



# Overview and History of R

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# 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.
- 2013: R version 3.0.2 is released on December 2013.

# 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 -> 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**.

# Design of the R System

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>



# Getting Help

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# Asking Questions

- Asking questions via email is different from asking questions in person
- People on the other side do not have the background information you have
  - they also don't know you personally (usually)
- Other people are busy; their time is limited
- The instructor (me) is here to help in all circumstances but may not be able to answer all questions!

# Finding Answers

- Try to find an answer by searching the archives of the forum you plan to post to.
- Try to find an answer by searching the Web.
- Try to find an answer by reading the manual.
- Try to find an answer by reading a FAQ.
- Try to find an answer by inspection or experimentation.
- Try to find an answer by asking a skilled friend.
- If you're a programmer, try to find an answer by reading the source code.

# Asking Questions

- It's important to let other people know that you've done all of the previous things already
- If the answer is in the documentation, the answer will be "Read the documentation"
  - one email round wasted

# Example: Error Messages

```
> library(datasets)
> data(airquality)
> cor(airquality)
Error in cor(airquality) : missing observations in cov/cor
```

# Google is your friend

A screenshot of a Google search results page. The search bar at the top contains the query "missing observations in cov/cor". Below the search bar, it says "About 508,000 results (0.28 seconds)". On the right side of the search bar is a user profile icon.

## [R] Error in cor.default(x1, x2) : missing observations in cov/cor

<https://stat.ethz.ch/pipermail/r-help/2008.../155523.html> - Block stat.ethz.ch

Mar 1, 2008 – [R] Error in cor.default(x1, x2) : missing observations in cov/cor. Daniel Malter daniel at umd.edu. Thu Feb 28 01:40:58 CET 2008. Previous message: [R] Error in ...

[R] Null values In R - Dec 3, 2008

[R] cor(data.frame) infelicities - Dec 3, 2007

[BioC] maanova: missing observations in cov/cor - Aug 25, 2006

[R-sig-Geo] Some comments on calculations with 'asc' (and 'kasc' ...) - Mar 27, 2005

More results from stat.ethz.ch »

## EMBnet 2007 Course - Introduction to Statistics for Biologists

[www.ch.embnet.org/CoursEMBnet/Base07/tp2b.html](http://www.ch.embnet.org/CoursEMBnet/Base07/tp2b.html) - Block ch.embnet.org

you will get an error: Error in cor(thuesen) : missing observations in cov/cor . [You will have seen a similar message above with sd .] This is because of the ...

## R help - Error in cor.default(x1, x2) : missing observations in cov/cor

[r.789695.n4.nabble.com/Error-in-cor-default-x1-x2-missing-...](http://r.789695.n4.nabble.com/Error-in-cor-default-x1-x2-missing-...) - Block

r.789695.n4.nabble.com

13 posts - 5 authors - Feb 27, 2008

Error in cor.default(x1, x2) : missing observations in cov/cor. Hello, I'm trying to do cor(x1,x2) and I get the following error: Error in cor.default(x1, ...

# Asking Questions

- What steps will reproduce the problem?
- What is the expected output?
- What do you see instead?
- What version of the product (e.g. R, packages, etc.) are you using?
- What operating system?
- Additional information

# Subject Headers

- Stupid: "Help! Can't fit linear model!"
- Smart: "R 3.0.2 lm() function produces seg fault with large data frame, Mac OS X 10.9.1"
- Smarter: "R 3.0.2 lm() function on Mac OS X 10.9.1 -- seg fault on large data frame"

# Do

- Describe the goal, not the step
- Be explicit about your question
- Do provide the minimum amount of information necessary (volume is not precision)
- Be courteous (it never hurts)
- Follow up with the solution (if found)

# Don't

- Claim that you've found a bug
- Grovel as a substitute for doing your homework
- Post homework questions on mailing lists (we've seen them all)
- Email multiple mailing lists at once
- Ask others to debug your broken code without giving a hint as to what sort of problem they should be searching for

# Case Study: A Recent Post to the R-devel Mailing List

Subject: large dataset - confused

Message:

I'm trying to load a dataset into R, but  
I'm completely lost. This is probably  
due mostly to the fact that I'm a  
complete R newb, but it's got me stuck  
in a research project.

# Response

Yes, you are lost. The R posting guide is at <http://www.r-project.org/posting-guide.html> and will point you to the right list and also the manuals (at e.g. <http://cran.r-project.org/manuals.html>, and one of them seems exactly what you need).

# Analysis: What Went Wrong?

- Question was sent to the wrong mailing list (R-devel instead of R-help)
- Email subject was very vague
- Question was very vague
- Problem was not reproducible
- No evidence of any effort made to solve the problem
- RESULT: Recipe for disaster!

# Places to Turn

- Class discussion board; your fellow students
- [r-help@r-project.org](mailto:r-help@r-project.org)
- Other project-specific mailing lists (This talk inspired by Eric Raymond's "How to ask questions the smart way")



# Introduction to the R Language

Data Types and Basic Operations

Roger Peng, Associate Professor  
Johns Hopkins Bloomberg School of Public Health

# 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 are 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
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

# 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
> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

# 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

Roger Peng, Associate Professor  
Johns Hopkins Bloomberg School of Public Health

# 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 that 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  $1 \times 1$  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

6/14

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

```
> 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
```



# Reading and Writing Data Part I

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# 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 and 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?

$$1,500,000 \times 120 \times 8 \text{ bytes/numeric}$$

$$= 1440000000 \text{ bytes}$$

$$= 1440000000 / 2^{20} \text{ bytes/MB}$$

$$= 1,373.29 \text{ MB}$$

$$= 1.34 \text{ GB}$$



# Reading and Writing Data Part II

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# 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-ing 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

```
> 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

Roger Peng, Associate Professor  
Johns Hopkins Bloomberg School of Public Health

# 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 iteration of a loop
- `return`: exit a function

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

# 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  
}
```

# if

Of course, the else clause is not necessary.

```
if(<condition1>) {  
}  
  
if(<condition2>) {  
}
```

# for

for loops take an iterator 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
    }
}
```

# repeat

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

# Control Structures

## 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 mentioned here are primarily useful for writing programs; for command-line interactive work, the `*apply` functions are more useful.



# Functions

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# 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)
```

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

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

```
f <- function(a, b) {  
  print(a)  
  print(b)  
}  
f(45)
```

```
## [1] 45
```

```
## Error: 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 :"
```



# Coding Standards for R

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# Coding Standards for R

1. Always use text files / text editor

# Coding Standards for R

1. Always use text files / text editor
2. Indent your code

# Coding Standards for R

1. Always use text files / text editor
2. Indent your code
3. Limit the width of your code (80 columns?)

# Indenting

- Indenting improves readability
- Fixing line length (80 columns) prevents lots of nesting and very long functions
- Suggested: Indents of 4 spaces at minimum; 8 spaces ideal

# Coding Standards for R

1. Always use text files / text editor
2. Indent your code
3. Limit the width of your code (80 columns?)
4. Limit the length of individual functions



# Introduction to the R Language

## Scoping Rules

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# 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 inside 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`, `nls`, 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  $\mu = 1$

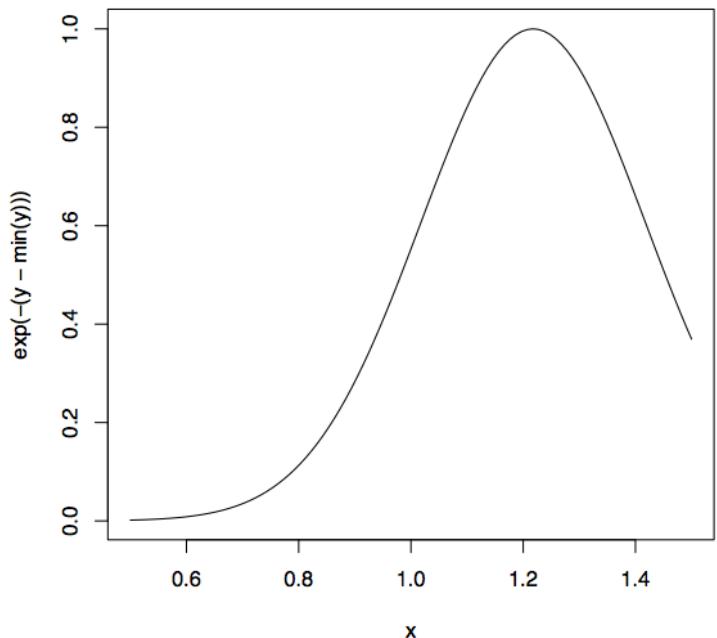
```
> nLL <- make.NegLogLik(normals, c(1, FALSE))
> optimize(nLL, c(1e-6, 10))$minimum
[1] 1.800596
```

# Plotting the Likelihood

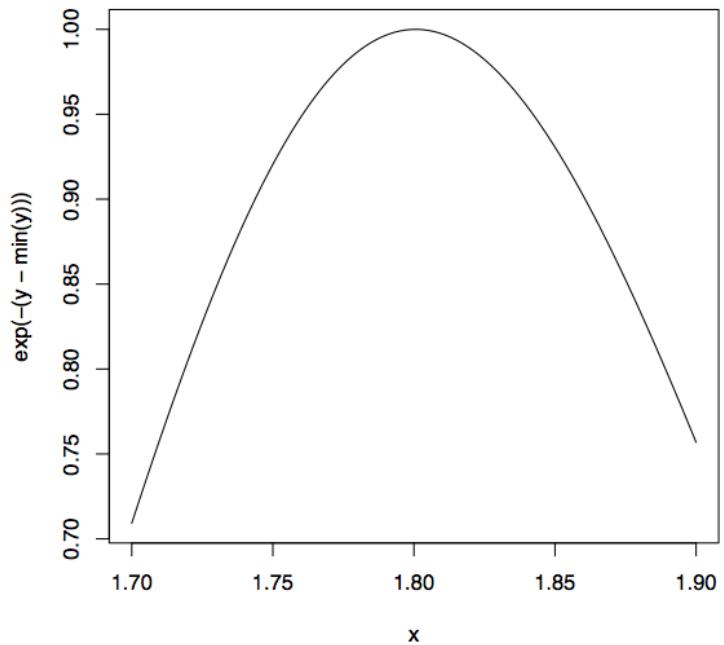
```
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")

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 cleaned up
- Reference: Robert Gentleman and Ross Ihaka (2000). “Lexical Scope and Statistical Computing,” *JCGS*, 9, 491–508.



# Introduction to the R Language

## Vectorized Operations

Roger Peng, Associate Professor  
Johns Hopkins Bloomberg School of Public Health

# 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
```



# Dates and Times in R

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# Dates and Times in R

R has developed a special representation of dates and times

- Dates are represented by the `Date` class
- Times are represented by the `POSIXct` or the `POSIXlt` class
- Dates are stored internally as the number of days since 1970-01-01
- Times are stored internally as the number of seconds since 1970-01-01

# Dates in R

Dates are represented by the Date class and can be coerced from a character string using the `as.Date()` function.

```
x <- as.Date("1970-01-01")
x
## [1] "1970-01-01"
unclass(x)
## [1] 0
unclass(as.Date("1970-01-02"))
## [1] 1
```

# Times in R

Times are represented using the `POSIXct` or the `POSIXlt` class

- `POSIXct` is just a very large integer under the hood; it uses a useful class when you want to store times in something like a data frame
- `POSIXlt` is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month

There are a number of generic functions that work on dates and times

- `weekdays`: give the day of the week
- `months`: give the month name
- `quarters`: give the quarter number (“Q1”, “Q2”, “Q3”, or “Q4”)

# Times in R

Times can be coerced from a character string using the `as.POSIXlt` or `as.POSIXct` function.

```
x <- Sys.time()
x
## [1] "2013-01-24 22:04:14 EST"
p <- as.POSIXlt(x)
names(unclass(p))
## [1] "sec"      "min"      "hour"     "mday"     "mon"
## [6] "year"     "wday"     "yday"     "isdst"
p$sec
## [1] 14.34
```

# Times in R

You can also use the `POSIXct` format.

```
x <- Sys.time()
x  ## Already in 'POSIXct' format
## [1] "2013-01-24 22:04:14 EST"
unclass(x)
## [1] 1359083054
x$sec
## Error: $ operator is invalid for atomic vectors
p <- as.POSIXlt(x)
p$sec
## [1] 14.37
```

# Times in R

Finally, there is the `strptime` function in case your dates are written in a different format

```
datestring <- c("January 10, 2012 10:40", "December 9, 2011  
x <- strptime(datestring, "%B %d, %Y %H:%M")  
x  
## [1] "2012-01-10 10:40:00" "2011-12-09 09:10:00"  
class(x)  
## [1] "POSIXlt" "POSIXt"
```

I can *never* remember the formatting strings. Check `?strptime` for details.

# Operations on Dates and Times

You can use mathematical operations on dates and times. Well, really just + and -. You can do comparisons too (i.e. ==, <=)

```
x <- as.Date("2012-01-01")
y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
x-y
## Warning: Incompatible methods ("-.Date",
## "-.POSIXt") for "-"
## Error: non-numeric argument to binary operator
x <- as.POSIXlt(x)
x-y
## Time difference of 356.3 days
```

# Operations on Dates and Times

Even keeps track of leap years, leap seconds, daylight savings, and time zones.

```
x <- as.Date("2012-03-01") y <- as.Date("2012-02-28")
x-y
## Time difference of 2 days
x <- as.POSIXct("2012-10-25 01:00:00")
y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
y-x
## Time difference of 1 hours
```

# Summary

- Dates and times have special classes in R that allow for numerical and statistical calculations
- Dates use the `Date` class
- Times use the `POSIXct` and `POSIXlt` class
- Character strings can be coerced to Date/Time classes using the `strptime` function or the `as.Date`, `as.POSIXlt`, or `as.POSIXct`



# Introduction to the R Language

## Loop Functions

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# 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: (1) a list `x`; (2) a function (or the name of a function) `FUN`; (3) 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))  
## }  
## <bytecode: 0x7ff7a1951c00>  
## <environment: namespace:base>
```

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] 0.4671
```

# 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] 0.5261
##
## $c
## [1] 1.421
##
## $d
## [1] 4.927
```

# 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,]
```

# 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
```



# Introduction to the R Language

Loop Functions - apply

Roger Peng, Associate Professor  
Johns Hopkins Bloomberg School of Public Health

# apply

`apply` is used to 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]
```

# apply

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
```



# Introduction to the R Language

## Loop Functions - tapply

Roger Peng, Associate Professor  
Johns Hopkins Bloomberg School of Public Health

# 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 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
```

# 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
```

# split

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
```



# Introduction to the R Language

## Loop Functions - tapply

Roger Peng, Associate Professor  
Johns Hopkins Bloomberg School of Public Health

# 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 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
```

# 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
```

# split

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
```



# Introduction to the R Language

## Loop Functions - mapply

Roger Peng, Associate Professor  
Johns Hopkins Bloomberg School of Public Health

# 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

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# 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)
```

```
## Warning: NaNs produced
```

```
## [1] NaN
```

# 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: 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: 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:
```

# Debugging

## 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

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

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

# Generating Random Numbers

```
> 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

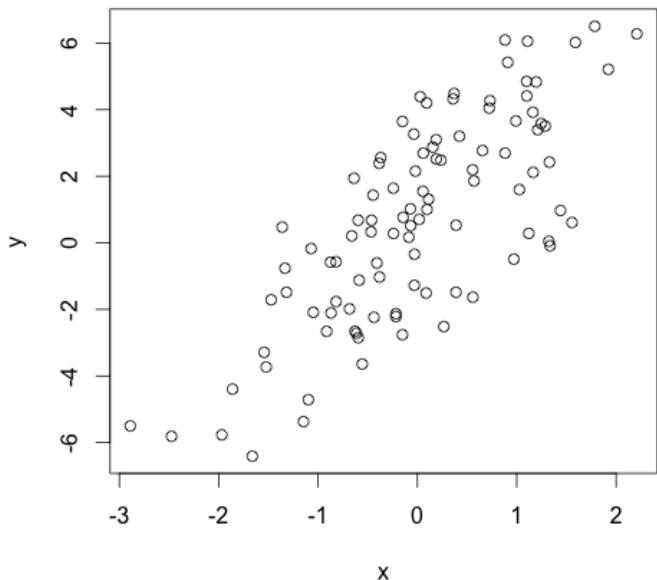
Suppose we want to simulate from the following linear model

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

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
-6.4080 -1.5400  0.6789  0.6893  2.9300  6.5050
> plot(x, y)
```

# Generating Random Numbers From a Linear Model

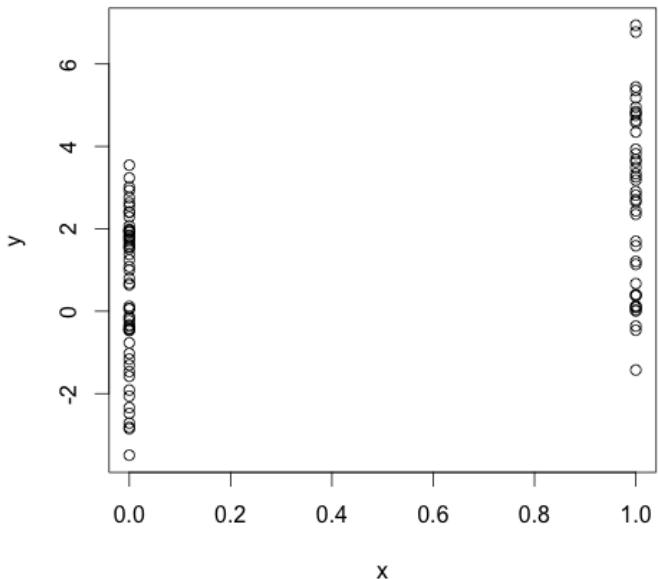


# Generating Random Numbers From a Linear Model

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
-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

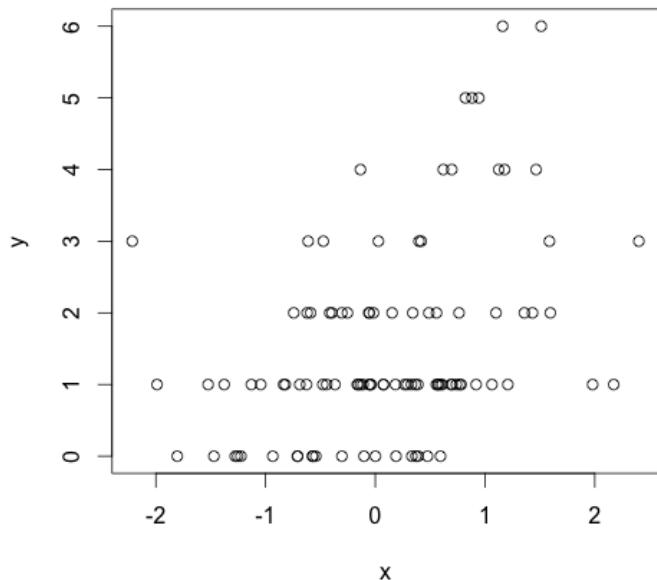
$$Y \sim \text{Poisson}(\mu)$$

$$\log \mu = \beta_0 + \beta_1 x$$

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
```

# Simulation

## 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



# Profiling R Code

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# Why is My Code So Slow?

- Profiling is a systematic way to examine how much time is spent in different parts of a program
- Useful when trying to optimize your code
- Often code runs fine once, but what if you have to put it in a loop for 1,000 iterations? Is it still fast enough?
- Profiling is better than guessing

# On Optimizing Your Code

- Getting biggest impact on speeding up code depends on knowing where the code spends most of its time
- This cannot be done without performance analysis or profiling

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil

--Donald Knuth

# General Principles of Optimization

- Design first, then optimize
- Remember: Premature optimization is the root of all evil
- Measure (collect data), don't guess.
- If you're going to be scientist, you need to apply the same principles here!

# Using `system.time()`

- Takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression
- Computes the time (in seconds) needed to execute an expression
  - If there's an error, gives time until the error occurred
- Returns an object of class `proc_time`
  - **user time**: time charged to the CPU(s) for this expression
  - **elapsed time**: "wall clock" time

# Using `system.time()`

- Usually, the user time and elapsed time are relatively close, for straight computing tasks
- Elapsed time may be *greater than* user time if the CPU spends a lot of time waiting around
- Elapsted time may be *smaller than* the user time if your machine has multiple cores/processors (and is capable of using them)
  - Multi-threaded BLAS libraries (vecLib/Accelerate, ATLAS, ACML, MKL)
  - Parallel processing via the **parallel** package

# Using system.time()

```
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
  user  system elapsed
0.004   0.002   0.431

## Elapsed time < user time
hilbert <- function(n) {
  i <- 1:n
  1 / outer(i - 1, i, "+")
}
x <- hilbert(1000)
system.time(svd(x))
  user  system elapsed
1.605   0.094   0.742
```

# Timing Longer Expressions

```
system.time({  
  n <- 1000  
  r <- numeric(n)  
  for (i in 1:n) {  
    x <- rnorm(n)  
    r[i] <- mean(x)  
  }  
})
```

```
##    user  system elapsed  
##  0.097   0.002   0.099
```

# Beyond `system.time()`

- Using `system.time()` allows you to test certain functions or code blocks to see if they are taking excessive amounts of time
- Assumes you already know where the problem is and can call `system.time()` on it
- What if you don't know where to start?

# The R Profiler

- The `Rprof()` function starts the profiler in R
  - R must be compiled with profiler support (but this is usually the case)
- The `summaryRprof()` function summarizes the output from `Rprof()` (otherwise it's not readable)
- DO NOT use `system.time()` and `Rprof()` together or you will be sad

# The R Profiler

- `Rprof()` keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent in each function
- Default sampling interval is 0.02 seconds
- NOTE: If your code runs very quickly, the profiler is not useful, but then you probably don't need it in that case

# R Profiler Raw Output

```
## lm(y ~ x)

sample.interval=10000

"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"lm.fit" "lm"
"lm.fit" "lm"
"lm.fit" "lm"
```

# Using `summaryRprof()`

- The `summaryRprof()` function tabulates the R profiler output and calculates how much time is spent in which function
- There are two methods for normalizing the data
  - "by.total" divides the time spent in each function by the total run time
  - "by.self" does the same but first subtracts out time spent in functions above in the call stack

# By Total

```
$by.total
```

	total.time	total.pct	self.time	self.pct
"lm"	7.41	100.00	0.30	4.05
"lm.fit"	3.50	47.23	2.99	40.35
"model.frame.default"	2.24	30.23	0.12	1.62
"eval"	2.24	30.23	0.00	0.00
"model.frame"	2.24	30.23	0.00	0.00
"na.omit"	1.54	20.78	0.24	3.24
"na.omit.data.frame"	1.30	17.54	0.49	6.61
"lapply"	1.04	14.04	0.00	0.00
"[.data.frame"	1.03	13.90	0.79	10.66
"[ "	1.03	13.90	0.00	0.00
"as.list.data.frame"	0.82	11.07	0.82	11.07
"as.list"	0.82	11.07	0.00	0.00

# By Self

\$by.self	self.time	self.pct	total.time	total.pct
"lm.fit"	2.99	40.35	3.50	47.23
"as.list.data.frame"	0.82	11.07	0.82	11.07
"[ .data.frame"	0.79	10.66	1.03	13.90
"structure"	0.73	9.85	0.73	9.85
"na.omit.data.frame"	0.49	6.61	1.30	17.54
"list"	0.46	6.21	0.46	6.21
"lm"	0.30	4.05	7.41	100.00
"model.matrix.default"	0.27	3.64	0.79	10.66
"na.omit"	0.24	3.24	1.54	20.78
"as.character"	0.18	2.43	0.18	2.43
"model.frame.default"	0.12	1.62	2.24	30.23
"anyDuplicated.default"	0.02	0.27	0.02	0.27

# summaryRprof() Output

```
$sample.interval
```

```
[1] 0.02
```

```
$sampling.time
```

```
[1] 7.41
```

# Summary

- `Rprof()` runs the profiler for performance analysis of R code
- `summaryRprof()` summarizes the output of `Rprof()` and gives percent of time spent in each function (with two types of normalization)
- Good to break your code into functions so that the profiler can give useful information about where time is being spent
- C or Fortran code is not profiled



# Introduction to the R Language

## Scoping Rules

Roger D. Peng, Associate Professor of Biostatistics  
Johns Hopkins Bloomberg School of Public Health

# 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 inside 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`, `nls`, 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  $\mu = 1$

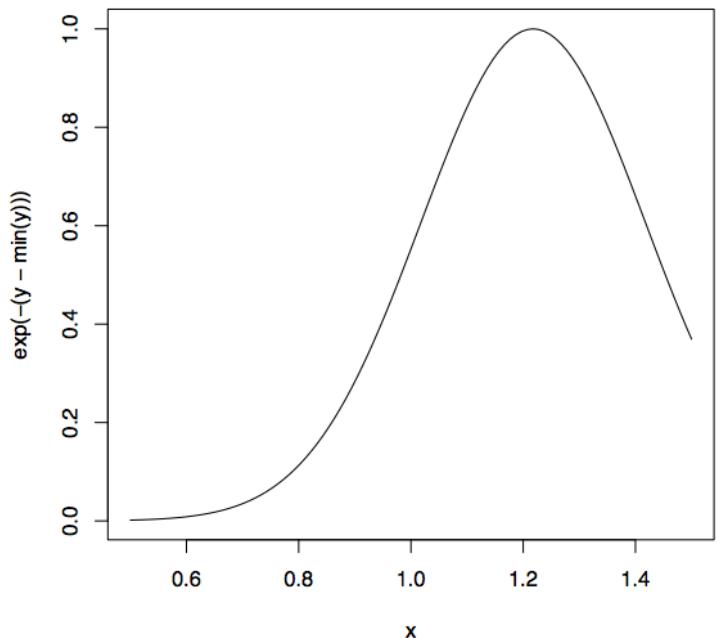
```
> nLL <- make.NegLogLik(normals, c(1, FALSE))
> optimize(nLL, c(1e-6, 10))$minimum
[1] 1.800596
```

# Plotting the Likelihood

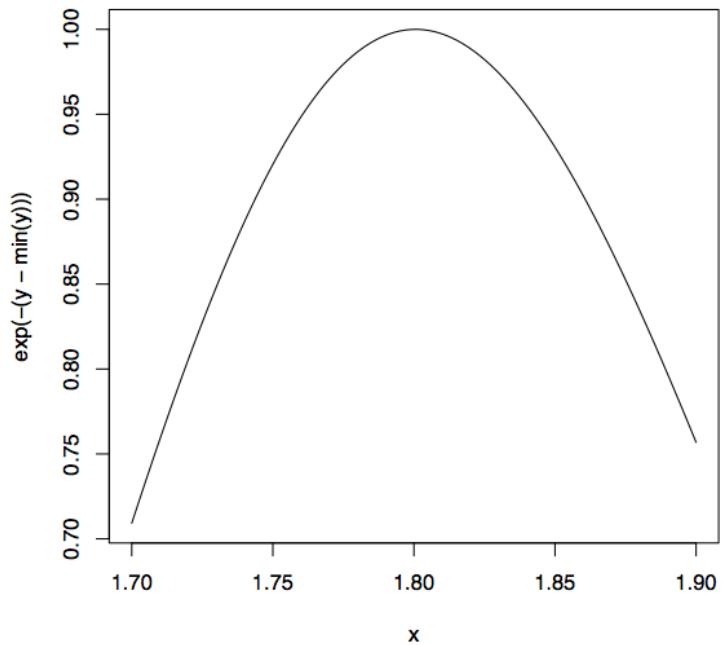
```
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")

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 cleaned up
- Reference: Robert Gentleman and Ross Ihaka (2000). “Lexical Scope and Statistical Computing,” *JCGS*, 9, 491–508.