a list of other arguments to `FUN`.
- `SIMPLIFY` indicates whether the result should be simplified

mapply

The following is tedious to type

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

Instead we can do

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

Vectorizing a Function

```
> noise <- function(n, mean, sd) {
+         rnorm(n, mean, sd)
+ }
> noise(5, 1, 2)
[1]  2.4831198  2.4790100  0.4855190 -1.2117759
[5] -0.2743532

> noise(1:5, 1:5, 2)
[1] -4.2128648 -0.3989266  4.2507057  1.1572738
[5]  3.7413584
```

Instant Vectorization

```
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 1.037658

[[2]]
[1] 0.7113482 2.7555797

[[3]]
[1] 2.769527 1.643568 4.597882

[[4]]
[1] 4.476741 5.658653 3.962813 1.204284

[[5]]
[1] 4.797123 6.314616 4.969892 6.530432 6.723254
```

Instant Vectorization

Which is the same as

```
list(noise(1, 1, 2), noise(2, 2, 2),
     noise(3, 3, 2), noise(4, 4, 2),
     noise(5, 5, 2))
```

Debugging

Computing for Data Analysis

Something's Wrong!

Indications that something's not right

- `message`: A generic notification/diagnostic message produced by the `message` function; execution of the function continues
- `warning`: An indication that something is wrong but not necessarily fatal; execution of the function continues; generated by the `warning` function
- `error`: An indication that a fatal problem has occurred; execution stops; produced by the `stop` function
- `condition`: A generic concept for indicating that something unexpected can occur; programmers can create their own conditions

Something's Wrong!

Warning

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

Something's Wrong

```
printmessage <- function(x) {
        if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than or equal to zero")
        invisible(x)
}
```

Something's Wrong

```
printmessage <- function(x) {
        if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than or equal to zero")
        invisible(x)
}
> printmessage(1)
[1] "x is greater than zero"
> printmessage(NA)
Error in if (x > 0) { : missing value where TRUE/FALSE needed
```

Something's Wrong!

```
printmessage2 <- function(x) {
        if(is.na(x))
                print("x is a missing value!")
        else if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than or equal to zero")
        invisible(x)
}
```

Something's Wrong!

```
printmessage2 <- function(x) {
        if(is.na(x))
                print("x is a missing value!")
        else if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than or equal to zero")
        invisible(x)
}
> x <- log(-1)
Warning message:
In log(-1) : NaNs produced
> printmessage2(x)
[1] "x is a missing value!"
```

Something's Wrong!

How do you know that something is wrong with your function?

- What was your input? How did you call the function?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does what you get differ from what you were expecting?
- Were your expectations correct in the first place?
- Can you reproduce the problem (exactly)?

Debugging Tools in R

The primary tools for debugging functions in R are

- `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 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 a specific places
- `recover`: allows you to modify the error behavior so that you can browse the function call stack

These are interactive tools specifically designed to allow you to pick through a function. There's also the more blunt technique of inserting `print`/`cat` statements in the function.

traceback

```
> mean(x)
Error in mean(x) : object 'x' not found
> traceback()
1: mean(x)
>
```

traceback

```
> lm(y ~ x)
Error in eval(expr, envir, enclos) : object 'y' not found
> traceback()
7: eval(expr, envir, enclos)
6: eval(predvars, data, env)
5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)
4: model.frame(formula = y ~ x, drop.unused.levels = TRUE)
3: eval(expr, envir, enclos)
2: eval(mf, parent.frame())
1: lm(y ~ x)
```

debug

```
> debug(lm)
> lm(y ~ x)
debugging in: lm(y ~ x)
debug: {
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    ...
    if (!qr)
        z$qr <- NULL
    z
}
Browse[2]>
```

debug

```
Browse[2]> n
debug: ret.x <- x
Browse[2]> n
debug: ret.y <- y
Browse[2]> n
debug: cl <- match.call()
Browse[2]> n
debug: mf <- match.call(expand.dots = FALSE)
Browse[2]> n
debug: m <- match(c("formula", "data", "subset", "weights", "na.action",
    "offset"), names(mf), 0L)
```

recover

```
> options(error = recover)
> read.csv("nosuchfile")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'nosuchfile': No such file or directory

Enter a frame number, or 0 to exit

1: read.csv("nosuchfile")
2: read.table(file = file, header = header, sep = sep, quote = quote, dec =
3: file(file, "rt")

Selection:
```

Summary

- There are three main indications of a problem/condition: message, warning, error; only an error is fatal
- When analyzing a function with a problem, make sure you can reproduce the problem, clearly state your expectations and how the output differs from your expectation
- Interactive debugging tools `traceback`, `debug`, `browser`, `trace`, and `recover` can be used to find problematic code in functions
- Debugging tools are not a substitute for thinking!

Simulation

Computing for Data Analysis

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/SD) 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

Generating Random Numbers

Probability distribution functions usually have four functions associated with them. The functions are prefixed with a

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

Generating Random Numbers

Working with the Normal distributions requires using these four functions

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

If Φ is the cumulative distribution function for a standard Normal distribution, then `pnorm`$(q) = \Phi(q)$ and `qnorm`$(\mathrm{p}) = \Phi^{-1}(p)$.

Generating Random Numbers

Generating random Normal variates

```
> x <- rnorm(10)
> x
 [1] 1.38380206 0.48772671 0.53403109 0.66721944
 [5] 0.01585029 0.37945986 1.31096736 0.55330472
 [9] 1.22090852 0.45236742
> x <- rnorm(10, 20, 2)
> x
 [1] 23.38812 20.16846 21.87999 20.73813 19.59020
 [6] 18.73439 18.31721 22.51748 20.36966 21.04371
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.32   19.73   20.55   20.67   21.67   23.39
```

Generating Random Numbers

Setting the random number seed with `set.seed` ensures reproducibility

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814
[5] -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
```

Always set the random number seed when conducting a simulation!

Generating Random Numbers

Generating Poisson data

```
> rpois(10, 1)
 [1] 3 1 0 1 0 0 1 0 1 1
> rpois(10, 2)
 [1] 6 2 2 1 3 2 2 1 1 2
> rpois(10, 20)
 [1] 20 11 21 20 20 21 17 15 24 20

> ppois(2, 2)  ## Cumulative distribution
[1] 0.6766764  ## Pr(x <= 2)
> ppois(4, 2)
[1] 0.947347   ## Pr(x <= 4)
> ppois(6, 2)
[1] 0.9954662  ## Pr(x <= 6)
```

Generating Random Numbers From a Linear Model

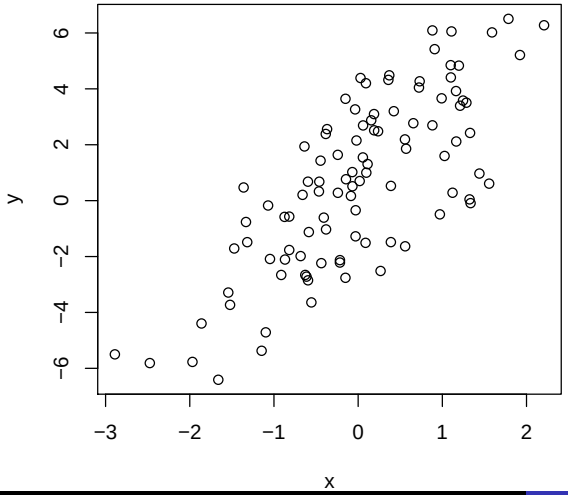
Suppose we want to simulate from the following linear model

$$y = \beta_0 + \beta_1 x + \varepsilon$$

where $\varepsilon \sim \mathcal{N}(0, 2^2)$. Assume $x \sim \mathcal{N}(0, 1^2)$, $\beta_0 = 0.5$ and $\beta_1 = 2$.

```
> set.seed(20)
> x <- rnorm(100)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-6.4080 -1.5400  0.6789  0.6893  2.9300  6.5050
> plot(x, y)
```

Generating Random Numbers From a Linear Model

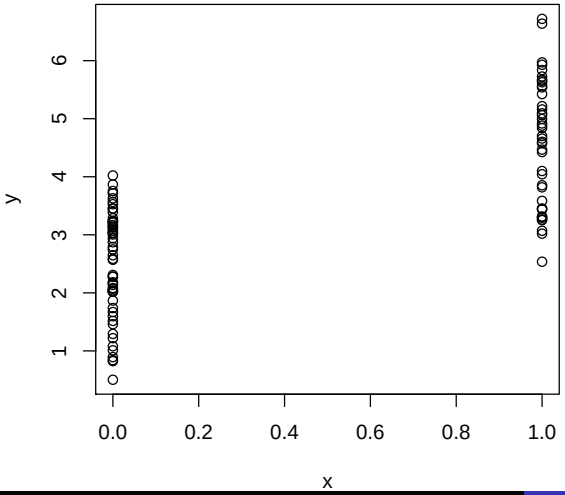


Generating Random Numbers From a Linear Model

What if x is binary?

```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-3.4940 -0.1409  1.5770  1.4320  2.8400  6.9410
> plot(x, y)
```

Generating Random Numbers From a Linear Model



Generating Random Numbers From a Generalized Linear Model

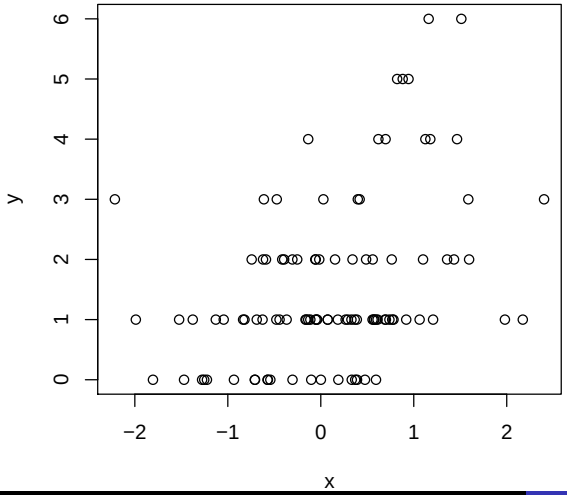
Suppose we want to simulate from a Poisson model where

$$\begin{aligned} Y &\sim \text{Poisson}(\mu) \\ \log \mu &= \beta_0 + \beta_1 x \end{aligned}$$

and $\beta_0 = 0.5$ and $\beta_1 = 0.3$. We need to use the `rpois` function for this

```
> set.seed(1)
> x <- rnorm(100)
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    1.00    1.00    1.55    2.00    6.00
> plot(x, y)
```

Generating Random Numbers From a Generalized Linear Model



Random Sampling

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

```
> set.seed(1)
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
> 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)
 [1]  2  3  4  1  9  5 10  8  6  7
> sample(1:10, replace = TRUE)  ## Sample w/replacement
 [1] 2 9 7 8 2 8 5 9 7 8
```

Summary

- Drawing samples from specific probability distributions can be done with `r*` functions
- Standard distributions are built in: Normal, Poisson, Binomial, Exponential, Gamma, etc.
- The `sample` function can be used to draw random samples from arbitrary vectors
- Setting the random number generator seed via `set.seed` is critical for reproducibility

Introduction to the R Language

Plotting

Computing for Data Analysis

Plotting

The plotting and graphics engine in R is encapsulated in a few base and recommend packages:

- **graphics**: contains plotting functions for the "base" graphing systems, including `plot`, `hist`, `boxplot` and many others.
- **lattice**: contains code for producing Trellis graphics, which are independent of the "base" graphics system; includes functions like `xyplot`, `bwplot`, `levelplot`
- **grid**: implements a different graphing system independent of the "base" system; the **lattice** package builds on top of **grid**; we seldom call functions from the **grid** package directly
- **grDevices**: contains all the code implementing the various graphics devices, including X11, PDF, PostScript, PNG, etc.

The Process of Making a Plot

When making a plot one must first make a few choices (not necessarily in this order):

- To what device will the plot be sent? The default in Unix is `x11`; on Windows it is `windows`; on Mac OS X it is `quartz`
- Is the plot for viewing temporarily on the screen, or will it eventually end up in a paper? Are you using it in a presentation? Plots included in a paper/presentation need to use a file device rather than a screen device.
- Is there a large amount of data going into the plot? Or is it just a few points?
- Do you need to be able to resize the graphic?

The Process of Making a Plot

- What graphics system will you use: base or grid/lattice? These generally cannot be mixed.
- Base graphics are usually constructed piecemeal, with each aspect of the plot handled separately through a series of function calls; this is often conceptually simpler and allows plotting to mirror the thought process
- Lattice/grid graphics are usually created in a single function call, so all of the graphics parameters have to specified at once; specifying everything at once allows R to automatically calculate the necessary spacings and font sizes.

Base Graphics

Base graphics are used most commonly and are a very powerful system for creating 2-D graphics.

- Calling `plot(x, y)` or `hist(x)` will launch a graphics device (if one is not already open) and draw the plot on the device
- If the arguments to `plot` are not of some special class, then the *default method* for `plot` is called; this function has *many* arguments, letting you set the title, x axis lable, y axis label, etc.
- The base graphics system has *many* parameters that can set and tweaked; these parameters are documented in `?par`; it wouldn't hurt to memorize this help page!

Some Important Base Graphics Parameters

The `par` function is used to specify global graphics parameters that affect all plots in an R session. These parameters can often be overridden as arguments to specific plotting functions.

- pch: the plotting symbol (default is open circle)
- lty: the line type (default is solid line), can be dashed, dotted, etc.
- lwd: the line width, specified as an integer multiple
- col: the plotting color, specified as a number, string, or hex code; the `colors` function gives you a vector of colors by name
- las: the orientation of the axis labels on the plot

Some Important Base Graphics Parameters

- bg: the background color
- mar: the margin size
- oma: the outer margin size (default is 0 for all sides)
- mfrow: number of plots per row, column (plots are filled row-wise)
- mfcol: number of plots per row, column (plots are filled column-wise)

Some Important Base Graphics Parameters

Some default values.

```
> par("lty")
[1] "solid"
> par("lwd")
[1] 1
> par("col")
[1] "black"
> par("pch")
[1] 1
```

Some Important Base Graphics Parameters

Some default values.

```
> par("bg")
[1] "transparent"
> par("mar")
[1] 5.1 4.1 4.1 2.1
> par("oma")
[1] 0 0 0 0
> par("mfrow")
[1] 1 1
> par("mfcol")
[1] 1 1
```

Some Important Base Plotting Functions

- `plot`: make a scatterplot, or other type of plot depending on the class of the object being plotted
- `lines`: add lines to a plot, given a vector x values and a corresponding vector of y values (or a 2-column matrix); this function just connects the dots
- `points`: add points to a plot
- `text`: add text labels to a plot using specified x, y coordinates
- `title`: add annotations to x, y axis labels, title, subtitle, outer margin
- `mtext`: add arbitrary text to the margins (inner or outer) of the plot
- `axis`: adding axis ticks/labels

Useful Graphics Devices

The list of devices is found in `?Devices`; there are also devices created by users on CRAN

- `pdf`: useful for line-type graphics, vector format, resizes well, usually portable
- `postscript`: older format, also vector format and resizes well, usually portable, can be used to create encapsulated postscript files, Windows systems often don't have a postscript viewer
- `xfig`: good of you use Unix and want to edit a plot by hand

Useful Graphics Devices

- `png`: bitmapped format, good for line drawings or images with solid colors, uses lossless compression (like the old GIF format), most web browsers can read this format natively, good for plotting many many many points, does not resize well
- `jpeg`: good for photographs or natural scenes, uses lossy compression, good for plotting many many many points, does not resize well, can be read by almost any computer and any web browser, not great for line drawings
- `bitmap`: needed to create bitmap files (png, jpeg) in certain situations (uses Ghostscript), also can be used to create a variety of other bitmapped formats not mentioned
- `bmp`: a native Windows bitmapped format

Copying Plots

There are two basic approaches to plotting.

1. Launch a graphics device
2. Make a plot; annotate if needed
3. Close graphics device

Or

1. Make a plot on a screen device (default); annotate if needed
2. Copy the plot to another device if necessary (not an exact process)

Copying Plots

Copying a plot to another device can be useful because some plots require a lot of code and it can be a pain to type all that in again for a different device.

- `dev.copy`: copy a plot from one device to another
- `dev.copy2pdf`: copy a plot to a Portable Document Format (PDF) file
- `dev.list`: show the list of open graphics devices
- `dev.next`: switch control to the next graphics device on the device list
- `dev.set`: set control to a specific graphics device
- `dev.off`: close the current graphics device

NOTE: Copying a plot is not an exact operation!

Lattice Functions

- `xyplot`: this is the main function for creating scatterplots
- `bwplot`: box-and-whiskers plots ("boxplots")
- `histogram`: histograms
- `stripplot`: like a boxplot but with actual points
- `dotplot`: plot dots on "violin strings"
- `splom`: scatterplot matrix; like `pairs` in base graphics system
- `levelplot`, `contourplot`: for plotting "image" data

Lattice Functions

Lattice functions generally take a formula for their first argument, usually of the form

```
y ~ x | f * g
```

- On the left of the ~ is the y variable, on the right is the x variable
- After the | are *conditioning variables* — they are optional; the * indicates an interaction
- The second argument is the data frame or list from which the variables in the formula should be obtained.
- If no data frame or list is passed, then the parent frame is used.
- If no other arguments are passed, there are defaults that can be used.

Lattice Behavior

Lattice functions behave differently from base graphics functions in one critical way.

- Base graphics functions plot data directly the graphics device
- Lattice graphics functions return an object of class `trellis`.
- The print methods for lattice functions actually do the work of plotting the data on the graphics device.
- Lattice functions return "plot objects" that can, in principle, be stored (but it's usually better to just save the code + data).
- On the command line, `trellis` objects are *auto-printed* so that it appears the function is plotting the data

Lattice Panel Functions

Lattice functions have a `panel` function which controls what happens inside each panel of the entire plot.

```
x <- rnorm(100)
y <- x + rnorm(100, sd = 0.5)
f <- gl(2, 50, labels = c("Group 1", "Group 2"))
xyplot(y ~ x | f)
```

plots y vs. x conditioned on f.

Lattice Panel Functions

```
xyplot(y ~ x | f,
       panel = function(x, y, ...) {
               panel.xyplot(x, y, ...)
               panel.abline(h = median(y),
                            lty = 2)
       })
```

plots y vs. x conditioned on f with horizontal (dashed) line drawn at the median of y for each panel.

Lattice Panel Functions

Adding a regression line

```
xyplot(y ~ x | f,
       panel = function(x, y, ...) {
               panel.xyplot(x, y, ...)
               panel.lmline(x, y, col = 2)
       })
```

fits and plots a simple linear regression line to each panel of the plot.

Mathematical Annotation

R can produce LaTeX-like symbols on a plot for mathematical annotation. This is very handy and is useful for making fun of people who use other statistical packages.

- Math symbols are "expressions" in R and need to be wrapped in the `expression` function
- There is a set list of allowed symbols and this is documented in ?plotmath
- Plotting functions that take arguments for text generally allow expressions for math symbols

Mathematical Annotation

Some examples.

```
plot(0, 0, main = expression(theta == 0),
     ylab = expression(hat(gamma) == 0),
     xlab = expression(sum(x[i] * y[i], i==1, n)))
```

Pasting strings together.

```
x <- rnorm(100)
hist(x,
     xlab=expression("The mean (" * bar(x) * ") is " *
                     sum(x[i]/n,i==1,n)))
```

Substituting

What if you want to use a computed value in the annotation?

```
x <- rnorm(100)
y <- x + rnorm(100, sd = 0.5)
plot(x, y,
     xlab=substitute(bar(x) == k, list(k=mean(x))),
     ylab=substitute(bar(y) == k, list(k=mean(y)))
     )
```

Or in a loop of plots

```
par(mfrow = c(2, 2))
for(i in 1:4) {
        x <- rnorm(100)
        hist(x, main=substitute(theta==num,list(num=i)))
}
```

Summary of Important Help Pages

- ?par
- ?plot
- ?xyplot
- ?plotmath
- ?axis

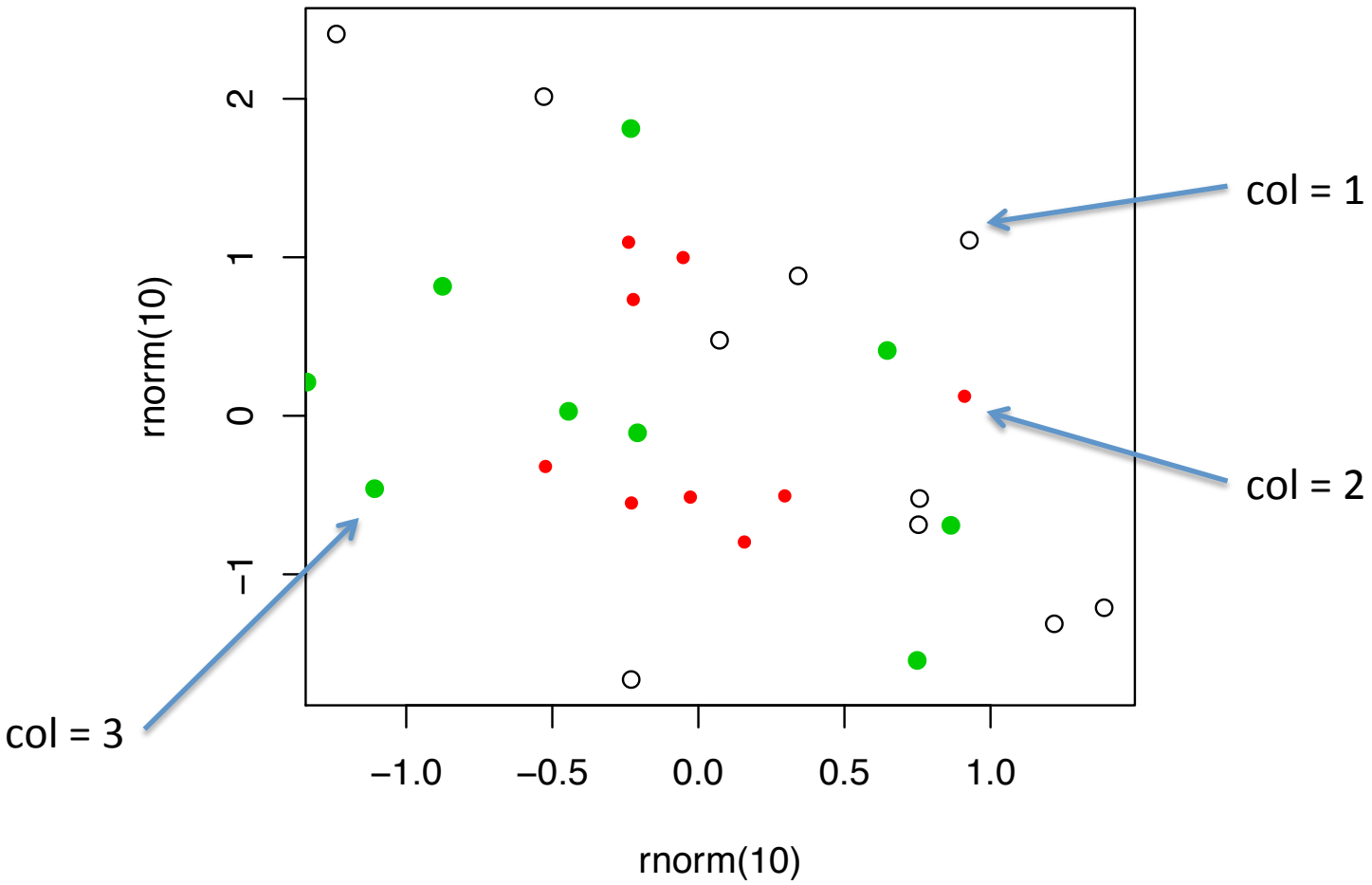
Plotting and Color in R

Computing for Data Analysis

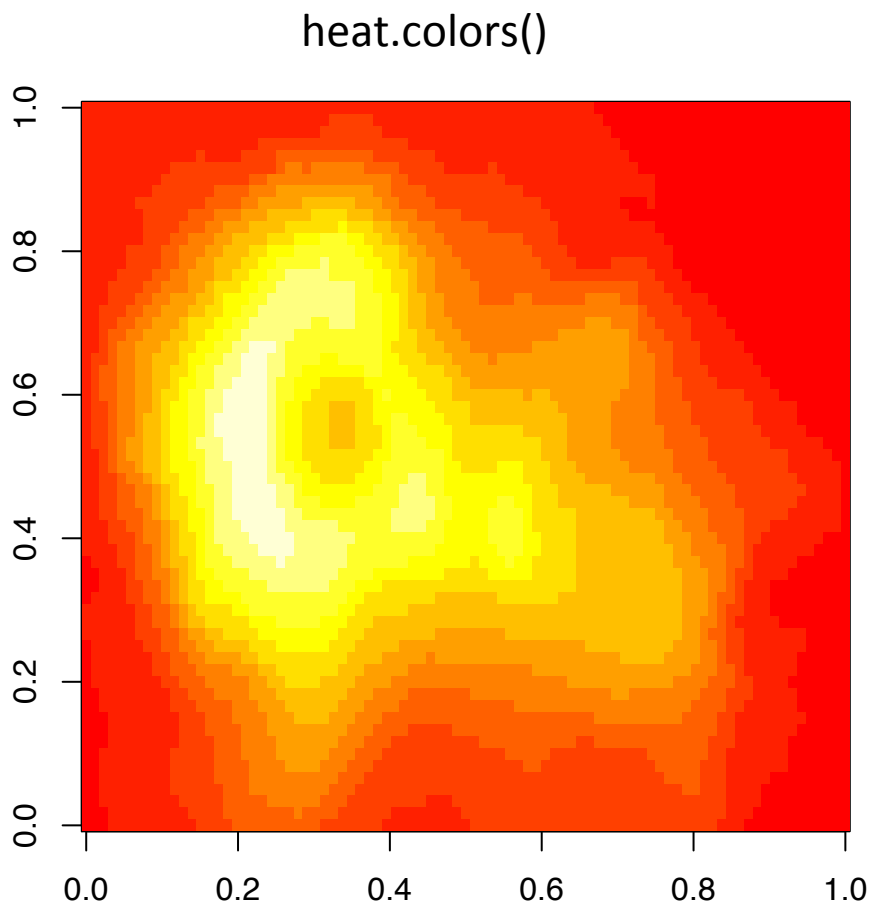
Plotting and Color

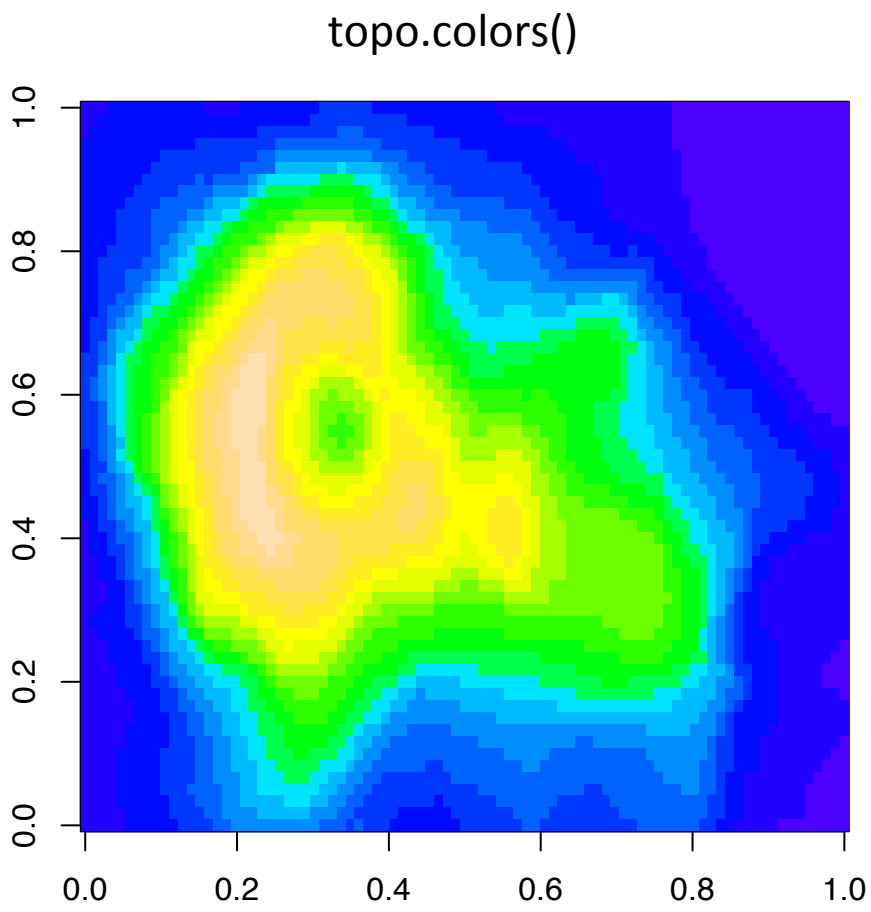
- The default color schemes for most plots in R are horrendous
  - I don't have good taste and even I know that
- Recently there have been developments to improve the handling/specification of colors in plots/graphs/etc.
- There are functions in R and in external packages that are very handy

Colors 1, 2, and 3



Default Image Plots in R





Color Utilities in R

- The **grDevices** package has two functions
  - `colorRamp`
  - `colorRampPalette`
- These functions take palettes of colors and help to interpolate between the colors
- The function `colors()` lists the names of colors you can use in any plotting function

Color Palette Utilities in R

- `colorRamp`: Take a palette of colors and return a function that takes values between 0 and 1, indicating the extremes of the color palette (e.g. see the 'gray' function)
- `colorRampPalette`: Take a palette of colors and return a function that takes integer arguments and returns a vector of colors interpolating the palette (like `heat.colors` or `topo.colors`)

colorRamp

```
> pal <- colorRamp(c("red", "blue"))

> pal(0)
     [,1] [,2] [,3]
[1,]  255    0    0

> pal(1)
     [,1] [,2] [,3]
[1,]    0    0  255

> pal(0.5)
      [,1] [,2]  [,3]
[1,] 127.5    0 127.5
```

colorRamp

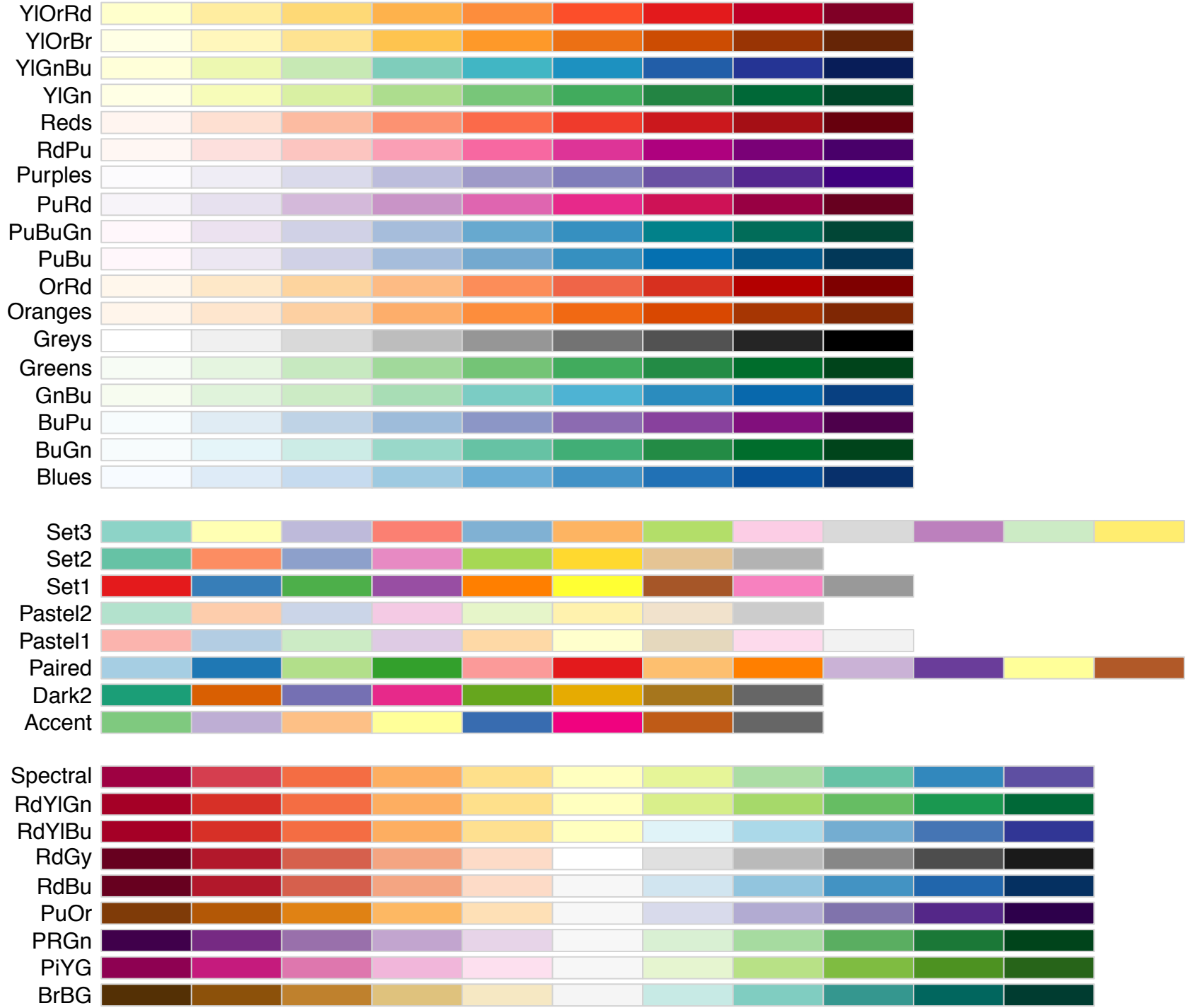
```
> pal(seq(0, 1, len = 10))
           [,1] [,2]      [,3]
 [1,] 255.00000    0   0.00000
 [2,] 226.66667    0  28.33333
 [3,] 198.33333    0  56.66667
 [4,] 170.00000    0  85.00000
 [5,] 141.66667    0 113.33333
 [6,] 113.33333    0 141.66667
 [7,]  85.00000    0 170.00000
 [8,]  56.66667    0 198.33333
 [9,]  28.33333    0 226.66667
[10,]   0.00000    0 255.00000
```

colorRampPalette

```
> pal <- colorRampPalette(c("red", "yellow"))

> pal(2)
[1] "#FF0000" "#FFFF00"

> pal(10)
 [1] "#FF0000" "#FF1C00" "#FF3800" "#FF5500" "#FF7100"
 [6] "#FF8D00" "#FFAA00" "#FFC600" "#FFE200" "#FFFF00”
```

RColorBrewer Package

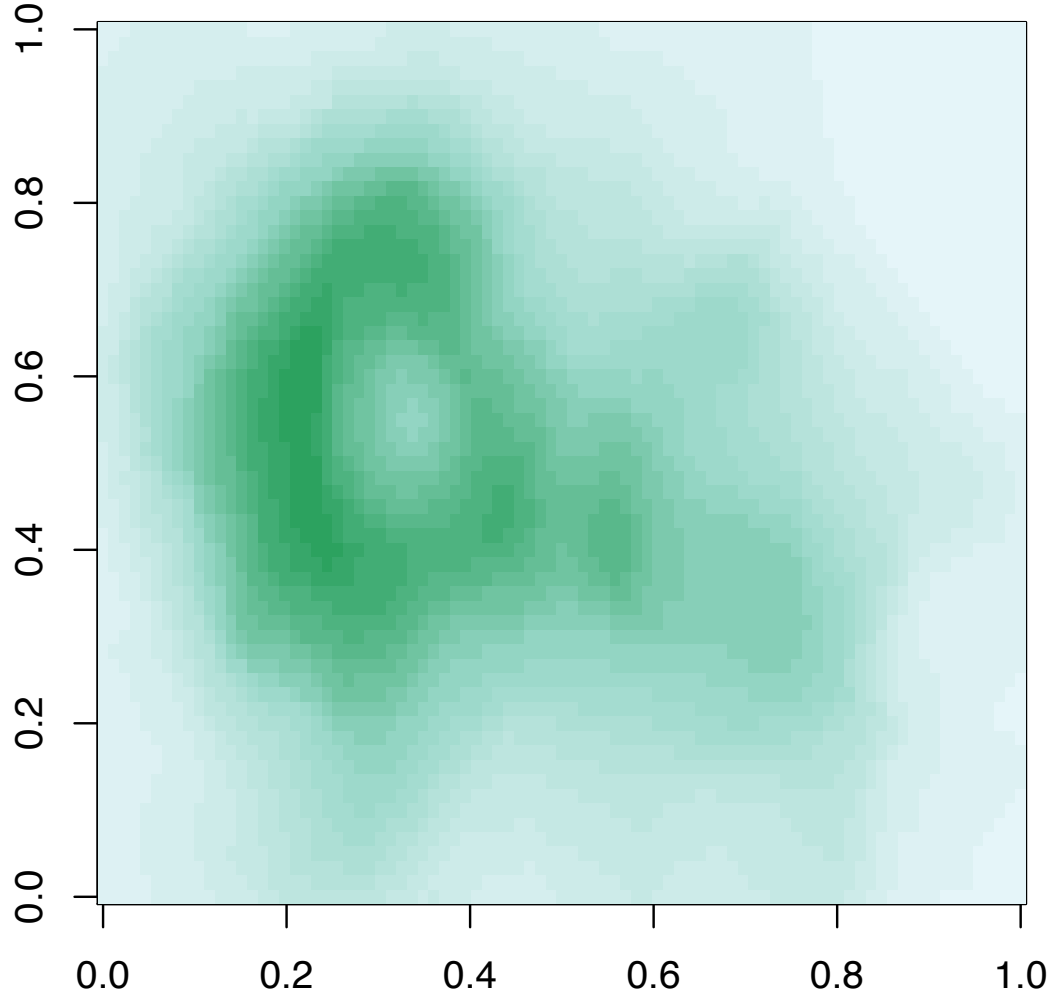
- One package on CRAN that contains interesting/useful color palettes
- There are 3 types of palettes
  - Sequential
  - Diverging
  - Qualitative
- Palette information can be used in conjunction with the `colorRamp()` and `colorRampPalette()`


YlOrRd
YlOrBr
YlGnBu
YlGn
Reds
RdPu
Purples
PuRd
PuBuGn
PuBu
OrRd
Oranges
Greys
Greens
GnBu
BuPu
BuGn
Blues
Set3
Set2
Set1
Pastel2
Pastel1
Paired
Dark2
Accent
Spectral
RdYlGn
RdYlBu
RdGy
RdBu
PuOr
PRGn
PiYG
BrBG

RColorBrewer and colorRampPalette

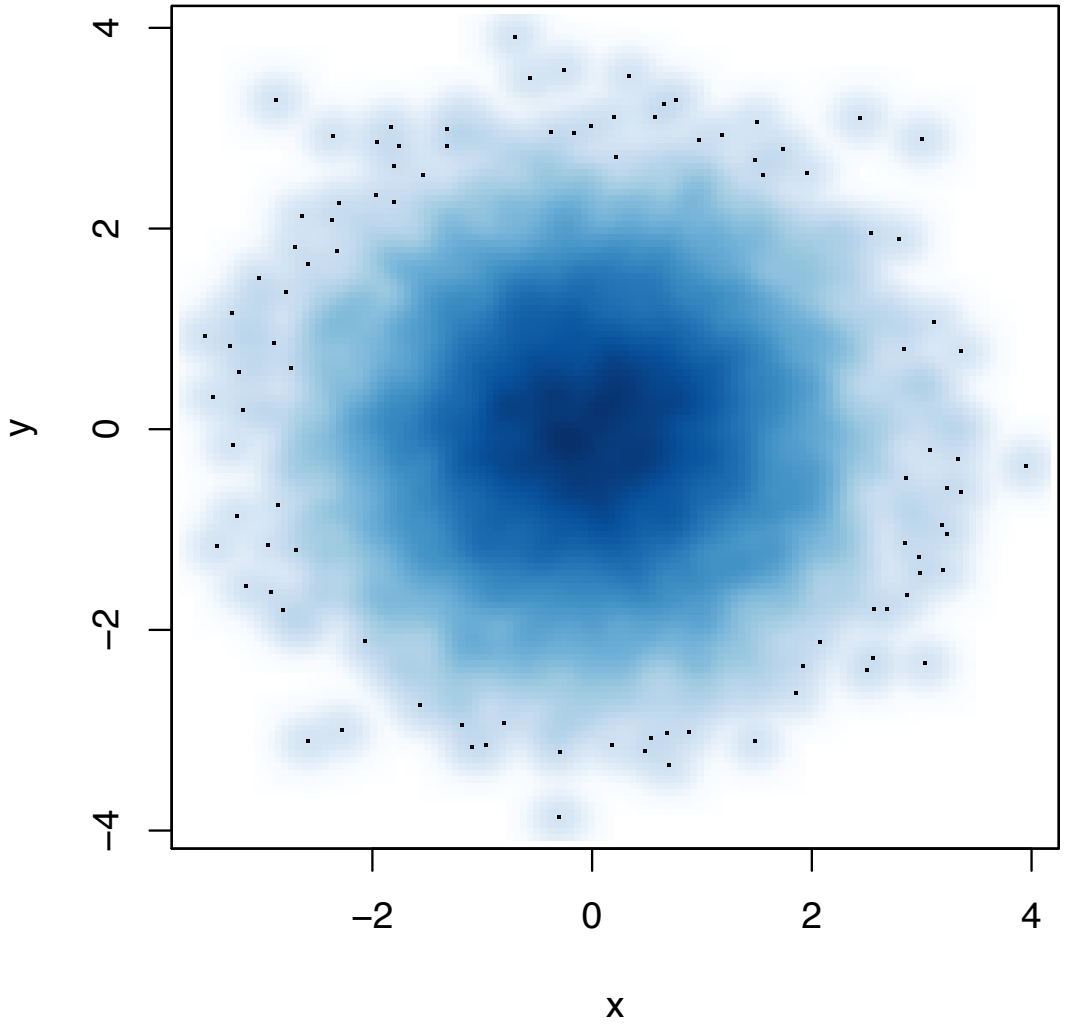
```
> library(RColorBrewer)

> cols <- brewer.pal(3, "BuGn")

> cols
[1] "#E5F5F9" "#99D8C9" "#2CA25F"

> pal <- colorRampPalette(cols)

> image(volcano, col = pal(20))
```

RColorBrewer and colorRampPalette



The smoothScatter function

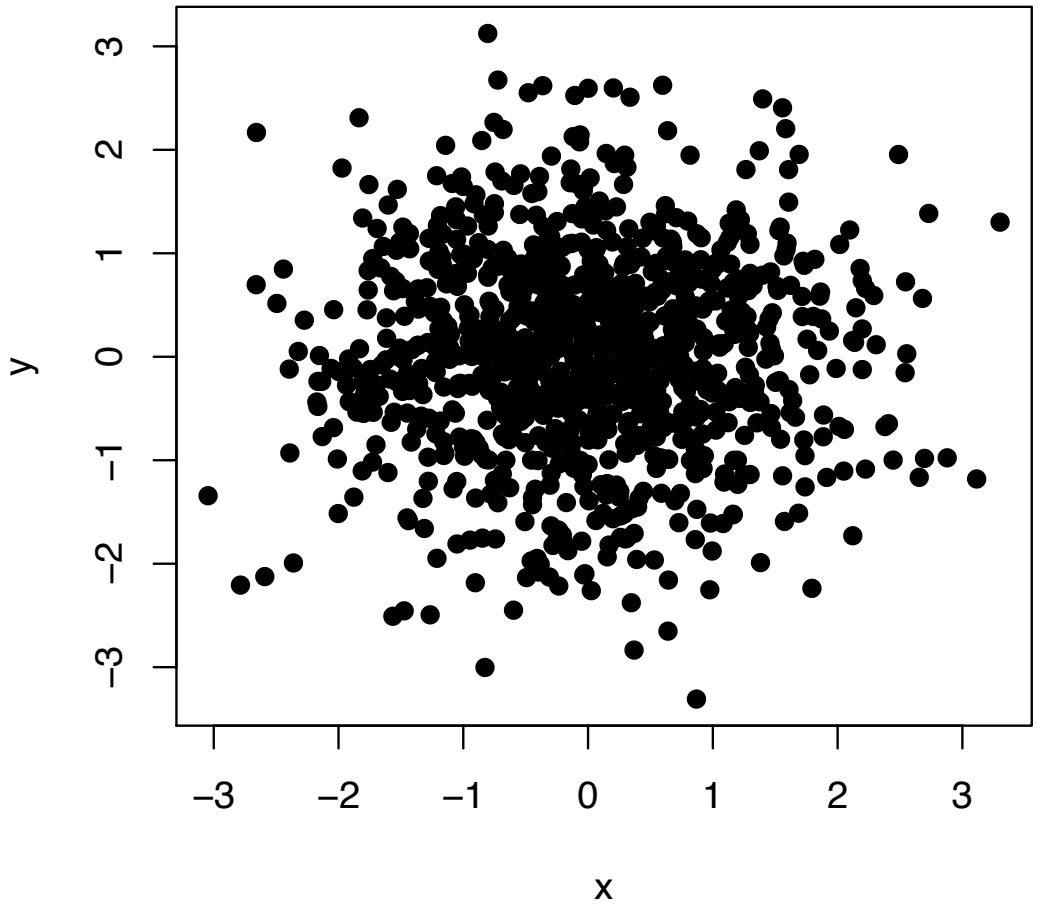
```
x <- rnorm(10000)
y <- rnorm(10000)
smoothScatter(x, y)
```



Some Other Plotting Notes

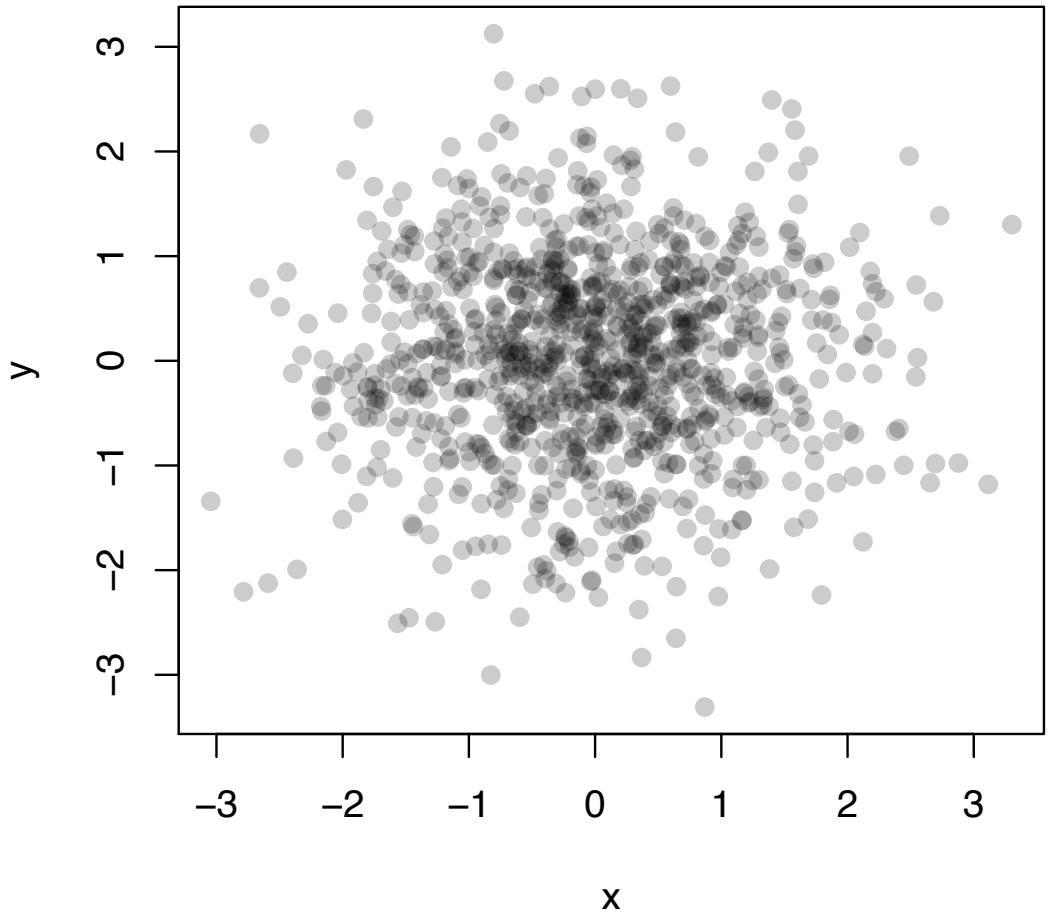
- The `rgb` function can be used to produce any color via red, green, blue proportions
- Color transparency can be added via the `alpha` parameter to `rgb`
- The **colorspace** package can be used for a different control over colors

Scatterplot with no transparency



plot(x, y, pch = 19)

Scatterplot with transparency



plot(x, y, col = rgb(0, 0, 0, 0.2), pch = 19)

Summary

- Careful use of colors in plots/maps/etc. can make it easier for the reader to get what you're trying to say (why make it harder?)
- The **RColorBrewer** package is an R package that provides color palettes for sequential, categorical, and diverging data
- The `colorRamp` and `colorRampPalette` functions can be used in conjunction with color palettes to connect data to colors
- Transparency can sometimes be used to clarify plots with many points

Regular Expressions

Computing for Data Analysis

Regular expressions

- Regular expressions can be thought of as a combination of literals and *metacharacters*
- To draw an analogy with natural language, think of literal text forming the words of this language, and the metacharacters defining its grammar
- Regular expressions have a rich set of metacharacters

Literals

Simplest pattern consists only of literals. The literal “nuclear” would match to the following lines:

```
Ooh. I just learned that to keep myself alive after a
nuclear blast! All I have to do is milk some rats
then drink the milk. Aweosme. :}

Laozi says nuclear weapons are mas macho

Chaos in a country that has nuclear weapons -- not good.

my nephew is trying to teach me nuclear physics, or
possibly just trying to show me how smart he is
so I'll be proud of him [which I am].

lol if you ever say "nuclear" people immediately think
DEATH by radiation LOL
```

Literals

The literal “Obama” would match to the following lines

```
Politics r dum. Not 2 long ago Clinton was sayin Obama
was crap n now she sez vote 4 him n unite? WTF?
Screw em both + Mcain. Go Ron Paul!

Clinton conceeds to Obama but will her followers listen??

Are we sure Chelsea didn't vote for Obama?

thinking ... Michelle Obama is terrific!

jetlag..no sleep...early mornig to starbux..Ms. Obama
was moving
```

Regular Expressions

- Simplest pattern consists only of literals; a match occurs if the sequence of literals occurs anywhere in the text being tested
- What if we only want the word "Obama"? or sentences that end in the word "Clinton", or "clinton" or "clinto"?

Regular Expressions

We need a way to express

- whitespace word boundaries
- sets of literals
- the beginning and end of a line
- alternatives (“war” or “peace”)

Metacharacters to the rescue!

Metacharacters

Some metacharacters represent the start of a line

```
^i think
```

will match the lines

```
i think we all rule for participating
i think i have been outed
i think this will be quite fun actually
i think i need to go to work
i think i first saw zombo in 1999.
```

Metacharacters

$ represents the end of a line

```
morning$
```

will match the lines

```
well they had something this morning
then had to catch a tram home in the morning
dog obedience school in the morning
and yes happy birthday i forgot to say it earlier this morning
I walked in the rain this morning
good morning
```

Character Classes with []

We can list a set of characters we will accept at a given point in the match

```
[Bb][Uu][Ss][Hh]
```

will match the lines

```
The democrats are playing, "Name the worst thing about Bush!"
I smelled the desert creosote bush, brownies, BBQ chicken
BBQ and bushwalking at Molonglo Gorge
Bush TOLD you that North Korea is part of the Axis of Evil
I'm listening to Bush - Hurricane (Album Version)
```

Character Classes with []

```
^[Ii] am
```

will match

```
i am so angry at my boyfriend i can't even bear to
look at him

i am boycotting the apple store

I am twittering from iPhone

I am a very vengeful person when you ruin my sweetheart.

I am so over this. I need food. Mmmm bacon...
```

Character Classes with []

Similarly, you can specify a range of letters [a-z] or [a-zA-Z]; notice that the order doesn't matter

```
^[0-9][a-zA-Z]
```

will match the lines

```
7th inning stretch
2nd half soon to begin. OSU did just win something
3am - cant sleep - too hot still.. :(
5ft 7 sent from heaven
1st sign of starvagtion
```

Character Classes with []

When used at the beginning of a character class, the "^" is also a metacharacter and indicates matching characters NOT in the indicated class

```
[^?.]$
```

will match the lines

```
i like basketballs
6 and 9
dont worry... we all die anyway!
Not in Baghdad
helicopter under water? hmmm
```

More Metacharacters

"." is used to refer to any character. So

```
9.11
```

will match the lines

```
its stupid the post 9-11 rules
if any 1 of us did 9/11 we would have been caught in days.
NetBios: scanning ip 203.169.114.66
Front Door 9:11:46 AM
Sings: 0118999881999119725...3 !
```

More Metacharacters: |

This does not mean "pipe" in the context of regular expressions; instead it translates to "or"; we can use it to combine two expressions, the subexpressions being called alternatives

```
flood|fire
```

will match the lines

```
is firewire like usb on none macs?
the global flood makes sense within the context of the bible
yeah ive had the fire on tonight
... and the floods, hurricanes, killer heatwaves, rednecks, gun nuts, etc.
```

More Metacharacters: |

We can include any number of alternatives...

```
flood|earthquake|hurricane|coldfire
```

will match the lines

```
Not a whole lot of hurricanes in the Arctic.
We do have earthquakes nearly every day somewhere in our State
hurricanes swirl in the other direction
coldfire is STRAIGHT!
'cause we keep getting earthquakes
```

More Metacharacters: |

The alternatives can be real expressions and not just literals

`^[Gg]ood|[Bb]ad`

will match the lines

```
good to hear some good knews from someone here
Good afternoon fellow american infidels!
good on you-what do you drive?
Katie... guess they had bad experiences...
my middle name is trouble, Miss Bad News
```

More Metacharacters: (and)

Subexpressions are often contained in parentheses to constrain the alternatives

```
^([Gg]ood|[Bb]ad)
```

will match the lines

```
bad habbit
bad coordination today
good, becuase there is nothing worse than a man in kinky underwear
Badcop, its because people want to use drugs
Good Monday Holiday
Good riddance to Limey
```

More Metacharacters: ?

The question mark indicates that the indicated expression is optional

```
[Gg]eorge( [Ww]\.)? [Bb]ush
```

will match the lines

```
i bet i can spell better than you and george bush combined
BBC reported that President George W. Bush claimed God told him to invade
a bird in the hand is worth two george bushes
```

One thing to note...

In the following

```
[Gg]eorge( [Ww]\.)? [Bb]ush
```

we wanted to match a "." as a literal period; to do that, we had to "escape" the metacharacter, preceding it with a backslash In general, we have to do this for any metacharacter we want to include in our match

More metacharacters: * and +

The * and + signs are metacharacters used to indicate repetition; * means "any number, including none, of the item" and + means "at least one of the item"

`(.*)`

will match the lines

```
anyone wanna chat? (24, m, germany)
hello, 20.m here... ( east area + drives + webcam )
(he means older men)
()
```

More metacharacters: * and +

The * and + signs are metacharacters used to indicate repetition; * means "any number, including none, of the item" and + means "at least one of the item"

```
[0-9]+ (.*)[0-9]+
```

will match the lines

```
working as MP here 720 MP battallion, 42nd birgade
so say 2 or 3 years at colleage and 4 at uni makes us 23 when and if we fi
it went down on several occasions for like, 3 or 4 *days*
Mmmm its time 4 me 2 go 2 bed
```

More metacharacters: { and }

{ and } are referred to as interval quantifiers; the let us specify the minimum and maximum number of matches of an expression

```
[Bb]ush( +[^ ]+ +){1,5} debate
```

will match the lines

```
Bush has historically won all major debates he's done.
in my view, Bush doesn't need these debates..
bush doesn't need the debates? maybe you are right
That's what Bush supporters are doing about the debate.
Felix, I don't disagree that Bush was poorly prepared for the debate.
indeed, but still, Bush should have taken the debate more seriously.
Keep repeating that Bush smirked and scowled during the debate
```

More metacharacters: and

- m,n means at least m but not more than n matches
- m means exactly m matches
- m, means at least m matches

More metacharacters: (and) revisited

- In most implementations of regular expressions, the parentheses not only limit the scope of alternatives divided by a "|", but also can be used to "remember" text matched by the subexpression enclosed
- We refer to the matched text with \1, \2, etc.

More metacharacters: (and) revisited

So the expression

```
+([a-zA-Z]+) +\1 +
```

will match the lines

```
time for bed, night night twitter!

blah blah blah blah

my tattoo is so so itchy today

i was standing all all alone against the world outside...

hi anybody anybody at home

estudiando css css css css.... que desastritooooo
```

More metacharacters: (and) revisited

The * is "greedy" so it always matches the *longest* possible string that satisfies the regular expression. So

```
^s(.*)s
```

matches

```
sitting at starbucks

setting up mysql and rails

studying stuff for the exams

spaghetti with marshmallows

stop fighting with crackers

sore shoulders, stupid ergonomics
```

More metacharacters: (and) revisited

The greediness of * can be turned off with the ?, as in

```
^s(.*?)s$
```

Summary

- Regular expressions are used in many different languages; not unique to R.
- Regular expressions are composed of literals and metacharacters that represent sets or classes of characters/words
- Text processing via regular expressions is a very powerful way to extract data from "unfriendly" sources (not all data comes as a CSV file)

(Thanks to Mark Hansen for some material in this lecture.)

Regular Expressions in R

Computing for Data Analysis

Regular Expression Functions

The primary R functions for dealing with regular expressions are

- `grep`, `grepl`: Search for matches of a regular expression/pattern in a character vector; either return the indices into the character vector that match, the strings that happen to match, or a TRUE/FALSE vector indicating which elements match
- `regexpr`, `gregexpr`: Search a character vector for regular expression matches and return the indices of the string where the match begins and the length of the match
- `sub`, `gsub`: Search a character vector for regular expression matches and replace that match with another string
- `regexec`: Easier to explain through demonstration.

Here is an excerpt of the Baltimore City homicides dataset:

```
> homicides <- readLines("homicides.txt")
> homicides[1]
[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon
Nelson</dt><dd class=\"address\">3400 Clifton Ave.<br />Baltimore, MD
21216</dd><dd>black male, 17 years old</dd>
<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd></dl>'"

> homicides[1000]
[1] "39.33626300000, -76.55553990000, icon_homicide_shooting, 'p1200',...
```

How can I find the records for all the victims of shootings (as opposed to other causes)?

grep

```
> length(grep("iconHomicideShooting", homicides))
[1] 228
> length(grep("iconHomicideShooting|icon_homicide_shooting", homicides))
[1] 1003
> length(grep("Cause: shooting", homicides))
[1] 228
> length(grep("Cause: [Ss]hooting", homicides))
[1] 1003
> length(grep("[Ss]hooting", homicides))
[1] 1005
```

grep

```
> i <- grep("[cC]ause: [Ss]hooting", homicides)
> j <- grep("[Ss]hooting", homicides)
> str(i)
 int [1:1003] 1 2 6 7 8 9 10 11 12 13 ...
> str(j)
 int [1:1005] 1 2 6 7 8 9 10 11 12 13 ...
> setdiff(i, j)
integer(0)
> setdiff(j, i)
[1] 318 859
```

```
> homicides[859]
[1] "39.33743900000, -76.66316500000, icon_homicide_bluntforce,
'p914', '<dl><dt><a href=\"http://essentials.baltimoresun.com/
micro_sun/homicides/victim/914/steven-harris\">Steven Harris</a>
</dt><dd class=\"address\">4200 Pimlico Road<br />Baltimore, MD 21215
</dd><dd>Race: Black<br />Gender: male<br />Age: 38 years old</dd>
<dd>Found on July 29, 2010</dd><dd>Victim died at Scene</dd>
<dd>Cause: Blunt Force</dd><dd class=\"popup-note\"><p>Harris was
found dead July 22 and ruled a shooting victim; an autopsy
subsequently showed that he had not been shot,...</dd></dl>'"
```

grep

By default, grep returns the indices into the character vector where the regex pattern matches.

```
> grep("^New", state.name)
[1] 29 30 31 32
```

Setting `value = TRUE` returns the actual elements of the character vector that match.

```
> grep("^New", state.name, value = TRUE)
[1] "New Hampshire" "New Jersey"    "New Mexico"    "New York"
```

grepl returns a logical vector indicating which element matches.

```
> grepl("^New", state.name)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALS
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALS
[25] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALS
[37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALS
[49] FALSE FALSE
```

Some limitations of `grep`

- The `grep` function tells you which strings in a character vector match a certain pattern but it doesn't tell you exactly where the match occurs or what the match is (for a more complicated regex.
- The `regexpr` function gives you the index into each string where the match begins and the length of the match for that string.
- `regexpr` only gives you the first match of the string (reading left to right). `gregexpr` will give you all of the matches in a given string.

How can we find the date of the homicide?

```
> homicides[1]
[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon
Nelson</dt><dd class=\"address\">3400 Clifton Ave.<br />Baltimore,
MD 21216</dd><dd>black male, 17 years old</dd>
<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd></dl>'"
```

Can we just 'grep' on "Found"?

The word 'found' may be found elsewhere in the entry.

```
> homicides[954]
[1] "39.30677400000, -76.59891100000, icon_homicide_shooting, 'p816',
'<dl><dd class=\"address\">1400 N Caroline St<br />Baltimore, MD 21213</dd
<dd>Race: Black<br />Gender: male<br />Age: 29 years old</dd>
<dd>Found on March  3, 2010</dd><dd>Victim died at Scene</dd>
<dd>Cause: Shooting</dd><dd class=\"popup-note\"><p>Wheeler\\'s body
was found on the grounds of Dr. Bernard Harris Sr. Elementary
School</p></dd></dl>'"
```

Let's use the pattern

```
<dd>[F|f]ound(.*)</dd>
```

What does this look for?

```
> regexpr("<dd>[F|f]ound(.*)</dd>", homicides[1:10])
 [1] 177 178 188 189 178 182 178 187 182 183
attr(,"match.length")
 [1] 93 86 89 90 89 84 85 84 88 84
attr(,"useBytes")
[1] TRUE
> substr(homicides[1], 177, 177 + 93 - 1)
[1] "<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
 Trauma</dd><dd>Cause: shooting</dd>"
```

The previous pattern was too greedy and matched too much of the string. We need to use the ? metacharacter to make the regex "lazy".

```
> regexpr("<dd>[F|f]ound(.*?)</dd>", homicides[1:10])
 [1] 177 178 188 189 178 182 178 187 182 183
attr(,"match.length")
 [1] 33 33 33 33 33 33 33 33 33 33
attr(,"useBytes")
[1] TRUE

> substr(homicides[1], 177, 177 + 33 - 1)
[1] "<dd>Found on January 1, 2007</dd>"
```

regmatches

One handy function is `regmatches` which extracts the matches in the strings for you without you having to use `substr`.

```
> r <- regexpr("<dd>[F|f]ound(.*?)</dd>", homicides[1:5])
> regmatches(homicides[1:5], r)
[1] "<dd>Found on January 1, 2007</dd>" "<dd>Found on January 2, 2007</dd>"
[3] "<dd>Found on January 2, 2007</dd>" "<dd>Found on January 3, 2007</dd>"
[5] "<dd>Found on January 5, 2007</dd>"
```

sub/gsub

Sometimes we need to clean things up or modify strings by matching a pattern and replacing it with something else. For example, how can we extract the data from this string?

```
> x <- substr(homicides[1], 177, 177 + 33 - 1)
> x
[1] "<dd>Found on January 1, 2007</dd>"
```

We want to strip out the stuff surrounding the "January 1, 2007" piece.

```
> sub("<dd>[F|f]ound on |</dd>", "", x)
[1] "January 1, 2007</dd>"

> gsub("<dd>[F|f]ound on |</dd>", "", x)
[1] "January 1, 2007"
```

sub/gsub

sub/gsub can take vector arguments

```
> r <- regexpr("<dd>[F|f]ound(.*?)</dd>", homicides[1:5])
> m <- regmatches(homicides[1:5], r)
> m
[1] "<dd>Found on January 1, 2007</dd>" "<dd>Found on January 2, 2007</dd>
[3] "<dd>Found on January 2, 2007</dd>" "<dd>Found on January 3, 2007</dd>
[5] "<dd>Found on January 5, 2007</dd>"
> gsub("<dd>[F|f]ound on |</dd>", "", m)
[1] "January 1, 2007" "January 2, 2007" "January 2, 2007" "January 3, 2007
[5] "January 5, 2007"
> as.Date(d, "%B %d, %Y")
[1] "2007-01-01" "2007-01-02" "2007-01-02" "2007-01-03" "2007-01-05"
```

regexec

The `regexec` function works like `regexpr` except it gives you the indices for parenthesized sub-expressions.

```
> regexec("<dd>[F|f]ound on (.*?)</dd>", homicides[1])
[[1]]
[1] 177 190
attr(,"match.length")
[1] 33 15

> regexec("<dd>[F|f]ound on .*?</dd>", homicides[1])
[[1]]
[1] 177
attr(,"match.length")
[1] 33
```

Now we can extract the string in the parenthesized sub-expression.

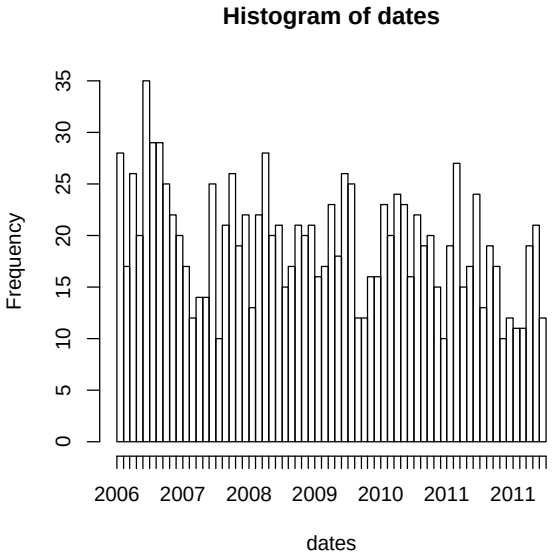
```
> regexec("<dd>[F|f]ound on (.*?)</dd>", homicides[1])
[[1]]
[1] 177 190
attr(,"match.length")
[1] 33 15

> substr(homicides[1], 177, 177 + 33 - 1)
[1] "<dd>Found on January 1, 2007</dd>"

> substr(homicides[1], 190, 190 + 15 - 1)
[1] "January 1, 2007"
```

Even easier with the `regmatches` function.

```
> r <- regexec("<dd>[F|f]ound on (.*?)</dd>", homicides[1:2])
> regmatches(homicides[1:2], r)
[[1]]
[1] "<dd>Found on January 1, 2007</dd>" "January 1, 2007"

[[2]]
[1] "<dd>Found on January 2, 2007</dd>" "January 2, 2007"
```

Let's make a plot of monthly homicide counts

```
> r <- regexec("<dd>[F|f]ound on (.*?)</dd>", homicides)
> m <- regmatches(homicides, r)
> dates <- sapply(m, function(x) x[2])
> dates <- as.Date(dates, "%B %d, %Y")
> hist(dates, "month", freq = TRUE)
```


Histogram of dates
Frequency
0
5
10
15
20
25
30
35
2006
2007
2008
2009
2010
2011
2011
dates

Summary

The primary R functions for dealing with regular expressions are

- `grep`, `grepl`: Search for matches of a regular expression/pattern in a character vector
- `regexpr`, `gregexpr`: Search a character vector for regular expression matches and return the indices where the match begins; useful in conjunction with `regmatches`
- `sub`, `gsub`: Search a character vector for regular expression matches and replace that match with another string
- `regexec`: Gives you indices of parethensized sub-expressions.

Classes and Methods in R

Computing for Data Analysis

Classes and Methods

- A system for doing object oriented programming
- R was originally quite interesting because it is both interactive *and* has a system for object orientation.
  - Other languages which support OOP (C++, Java, Lisp, Python, Perl) generally speaking are not interactive languages
- In R much of the code for supporting classes/methods is written by John Chambers himself (the creator of the original S language) and documented in the book *Programming with Data: A Guide to the S Language*
- A natural extension of Chambers' idea of allowing someone to cross the user $\longrightarrow$ programmer spectrum
- Object oriented programming is a bit different in R than it is in most languages — even if you are familiar with the idea, you may want to pay attention to the details

Two styles of classes and methods

S3 classes/methods

- Included with version 3 of the S language.
- Informal, a little kludgey
- Sometimes called *old-style* classes/methods

S4 classes/methods

- more formal and rigorous
- Included with S-PLUS 6 and R 1.4.0 (December 2001)
- Also called *new-style* classes/methods

Two worlds living side by side

- For now (and the forseeable future), S3 classes/methods and S4 classes/methods are separate systems (but they can be mixed to some degree).
- Each system can be used fairly independently of the other.
- Developers of new projects (you!) are encouraged to use the S4 style classes/methods.
  - Used extensively in the Bioconductor project
- But many developers still use S3 classes/methods because they are "quick and dirty" (and easier).
- In this lecture we will focus primarily on S4 classes/methods
- The code for implementing S4 classes/methods in R is in the **methods** package, which is usually loaded by default (but you can load it with `library(methods)` if for some reason it is not loaded)

Object Oriented Programming in R

- A *class* is a description of an thing. A class can be defined using `setClass()` in the **methods** package.
- An *object* is an instance of a class. Objects can be created using `new()`.
- A *method* is a function that only operates on a certain class of objects.
- A generic function is an R function which dispatches methods. A generic function typically encapsulates a "generic" concept (e.g. `plot`, `mean`, `predict`, ...)
  - The generic function does not actually do any computation.
- A *method* is the implementation of a generic function for an object of a particular class.

Things to look up

- The help files for the 'methods' package are extensive — do read them as they are the primary documentation
- You may want to start with `?Classes` and `?Methods`
- Check out `?setClass`, `?setMethod`, and `?setGeneric`
- Some of it gets technical, but try your best for now—it will make sense in the future as you keep using it.
- Most of the documentation in the **methods** package is oriented towards developers/programmers as these are the primary people using classes/methods

Classes

All objects in R have a class which can be determined by the class function

```
> class(1)
[1] "numeric"
> class(TRUE)
[1] "logical"
> class(rnorm(100))
[1] "numeric"
> class(NA)
[1] "logical"
> class("foo")
[1] "character"
```

Classes (cont'd)

Data classes go beyond the atomic classes

```
> x <- rnorm(100)
> y <- x + rnorm(100)
> fit <- lm(y ~ x)  ## linear regression model
> class(fit)
```

```
[1] "lm"
```

Generics/Methods in R

- S4 and S3 style generic functions look different but conceptually, they are the same (they play the same role).
- When you program you can write new methods for an existing generic OR create your own generics and associated methods.
- Of course, if a data type does not exist in R that matches your needs, you can always define a new class along with generics/methods that go with it

An S3 generic function (in the 'base' package)

The `mean` function is generic

```
> mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x7fc25c27afc0>
<environment: namespace:base>
```

So is the `print` function

```
> print
```

```
function (x, ...)
UseMethod("print")
<bytecode: 0x7fc25bd8ee00>
<environment: namespace:base>
```

S3 methods

```
> methods("mean")

[1] mean.data.frame mean.Date
[3] mean.default    mean.difftime
[5] mean.POSIXct    mean.POSIXlt
```

An S4 generic function (from the 'methods' package)

The S4 equivalent of `print` is `show`

```
> show
```

```
standardGeneric for "show" defined from package "methods"

function (object)
standardGeneric("show")
<bytecode: 0x7fc25b5ced08>
<environment: 0x7fc25c51aea0>
Methods may be defined for arguments: object
Use  showMethods("show")  for currently available ones.
(This generic function excludes non-simple inheritance; see ?setIs)
```

The `show` function is usually not called directly (much like `print`) because objects are auto-printed

S4 methods

There are many different methods for the `show` generic function

```
> showMethods("show")
Function: show (package methods)
object="ANY"
object="classGeneratorFunction"
object="classRepresentation"
object="envRefClass"
object="function"
    (inherited from: object="ANY")
object="genericFunction"
object="genericFunctionWithTrace"
object="MethodDefinition"
object="MethodDefinitionWithTrace"
object="MethodSelectionReport"
object="MethodWithNext"
object="MethodWithNextWithTrace"
object="namedList"
```

Generic/method mechanism

The first argument of a generic function is an object of a particular class (there may be other arguments)

1. The generic function checks the class of the object.
2. A search is done to see if there is an appropriate method for that class.
3. If there exists a method for that class, then that method is called on the object and we're done.
4. If a method for that class does not exist, a search is done to see if there is a default method for the generic. If a default exists, then the default method is called.
5. If a default method doesn't exist, then an error is thrown.

Examining Code for Methods

Examining the code for an S3 or S4 method requires a call to a special function

- You cannot just print the code for a method like other functions because the code for the method is usually hidden.
- If you want to see the code for an S3 method, you can use the function `getS3method`.
- The call is `getS3method(<generic>, <class>)`
- For S4 methods you can use the function `getMethod`
- The call is `getMethod(<generic>, <signature>)` (more details later)

S3 Class/Method: Example 1

What's happening here?

```
> set.seed(2)
> x <- rnorm(100)
> mean(x)
```

```
[1] -0.03069816
```

1. The class of x is "numeric"
2. But there is no mean method for "numeric" objects!
3. So we call the default function for `mean`.

S3 Class/Method: Example 1

```
> head(getS3method("mean", "default"))
```

```
1 function (x, trim = 0, na.rm = FALSE, ...)
2 {
3     if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
4         warning("argument is not numeric or logical: returning NA")
5         return(NA_real_)
6     }
```

```
> tail(getS3method("mean", "default"))
```

```
19          lo <- floor(n * trim) + 1
20          hi <- n + 1 - lo
21          x <- sort.int(x, partial = unique(c(lo, hi)))[lo:hi]
22      }
23      .Internal(mean(x))
24 }
```

S3 Class/Method: Example 2

What happens here?

```
> set.seed(3)
> df <- data.frame(x = rnorm(100), y = 1:100)
> sapply(df, mean)
```

```
          x           y
0.01103557 50.50000000
```

1. The class of df is "data.frame"; in a data frame each column can be an object of a different class
2. We `sapply` over the columns and call the `mean` function
3. In each column, `mean` checks the class of the object and dispatches the appropriate method.
4. Here we have a `numeric` column and an `integer` column; in both cases `mean` calls the default method

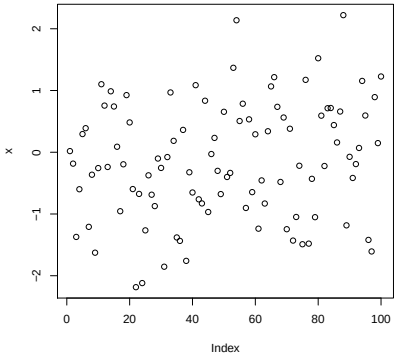
Calling Methods

NOTE: Some methods are visible to the user (i.e. `mean.default`), but you should **never** call methods directly. Rather, use the generic function and let the method be dispatched automatically.

S3 Class/Method: Example 3

The `plot` function is generic and its behavior depends on the object being plotted.

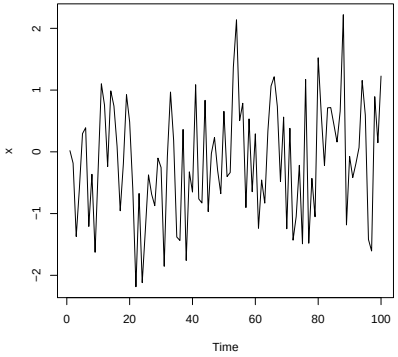
```
> set.seed(10)
> x <- rnorm(100)
> plot(x)
```



S3 Class/Method: Example 3

For time series objects, `plot` connects the dots

```
> set.seed(10)
> x <- rnorm(100)
> x <- as.ts(x)  ## Convert to a time series object
> plot(x)
```



Write your own methods!

If you write new methods for new classes, you'll probably end up writing methods for the following generics:

- print/show
- summary
- plot

There are two ways that you can extend the R system via classes/methods

- Write a method for a new class but for an existing generic function (i.e. like `print`)
- Write new generic functions and new methods for those generics

S4 Classes

Why would you want to create a new class?

- To represent new types of data (e.g. gene expression, space-time, hierarchical, sparse matrices)
- New concepts/ideas that haven't been thought of yet (e.g. a fitted point process model, mixed-effects model, a sparse matrix)
- To abstract/hide implementation details from the user

I say things are "new" meaning that R does not know about them (not that they are new to the statistical community).

S4 Class/Method: Creating a New Class

A new class can be defined using the `setClass` function

- At a minimum you need to specify the name of the class
- You can also specify data elements that are called *slots*
- You can then define methods for the class with the `setMethod` function
- Information about a class definition can be obtained with the `showClass` function

S4 Class/Method: Polygon Class

Creating new classes/methods is usually not something done at the console; you likely want to save the code in a separate file

```
setClass("polygon",
         representation(x = "numeric",
                        y = "numeric"))
```

The slots for this class are `x` and `y`. The slots for an S4 object can be accessed with the `@` operator.

S4 Class/Method: Polygon Class

A `plot` method can be created with the `setMethod` function.

- For `setMethod` you need to specify a generic function (`plot`), and a *signature*.
- A signature is a character vector indicating the classes of objects that are accepted by the method. In this case, the `plot` method will take one type of object–a `polygon` object.

```
setMethod("plot", "polygon",
          function(x, y, ...) {
                  plot(x@x, x@y, type = "n", ...)
                  xp <- c(x@x, x@x[1])
                  yp <- c(x@y, x@y[1])
                  lines(xp, yp)
          })
```

Notice that the slots of the polygon (the x- and y-coordinates) are accessed with the `@` operator.

S4 Class/Method: Polygon Class

Create a new class

```
> setClass("polygon",
+          representation(x = "numeric",
+                         y = "numeric"))
```

Create a plot method for this class

```
> setMethod("plot", "polygon",
+           function(x, y, ...) {
+                   plot(x@x, x@y, type = "n", ...)
+                   xp <- c(x@x, x@x[1])
+                   yp <- c(x@y, x@y[1])
+                   lines(xp, yp)
+           })
```

```
[1] "plot"
```

If things go well, you will not get any messages or errors and nothing useful will be returned by either `setClass` or `setMethod`.

S4 Class/Method: Polygon Class

After calling `setMethod` the new `plot` method will be added to the list of methods for `plot`.

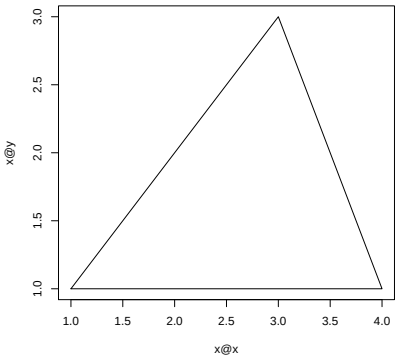
```
> showMethods("plot")
```

```
Function: plot (package graphics)
x="ANY"
x="polygon"
```

Notice that the signature for class `polygon` is listed. The method for `ANY` is the default method and it is what is called when now other signature matches

S4 Class/Method: Polygon class

```
> p <- new("polygon", x = c(1, 2, 3, 4), y = c(1, 2, 3, 1))
> plot(p)
```



Where to Look, Places to Start

- The best way to learn this stuff is to look at examples (and try the exercises for the course)
- There are now quite a few examples on CRAN which use S4 classes/methods.
- Bioconductor (http://www.bioconductor.org) — a rich resource, even if you know nothing about bioinformatics
- Some packages on CRAN (as far as I know) — SparseM, gpclib, flexmix, its, lme4, orientlib, pixmap
- The `stats4` package (comes with R) has a bunch of classes/methods for doing maximum likelihood analysis.