Introduction to R

Dason Kurkiewicz, Chris Bruno, Adam Loy

Outline

- Intro
- Data Management
- Basic Graphics
- Models
- Programming in R
- Advanced Data Manipulations

1 - Intro to R

Good Habits and Basic Introduction

Outline

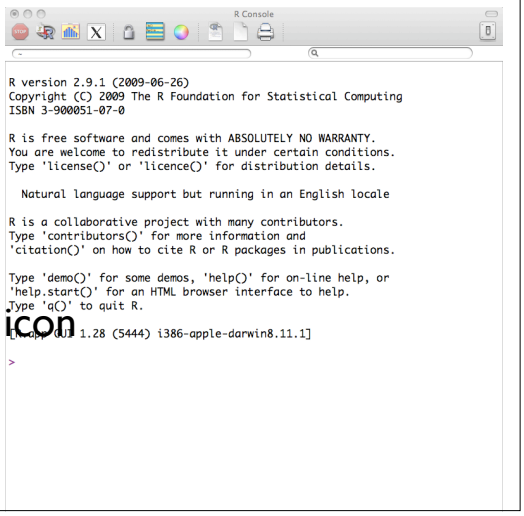
- Getting Started
- Good Habits
- Overgrown calculator
- Basic functions
- Getting help
- Doing statistics
- More...

What do you know already?

- Excel? JMP? Numbers?
- A programming language? CS course?
- R? SAS? SPSS?
- Have you used a text editor?

Install R

- On your own machine:
  - Go to http://www.r-project.org/
  - From CRAN, pick download site (ISU might be good)
  - Download from base:
    - R-2.11.1-win32.exe
    - Run the installation script
- On a lab machine:
  - Start R by double-clicking the icon



Setting up

- Open the R script in an editor, eg notepad, wordpad, emacs, ...
- Edit lines
- Cut and paste lines of code into the R interpreter window
- Or:
  - Windows: Ctrl-r
  - Mac: Apple-return
  - Linux Using Rkward: Shift-F7; Using Emacs w/ ESS: Ctrl-c n

Good Habits

- Avoid the interpreter.
- Keep a record of your work by using scripts.
- Makes coding easier
- Store your data and your scripts in a convenient location.
- Save often

- Working Directory

Navigating the R interpreter window

- Up/down arrow keys to retrieve previous lines
- Left/right arrow keys to move cursor along line
- Mouse click to set cursor position
- Delete to remove and re-type parts of command

Learning a language

- Grammar / Syntax
- Vocabulary
- "Thinking in that language"
- There are a lot of commands in R. Don't expect to memorize all of them.

Overgrown Calculator

- Basic mathematical operators:
  - +, -, *, /
- Basic Mathematical functions
  - exp, log, sin, cos
- Storing variables for later use using the assignment operator
- `a <- 32`
- Working with vectors . . .

Variables

- Variable names can't start with a number
- R is case-sensitive
- Some common letters are used internally by R and should be avoided as variable names `(c, q, t, C, D, F, T, I)`
- Try to keep names short but descriptive.
- There are reserved words that R won't let you us for variable names. A few examples:
  - `for, in, while, if, else, repeat, break`
- R will let you use the name of a predefined function. So try to not over write those!

Basics

- Basic algebra is the same
- Use `2*x` not 2x, `2^p` instead of 2^p
- Applying a function is similar
- Making a variable, use `<-` instead of =
- Everything in R is a vector
- Index a vector using [ ]

Examples

• $x = 2 / 3$	`x <- 2 / 3`
• $\sqrt{x}$	`sqrt(x)`
• $a = 2(x + 3)^2$	`a <- 2 *(x + 3)^2`
• $y = (1\ 2\ 3\ 5)^T$	`y <- c(1, 2, 3, 5)`
• y_1	`y[1]`
• $\sum y$	`sum(y)`
• 2y	`2*y`
• get the date	`date()`

...

Functions

- Typical format:
  - `foo(x, y = 1:length(x), ...)`
  - Some parameters have defaults set for you
  - `...` is special. Passes along extra parameters to functions used inside the function.

Getting Help

- `help.start()`
- `help(command)`
- `?command`
- `help.search("command")`
- `apropos()`
- Google!

Getting Out

- `q()`

Your Turn

- x = 3
- y = 5
- Square root of $(x^2 + y^5)$
- $\sin(e^{(2(pi)(x+y)/(y-x))})$
- Find the roots of $3t^2 - 2t - 17$

R Reference Card

- Download the R Reference Card from http://cran.r-project.org/doc/contrib/Short-refcard.pdf
- Open/Print so that you can glance at it while working

Vectors

- As mentioned before almost everything in R is a vector.
- Multiple ways to make a vector
- `c(1,2,3)`
- `a:b` creates a vector (a, a+1, ..., b-1, b)
- Common operations and functions work elementwise on vectors and return a new vector.
- `rep() and seq()` ...

Your Turn

- $x = (4\ 1\ 3\ 9)^T$
- $y = (1\ 2\ 3\ 5)^T$ (from examples, on previous slide)
- $d = \sqrt{\sum(x - y)^2}$
- $2(y_1 + x_2)$
- $z = (1, ..., 100)^T$
- $(\sum z_i)^2 - \sum(z_i^2)$
- pattern = $(1, 7, 7, 13, 13, 13, 19, 19, 19, 19, 25, 25, 25, 25, 25, 31, 31, 31, 31, 31, 31)^T$ (don't use `c()` for this)

Basic Statistical Functions

- Using the basic functions we've learned it wouldn't be hard to compute basic statistics.
- `x <- 1:100`
- `n <- length(x)`
- `xbar <- sum(x)/n`
- `s <- sqrt(sum((x-xbar)^2)/(n-1))`
- ... But we don't have to

Distributions

- R has a lot of distributions built in.
- We can typically obtain:
  - Density value
  - CDF value
  - Inverse CDF value
  - Random deviate
- Normal, Chi-square, F, T, Cauchy, Poisson, Binomial, Negative Binomial, Gamma, ..., lots more
- `library(help = stats)`

Your Turn

- Find the mean of 10, 1000, and 10000 random normal observations from N(0,1)
- Generate 1,000,000 random observations from N(0,1), square them, and find the .95 quantile of the observed data
- What is the .95 quantile from a Chi-square distribution with 1 df?
- x = 100 random normal observations from N(5,36)
- Calculate a 95% confidence interval for mu using x as your data [xbar ± $(t_{.95,99})(s)$]

Booleans

- R has support for logical values
- `TRUE, FALSE, T, F`
- Can result from a comparison
  - `<`
  - `>`
  - `<=`
  - `>=`
  - `==`
  - `!=`

Logical Operators

- `&` (AND)
- `|` (OR)
- Slightly different from:
- `&&` (different AND)
- `||` (different OR)
- ?"&"

Indexing

- Accessing just a part of a vector/matrix/dataframe.
- Multiple ways to index
  - `x[2]`
  - `x[c(1,3,7)]`
  - `x[c(T,F)]`
  - `x[x>10]`
  - `x[-1]`

Your Turn

- Using `pat <- seq(2,103,by = 3)`
- x = elements at even indices in pat
- y = elements in pat greater than mean(pat)
- z = even elements in pat
- prime = All primes between 1 and 100 (setdiff might be of interest)

Load Data

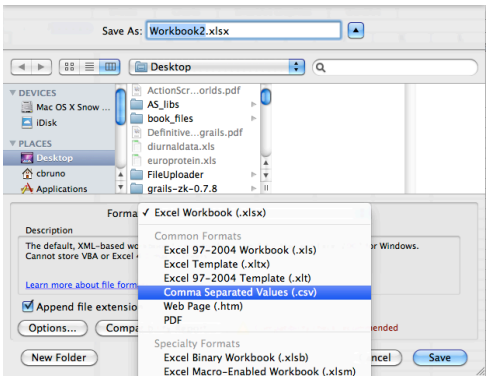
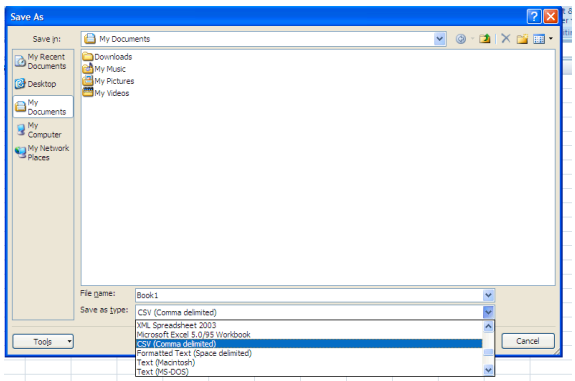
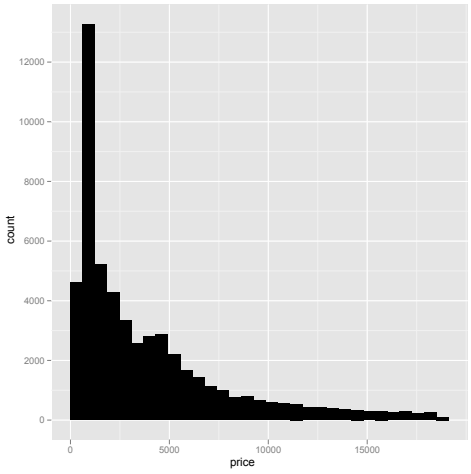
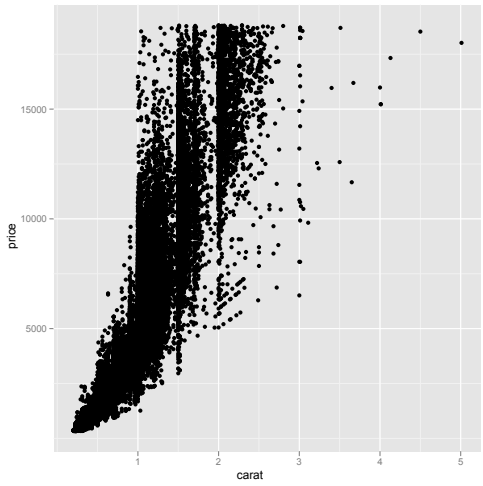
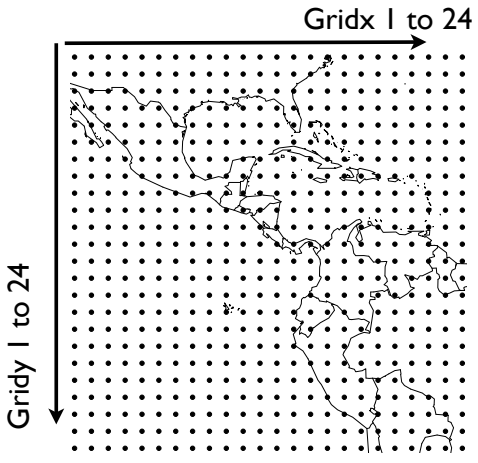
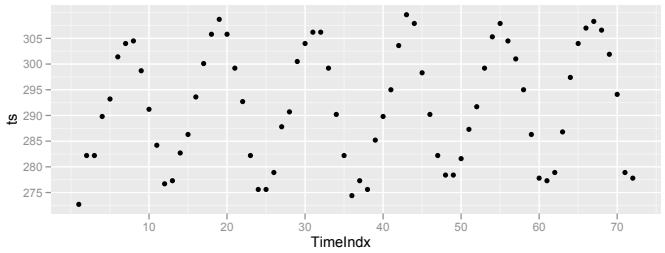
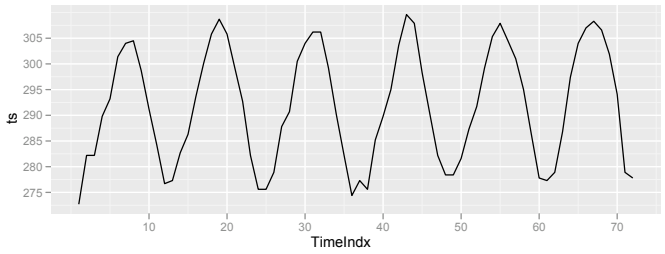
- ```
  >library(ggplot2)
  >data()
  >help(tips)
  ```

- Did the data import work?

Examining Objects

- `x`
- `head(x)`
- `summary(x)`
- `str(x)`
- `dim(x)`

Try these commands out for yourself!
  ```

Examine Object

- First few values of an object
  `head(tips)`

```
> head(tips)
  obs totbill  tip sex smoker day  time size
1   1   16.99 1.01   F     No Sun Night    2
2   2   10.34 1.66   M     No Sun Night    3
3   3   21.01 3.50   M     No Sun Night    3
4   4   23.68 3.31   M     No Sun Night    2
5   5   24.59 3.61   F     No Sun Night    4
6   6   25.29 4.71   M     No Sun Night    4
```


Examine Object

- Structure of an object
  `str(tips)`

```
> str(tips)
'data.frame':   244 obs. of  8 variables:
 $ obs    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ totbill: num  17 10.3 21 23.7 24.6 ...
 $ tip    : num  1.01 1.66 3.5 3.31 3.61 4.71 2 3.12 1.96 3.23 ...
 $ sex    : Factor w/ 2 levels "F","M": 1 2 2 2 1 2 2 2 2 2 ...
 $ smoker : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
 $ day    : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ time   : Factor w/ 2 levels "Day","Night": 2 2 2 2 2 2 2 2 2 2 ...
 $ size   : int  2 3 3 2 4 4 2 4 2 2 ...
```

Examine Object

- Dimension of an object
  `dim(tips)`

```
> dim(tips)
[1] 244   8
```

the tips data set has 244 rows (tables served) and 8 columns (variables recorded)

Examine Object

- Dimension of an object
  `summary(tips)`

```
> summary(tips)
      obs            totbill           tip         sex     smoker
 Min.   :  1.00   Min.   : 3.07   Min.   : 1.000   F: 87   No :151
 1st Qu.: 61.75   1st Qu.:13.35   1st Qu.: 2.000   M:157   Yes: 93
 Median :122.50   Median :17.80   Median : 2.900
 Mean   :122.50   Mean   :19.79   Mean   : 2.998
 3rd Qu.:183.25   3rd Qu.:24.13   3rd Qu.: 3.562
 Max.   :244.00   Max.   :50.81   Max.   :10.000
  day        time          size
 Fri:19   Day  : 68   Min.   :1.000
 Sat:87   Night:176   1st Qu.:2.000
 Sun:76               Median :2.000
 Thu:62               Mean   :2.570
                      3rd Qu.:3.000
                      Max.   :6.000
```

Extracting parts

- `x$variable`
- `x[, "variable"]`
- `x[rows, columns]  # rows, columns are # indices`
  - `x[1:5, 2:3]`
  - `x[c(1,5,6), c("sex","tip")]`
- `x$variable[rows]`

Your Turn

- Calculate basic summary stats for the variables.
- Create a variable for Tipping Rate
- Find the mean bill amount for each gender
- Are there any unusual points?
- Explore the data. Can you find any interesting trends?
- How many people in this data tipped tipped greater than 20% of their bill?

Introduction to R

Data Management

Data structures in R

- The primary data structures in R include the following:

  - Vectors
  - Lists
  - Data frames
  - Objects - everything in R is an Object

Data structures in R: Vectors

• Vectors are a collection of data of the same type. The data can be integers, floats (decimal numbers), complex numbers, text/strings, or logical values.

Example: a vector of floats: 2.34 3.20 4.55 10.24 30.12 7.14 3.68

• The *c()* function: The most common method of creating vector is to use the *c()* function. The *c()* function *combines* the data into a vector.

Example: `myVector <- c(2, 3, 4, 5)`

Now the object myVector consists of the data 2, 3, 4, 5.

• Since the members of a vector must all be of the same type, the c() function forces all data to be of the same type

Example 1: `myVector <- c(2, 3.15, 4.2, 6)`

The object myVector consists of the data 2.00, 3.15, 4.20, 6.00 (all floats)

Example 2: `myVector <- c(2, "hello", 4.69, 8)`

The object myVector consists of the data "2", "hello", "4.69", "8" (all strings)

Data structures in R: Vectors

• The seq(from, to, by) function can also be used to create vectors:

This function takes the arguments "from" (the starting value), "to" (the ending value) and "by" (increment value) and returns the corresponding vector of values.

Example: myVector <- seq(from=2, to=100, by=2)

myVector consists of all the even numbers from 2 to 100.

• A shorthand for the seq() function when you want to increment by 1 is a follows:

Example: myVector <- 1:50

myVector consists of numbers 1 through 50.

Data structures in R: Vectors

- To address a value in a vector, use brackets [ ] :

Example:

```
myVector <- c(2, 4, 8, 7, 10)
first_item <- myVector[1]
```

first_item stores the value 2. Similarly, myVector[2] and myVector[5] retrieve the values 4 and 10, respectively.

- Note that vectors in R are one-based (start at 1), unlike other programming languages (such as C or Java) whose array data structures are zero-based and start at 0.

- To determine the length of a vector, use the length() function.

Example:

```
myVector <- seq(from=5, to=100, by=5)
myVector_length <- length(myVector)
```

myVector_length stores the value 20.

Vectors

Your Turn:

From the R command line, create a vector of 10 numbers.

Use the summary statisitical functions such as
sum, mean, sd, min, and max
on the vector.

Use bracket notation to address elements in a vector.

Data structures in R: Lists

● Lists are a collection of R objects. Unlike vectors, lists can contain any R object and the objects do not have to be the same type. To create a list, use the list() function:

Example:
```
myList <- list(2, 3.5, "hello", c(2, 4, 5))
```

myList now consists of an integer 2, a numeric 3.5, a string "hello" and a vector with data values 2, 4, 5.

● Like vectors, the objects of a list can be addressed using brackets. Lists are also one-based:

Example:
```
myList <- list(2, 3.5, "hello", c(2, 4, 5))
myList_third <- myList[3]
```

myList_third stores the value "hello"

Data structures in R: Lists

● In addition to addressing objects using brackets, lists support the naming of objects, and addressing the objects by their name.

Example:
```
myList <- list(name="John Doe", age=25, position="electrician",
      clients=c("Walmart", "Target", "Best Buy"))

      myList$name
      [1] "John Doe"
      myList$age
      [1] 25
      myList$clients
      [1] "Walmart"  "Target"   "Best Buy"
```

Lists

Your Turn:

From the R command line, create a list of 5 objects.

Address the objects in the list using bracket notation and the name.

Convert a list to a vector using the as.vector function

Data structures in R: Dataframes

● Tabular data sets in R are stored as dataframes. The easiest way to think about dataframes is that they are a list of vectors. To construct a dataframe, use the data.frame() function. This function is similar to the list() function.

Example usage:

```
myData <- data.frame(col1=c(2, 4, 3), col2=c(4, 7, 8), ...)
```

The read.table() function returns a dataframe.

Data structures in R: Dataframes

- To set the column names of a dataframe, use the colnames() function.

Example:

```
> myData
  x y z
1 2 3 5
2 3 4 5
3 5 2 4
> colnames(myData) <- c("column1", "column2", "column3")
> myData
  column1 column2 column3
1       2       3       5
2       3       4       5
3       5       2       4
```


Data structures in R: Dataframes

- To address the columns of a dataframe, use either brackets or the column name (as done to address list objects). Returns a vector.

Example:

```
> myData
  column1 column2 column3
1       2       3       5
2       3       4
3       5       2       4
> myData$column1 #using column names
[1] 2 3 5
> myData[,1] #using brackets
[1] 2 3 5
```

To address rows, use brackets or names. You can specify row names using the rownames() function, in the same way you use the colnames() function.

Data structures in R: Dataframes

- You can also grab out portions of a dataframe using the subset function. The general format is

```
subset(dataframe, condition)
```

For example, suppose you have a dataframe that has a "Sex" column with entries either "M" (male) or "F" (female). To get a dataframe that includes only the "F" (female) rows, use subset as follows:

```
> myData
  Age Sex
1  33   M
2  24   F
3  31   F
4  26   M
5  45   M
> myData_females <- subset(myData, Sex=="F")
#note the double "=" to denote equality as opposed to assignment
> myData_females
  Age Sex
2  24   F
3  31   F
```


Dataframes

Your Turn:

From the R command line, use read.csv() to read in the Airfares.csv data set. Note that this dataset **does** use headers. (download from the workshop website)

Use the head() function to see the first several rows of the data.

Address the FARE column and calculate the mean fare and the standard deviation of fare rates.

Other values

- *NA*
- *Inf*, *-Inf*, *NaN*
- *Inf + x = Inf*
- …
- `is.finite()`
- `is.nan()`
- If you ever test if something is NaN you MUST use

Missing values

- `NA + x = NA, NA * x = NA`
- `x == NA`
- `is.na` returns logical vector, for single vector
- `complete.cases` does the same for a data.frame
- Many functions have parameter `na.rm`

Factors

- A special type of numeric (integer) data
- Numbers + labels
- Used for categorical variables
- On import, make sure numeric categorical variables are converted to factors
- factor creates a new factor with specified labels

Factors

- factor variables often have to be re-ordered for ease of comparisons
- We can specify the order of the levels by explicitly listing them, see help(factor)
- We can make the order of the levels in one variable dependent on the summary statistic of another variable, see help(reorder)

Checking for, and casting between types

- str, mode provide info on type
- is.XXX (with XXX either factor, int, numeric, logical, character, ...) checks for specific type
- as.XXX casts to specific type

Objects

- `ls(), rm()`
- `mode(), str()`
- `dim()`
- Changing data type

Examining Objects

- `x`
- `head(x)`
- `summary(x)`
- `str(x)`
- `dim(x)`

Examining Objects

- `head(tips)`

```
  total_bill  tip    sex smoker day   time size
1      16.99 1.01 Female     No Sun Dinner    2
2      10.34 1.66   Male     No Sun Dinner    3
3      21.01 3.50   Male     No Sun Dinner    3
4      23.68 3.31   Male     No Sun Dinner    2
5      24.59 3.61 Female     No Sun Dinner    4
6      25.29 4.71   Male     No Sun Dinner    4
```

Examining Objects

- `str(tips)`

```
'data.frame': 244 obs. of  7 variables:
$ total_bill: num  17 10.3 21 23.7 24.6 ...
$ tip       : num  1.01 1.66 3.5 3.31 3.61 4.71 2 3.12 1.96 3.23 ...
$ sex       : Factor w/ 2 levels "Female","Male": 1 2 2 2 1 2 2 2 2 2 ...
$ smoker    : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
$ day       : Factor w/ 4 levels "Fri","Sat","Sun",..: 3 3 3 3 3 3 3 3 3 3 ...
$ time      : Factor w/ 2 levels "Dinner","Lunch": 1 1 1 1 1 1 1 1 1 1 ...
$ size      : int  2 3 3 2 4 4 2 4 2 2 ...
```


Examining Objects

- `dim(tips)`

```
[1] 244   7
```

Examining Objects

- `summary(tips)`

```
total_bill         tip             sex      smoker      day
Min.   : 3.07   Min.   : 1.000   Female: 87   No :151   Fri :19
1st Qu.:13.35   1st Qu.: 2.000   Male  :157   Yes: 93   Sat :87
Median :17.80   Median : 2.900                          Sun :76
Mean   :19.79   Mean   : 2.998                          Thur:62
3rd Qu.:24.13   3rd Qu.: 3.562
Max.   :50.81   Max.   :10.000
    time          size
Dinner:176   Min.   :1.000
Lunch : 68   1st Qu.:2.000
             Median :2.000
             Mean   :2.570
             3rd Qu.:3.000
             Max.   :6.000
```


Data formats

- Tabular Flat Files

  Data store in a table, consisting of columns and rows. These include spreadsheets and delimited text files.

  Examples: Excel spreadsheets (.xls or .xlsx), Comma separated variables (.csv) and tab-delimited text files (usually .txt or .dat)

- Relational databases

  A collection of tables, each having one column identified as a key variable. Tables are related to each other through these keys.

  Examples: Microsoft Access, database software such as MySQL and Oracle.

  R can interface to a relational database such as MySQL. However, all data must be converted to a tabular format for use in R.

Common formats for flat files

- Text files

  Data stored as plain text is the simplest and preferred format. Rows are given stored on a line and columns are separated by a delimiter. Common delimiters include: commas (.csv), tabs, semicolons, and ampersands (.txt or .dat).

  In this seminar we will show how to import a text file into R.

- Excel spreadsheets

  Microsoft Excel default file format is either a binary (.xls) format or xml (.xlsx) format. This is necessary to support the formatting and formula functionality of Excel.

  R does not currently support the direct import of Excel files. Instead, data in an Excel spreadsheet can be saved as a comma delimited text file, by selecting the option under the "File->Save As" menu.

Common formats for flat files





- File formats of proprietary statistical software

  This includes the data file formats of statistical software such as SPSS (.sav), SAS (.ssd), Stata (.dta), & Minitab (.mtp). Using the "foreign" package, R supports the direct import of these data files using the read.*x*() function, where *x* is the file extension of the file format to be imported.

Importing text files into R

- read.table(file, header, sep, ...)

  This function can import any delimited text file into R.

  "file" is a string that is either the path to the file on your hard drive, or a URL of a file available over the internet.

  "header" can be True or False, and denotes whether the first line of the text file is variable (column) names or not.

  "sep" is a 1 character string that denotes the delimiter used in the text file.

- read.csv(file, header, ...)

  This is a convenience function that uses read.table to specifically import comma delimited files.

Importing text files into R

- scan(file, what, n, sep, ...)

  This reads in a list of numbers from a file (as opposed to a table of columns and rows).

  "file" is the path or URL of the file

  "what" denotes how the type of data being read. The default is double(0). Or supported types are logical, integer, numeric, complex, character, and raw.

  "n" is the maximum number of data values to be read

  "sep" is the delimiter

  scan() can also be used to read in values from the R console by entering stdin() for the file argument. Values can then be entered at the R console. To finish entering data values, leave a line blank and press enter.

Importing text files into R

Your Turn:

Read in a csv file:

```
csv_file <- read.csv(file.choose(), header=TRUE)
```

read in a tab-delimited text file:

```
tab_file <- read.table(file.choose(), header=TRUE, sep='\t')
```

read in a semi-colon delimited text file:

```
sc_file <- read.table(file.choose(), header=TRUE, sep=';')
```

Introduction to R

Basic Graphics

ggplot2

- We will use the ggplot2 package for graphics.
- Whenever you first want to plot during an R session you will need to enter the command:

```
library(ggplot2)
```

- If you do not have ggplot installed, you can install the package in the R session by entering the command:

```
install.packages("ggplot2")
```

- A reference manual can be found on Hadley Wickham's website http://had.co.nz/ggplot2/

qplot

- The workhorse function for basic plots in ggplot2 is qplot and has basic usage

```
qplot(x, ..., data, geom)
```

  - x: values of one variable of interest
  - data: data frame
  - geom: the type of plot to construct
- For additional help and examples use the command

```
?qplot
```


Diamonds Data

- ~54,000 round diamonds from http://www.diamondse.info/
- Additional information on carat, color, clarity, and cut
- For more information use the command

```
?diamonds
```

What can we learn from this data?

- Inspect the data
- Figure out what the variables are from http://www.diamondse.info/ and wikipedia
- Think of questions that you could answer with this data

Bar Charts

- Explore how one (or more) categorical variables are distributed.
- `qplot(cut, data = diamonds, geom = "bar")`

Histograms

- Explore how one (or more) quantitative variables are distributed.
- `qplot(price, data = diamonds, geom = "histogram", binwidth = 10)`
- `qplot(price, data = diamonds, geom = "histogram", binwidth = 500)`
- `qplot(price, data = diamonds, geom = "histogram", binwidth = 1000)`

Histograms

- Skewed to the right
  - There are lots of "inexpensive" diamonds and a few very, very expensive diamonds



Your turn

- Create histograms for carat and price per carat
- Experiment with bin width
- Describe what you see

Saving Plots

- There are a few ways to save your plots in R.
  - Using the GUI, choose file → save
  - `png(filename, width, height)`
  - `jpeg(filename, width, height)`
  - `pdf(file, width, height)`
- If the plot has already been rendered, then use
  - `dev.copy(device, filename)`
  - `ggsave(filename, width, height)`

Scatterplots

- Use scatterplots to display the relationship between two quantitative variables.



•Positive nonlinear association of moderate strength
•Variability in the price increases with carat
•There are "gaps" around 1.5 and 2 carats
•There may be outliers above 3 carats

Scatterplots

- ggplot2 will make a scatterplot by default when two variables are entered.
- `qplot(carat, price, data = diamonds)`
- `qplot(log(carat), log(price), data = diamonds)`
- `qplot(carat, price/carat, data = diamonds)`

Adding Smoothers

- `qplot(carat, price, data = diamonds, geom = c("point", "smooth"), method = lm, ylim = c(0, 20000))`
- `qplot(carat, price, data = diamonds, ylim = c(0, 20000)) + geom_smooth (method = lm)`

Adding Aesthetics

- We do not need to ignore the other variables when looking at scatterplots.
- We can map other variables to colour, shape, and size.
- `qplot(carat, price, data = diamonds, colour = color)`
- `qplot(carat, price, data = diamonds, shape = cut)`

Faceting

- Facets allow us to display plots for different subsets.
- facets = row variables ~ column variables (use '.' for none or blank for wrapping)
- `qplot(carat, price, data = diamonds, facets = . ~ color)`
- `qplot(carat, price, data = diamonds, facets = clarity ~ .)`

Boxplots

- Useful when we wish to investigate the relationship between a categorical and quantitative variable.
- Displaying side-by-side boxplots allow us to examine conditional distributions
- `qplot(clarity, price/carat, data = diamonds, geom = "boxplot")`

Your turn

- Add smoothers to the facetted scatterplots.
- Explore the relationships between price, carats, color, and clarity.
- What do you learn?

NASA Meteorological Data

- 24 x 24 grid across Central America
- satellite captured data:
  temperature (ts),
  near surface temperature (tsa)
  pressure (ps)
  ozone (o3)
  cloud coverage:
    low (ca_low)
    medium (ca_med)
    high (ca_high)
- for each location monthly averages for Jan 1995 to Dec 2000



Time Series Plots

for each location multiple measurements

```
qplot(TimeIndx, ts, geom="point",
data=subset(nasa, (Gridx==1) &
(Gridy==1)))
```



connected by a line

```
qplot(TimeIndx, ts, geom="line",
data=subset(nasa, (Gridx==1) &
(Gridy==1)))
```



4 - Intro to R

Models and Testing

Load data

- `library(ggplot2)`
- Automatically loads tips data
- `head(tips)`

Basic Tests

- `binom.test()`
- `chisq.test()`
- `t.test() #talk about this later`

Clean the data

- Data is not usually ready to work with.
- Examine data first.
- Rename variables?

Fitting Models

- Model grammar, Formula
- Linear Models
- Diagnostics

lm

- `lm(formula, data, weight, subset, na.action)`
- Formula
  - `y ~ x1 + x2 + x3*x4`

Simple linear regression

- Plot beforehand?
- Formula: `y ~ x`
- Let's use:
  - `tips.reg <- lm(tip ~ bill, data = tips)`

Output

- Coefficients, fitted values, residuals
- Anova table
- Can get predictions for future values

Diagnostics

- Residual sums of squares (deviance)
- AIC (Akaike Information Criterion)
- Residual plots, normal quantile plots

Your Turn

- Do a simple linear regression using psize as your predictor.
- Does it provide a good fit?
- Check the diagnostics
- Compare this fit to the previous model.

Multiple Regression

- Pretty much the same but we're adding more continuous variables.
- Formula:
  - `y ~ x1 + x2 (additive)`
  - `y ~ x1 + x2 + x1:x2 (with interaction)`

Your Turn

- Should we keep the interaction term in the model?
- Based on the model you choose what is the predicted value for a bill of 40 for a party size 3.

Anova

- Want to test means for multiple categories
- Formula is same: `y ~ x`
- Only difference is that x is categorical.
- R fits a treatment effects model by default

Constraints

- To make the model full rank R uses a "set first to 0" identifiability constraint. (`contr.treatment`)
- SAS uses "set last to 0" (`contr.SAS`)
- Another option: "sum to 0" (`contr.sum`)
- use `getOption("contrasts")` or `options()$contrasts` to check which one(s) you're using

Example

Friday	Saturday	Sunday	Thursday
μ	μ + Sat	μ + Sun	μ + Thur

Ancova and other models

- Can have categorical and continuous variables in a model.
- When adding terms
  - `a:b` is the interaction between a and b
  - `a*b` gives linear terms and interaction
  - `a*b` is the same as `a + b + a:b`

Your Turn

- Calculate basic summary stats for the variables.
- Create a few plots to show the relationships between the variables and tip.
- Are there any unusual points?
- Explore the data. Find a good model to predict tips.

Your Turn

- Create a new variable called tiprate
  - `tips$tiprate <- tips$tip/tips$bill`
  - Explore the relationship between the variables and tiprate.
  - Find a good model to explain tiprate.

Introduction to R

Programming in R

Control Structures

- Branching/Conditional statements:

the **if** statement

```
if(...) {
    ...
} else {
    ...
    }
```

example:

```
if(x < 0){
    cat("negative")
}
else{
    cat("non-negative")
}
```

Control Structures

Conditional Statements

logical comparisons

operator	meaning
==	equality
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

binary operators

operator	meaning
&&	AND both statements TRUE = TRUE
||	OR at least one statement TRUE = TRUE
!	NOT statement is FALSE = TRUE

Control Structures:

Conditional Statements

- Using an if statement in an assignment:

```
x <- if(...) 3.14 else 2.71
```

- The **ifelse** function

This function has the form ifelse(test, yes, no)

"test" is the logical condition, "yes" is what iftest() will return if the logical condition "test" is true, and "no" is what iftest() returns when false.

example:

```
x <- rnorm(100)
y <- ifelse(x>0, 1, -1)
z <- ifelse(x>0, 1, ifelse(x<0, -1, 0))
```

When x is positive, y & z store 1. When x is negative y stores -1 and z stores -1. When x = 0, y stores -1 and z stores 0.

- The **switch** statement

Control Structures:
Iteration and Looping

- The **for** loop:

do something *for* all items in a vector or list.

general syntax: here we iterate over the items in a vector (or list)

```
for (item in a_vector) {
   #operate on item
}
```

to iterate *n* times using a for loop, specify a vector of numbers 1:n in the for loop:

```
for (index in 1:100) {
    #do something 100 times
}
```


Control Structures:
Iteration and Looping

- The **while** loop:

do something *while* a logical statement is true.

general syntax: here we loop until *statement* is false

```
while (statement) {
   #do something
}
```

example: loop so long as x > 0

```
x <- 100
while (x > 0) {
     a <- runif(1, 1, 10)
     #do something
     x <- x - a
}
```

Control Structures:
Iteration and Looping

- The **repeat** statement:

  repeats the block of code until a break statement is reached

  general syntax: repeats until *statement* is true, when break is called

```
repeat {
    #do something
   if(statement) { break }
}
```

  example: repeat until x < 0

```
repeat {
     x <- rnorm(1, 0, 1)
      # do something
     if(x < 0){ break }
}
```

  note: the **break** command always terminates a loop

Control Structures:
Iteration and Looping

- lapply, sapply, and apply

  These functions can be used when the same or similar tasks need to be performed multiple times for all elements of a vector, list, or columns of an array. Easier and faster than a for loop.

- lapply

  Usage:

  lapply(alist, afunction) - applies afunction to all the elements of a list or vector, returns a list of the results. Example:

```
cap.state.name <-  lapply(state.name, toupper)
```

  cap.state.name is a list of all the state names capitalized.

Control Structures:
Iteration and Looping

- sapply

  Similar to lapply, but returns a simpler data structure: either a vector or array.

- apply

  Usage:

  apply(array/matrix, dim, afunction)

  Applies afunction to the dim dimension of matrix or array. Returns a vector. Example:

```
> x
     [,1] [,2] [,3]
[1,]    5    9    7
[2,]    7    8    6
[3,]    0    4    3
[4,]    7    6    5
> apply(x, 1, sum)
[1] 21 21  7 18
> apply(x, 2, sum)
[1] 19 27 21
```


Control Structures:
Iteration and Looping

Your Turn:

Write a program that iterates over a vector of states names and stores the states that begin with the letter M

R has a special vector for all the spaces names called state.name

To get the first character of a string, use the substr(string, start, stop) function

Example:

```
msg <- "hello"
first_char_msg <- substr(msg, 0, 1)
```

first_char_msg stores "h"

Functions

- What are functions?

Functions are blocks of code that allows R to be modular and facilitate code reuse. An R programmer can define their own functions as follows:

```
function_name <- function([arg1], [arg2], ...){
        #function code body
}
```

The function arguments are optional. Function arguments are the variables passed to the function, used by the function's code to perform calculations. A function can take no arguments.

A function can also return any R primitive or object using the return(object) statement.

Functions

- Examples:

```
# computes the mean of a vector of numbers

mean <- function(a_vector) {
     s <- sum(a_vector)
     x <- s/length(a_vector)
}


# checks to see if a string s starts with letter x
startsWith <- function(x, s){
      if(x == substr(s, 0, 1){
         return(TRUE)
       }
      return(FALSE)
}
```

Functions

- Default arguments

Function arguments can be assigned default values as follows:

```
sort_vector <- function(a_vector, ascending=TRUE){

     # sorting algorithm

}
```

when calling this function, the arguments given default values do not need to be specified. In this case, the defaults are used.

Example:

```
sort_vector(a_vector) # returns a_vector in ascending order

sort_vector(a_vector, FALSE) # returns a vector in descending order
```


Functions

- Variable Scoping

Variables that are bound to an R primitive or object outside a function are called global variables, and are accessible everywhere in an R program.

Example:

```
x <- 5
test <- function() {
   cat(x + 5)
}

test();  #prints out 25
```

Functions
variable scoping

- Variables bound inside a function are only accessible within that function. These are called local variables.

Example:

```
x <- 10
test <- function() {

  x <- 5
  cat(x + 20)
}

test() #prints 25
cat(x + 20) #prints 30
```

Note that the local variable assignment takes precedence inside the function test over the global assignment.

Functions
variable scoping

- R functions have no side effects-- they cannot change the value of global variables

Example:

```
x <- 10
test <- function(z) {
   z  <- z + 10
   cat(z)
}

test(x) #prints 20
cat(x) #prints 10
```

Note that x is still bound to 10 outside of test().

Advanced Function Topics

- Functions can be declared and used inside a function

Example:

```
test <- function(y) {
    square <- function(x) { return(x*x) }
    cube <- function(x)  {return(x^3) }
    return(square(x) + cube(x))
}

cat(test(4)) #prints 80
```


Advanced Function Topics

- Functions in R can also return function.

Example:

```
test <- function(y) {
    return(function(z) {y + z})
}

a <- test(10)
cat(a(4)) # prints 14
b <- test(2)
cat(b(4)) #prints 6
```

Note that the y in the returned function gets bound to the argument passed and never is never changed. Thus the variables in this function is *closed* inside its local environment. These types of constructs are referred to as closures.

Functions

Your Turn:

Write a function that computes the standard deviation of a vector of numbers.

Then write a function that determines whether a string contains the letter "s".

Intro to Simulation

- R is an excellent environment to do simulations. The programming techniques previously learned can be used to do simulations.

- In this section we cover simulation by example: dealing cards.

Intro to Simulation
dealing cards

- First we need some data to work on, in this case a deck of cards.

```
suit <- c("H" ,"C" ,"D", "S")
rank <- c(2:9, "T", "J", "Q", "K", "A")
deck <- NULL #create the deck
for(r in rank){
  deck <- c(deck, paste(r, suit))
}
```


Intro to Simulation
dealing cards

- Next we need to write functions to work on the deck of cards:

```
#returns a randomly shuffled deck
shuffle <- function(deck) {
    return(sample(deck,length(deck)))
}

#draws n cards from deck starting at card start
draw_cards <- function(deck, start, n=1) {

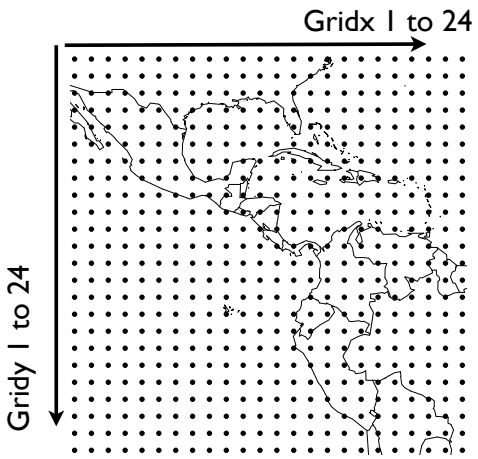
   cards <- NULL
   for(i in start:(start + n - 1)){
      if(i <= length(deck)){
           cards <- c(cards, deck[i])
      }
    }

    return(cards)
}
```

- We can simulate drawing cards, e.g. card games like poker.
- We can ask questions like "What is the probability of drawing 2 cards of the same suit from a deck of 52 cards?"

By simulating the drawing of 2 cards from a large number of randomly shuffled decks. These simulations are known as Monte Carlo simulations.

Example:

```
#checks to see it 2 cards are of the same suit
same_suit <- function(card1, card2) {
        return(substr(card1, 3,3) == substr(card2, 3,3))
}
#simulation
sscount <- 0
iterations <- 0
while(iterations < 10000) #do 10000 simulations
{
 sdeck <- shuffle(create_deck())
 twocards <- draw_cards(sdeck, n=2)
 if(same_suit(twocards[1], twocards[2])){
      sscount <- sscount + 1
 }
 iterations <- iterations + 1
}

p <- sscount/iterations #the estimated probability of 2 cards same suit
cat(p)
```


Simulation

Your Turn:

Use Monte Carlo simulation to determine the probability of drawing 2 cards of the same rank.

Introduction to R

Advanced Data Manipulation

plyr

- plyr allows the user to split a data set apart into smaller subsets, apply methods to the subsets, and combine the results.
- offers a replacement for loops
- To load the package you will need to enter the command:

```
library(plyr)
```

- If you do not have plyr installed, you can install the package in the R session by entering the command:

```
install.packages("plyr")
```

- A reference manual can be found on Hadley Wickham's website http://had.co.nz/plyr/

Baseball Data

- 21,699 records covering 1,288 players, spanning 1871 to 2007
- http://www.baseball-databank.org/
- For more information use the command

```
?baseball
```


Obtaining Summaries for Groups

- Finding summaries for one group is easy, but what if we want to find the same summaries for every group?
-
```
#Split
pieces <- split(baseball[,6:9], baseball$year)
```
-
```
# Apply
results <- vector("list", length(pieces))
names <- names(pieces)
for(i in seq(1, length(pieces))){
    piece <- pieces[[i]]
    results[[i]] <- mean(piece)
}
```
-
```
# Combine
    result <- do.call("rbind", results)
```

Obtaining Summaries for Groups

- An equivalent way to find the previous summary is:
- `ddply(baseball, .(year), function (df) mean(df[,6:9]))`

Understanding xxply

- There are 3 key inputs to xxply:
  - .data - data frame to be processed
  - .variables - splitting variables
  - .fun - function called on each piece
- The first letter of the function name gives the input type and the second gives the output type.

Understanding xxply

	array	data frame	list	discarded
array	`aaply`	`adply`	`alply`	`a_ply`
data frame	`daply`	`ddply`	`dlply`	`d_ply`
list	`laply`	`ldply`	`llply`	`l_ply`

Transform and Summarise

- Transform allows us to modify an existing data frame.
- `transform(df, var1 = expr1, ...)`
- Summarise creates a new data frame summarizing the data frame of interest.
- `summarise(df, var1 = expr1, ...)`

Transform and Summarise

- Calculating career year for Babe Ruth
- `baberuth <- subset(baseball, id == "ruthba01")`
- `baberuth <- transform(baberuth, cyear = year - min(year) + 1)`
- Calculating career year for all players
- `baseball <- ddply(baseball, .(id), transform, cyear = year - min(year) + 1)`

Transform and Summarise

- Total number of teams in the data frame
- `summarise(baseball, nteams = length (unique(team)))`
- Number of teams each player played for
- `ddply(baseball, "id", summarise, nteams = length(unique(team)))`

Your turn

- Create a summary including career length, rookie year and retirement year, and total number of games played (Hint: use summarise).
- Create a new variable for career batting average in the data frame.

Modeling

- Fitting a regression model relating seasonal batting average to career year
- ```
  model <- function(df){
      lm(h/ab ~ cyear, data = df
  }
  baseball <- subset(ab >= 25)
  bmodels <- dlply(baseball, .(id), model)
  bcoefs <- ldply(bmodels, function(x) coef(x))
  names(bcoefs)[2:3] <- c("intercept", "slope")
  ```
  ```

NASA Meteorological Data

- 24 x 24 grid across Central America
- satellite captured data:
  temperature (ts),
  near surface temperature (tsa)
  pressure (ps)
  ozone (o3)
  cloud coverage:
    low (ca_low)
    medium (ca_med)
    high (ca_high)
- for each location monthly averages for Jan 1995 to Dec 2000



Your turn

- Obtain summaries for each location (grid point)
  - overall
  - by year