Introduction to R

UVA StatLab Workshop Fall 2014

September 16, 2014

About R

R is

- a program (like SPSS, Stata, SAS, etc.) for statistical computing and graphics
- an interpreted programming language
  - work with data interactively, line by line
  - replicate previous work easily
- freely available (open-source) and maintained by volunteers
  - users can access and modify source code

Assumptions, Goals, and Expectations

Assumptions

- No experience with R
- Familiarity with basic statistical concepts

Goals

- Get you comfortable enough to start using R

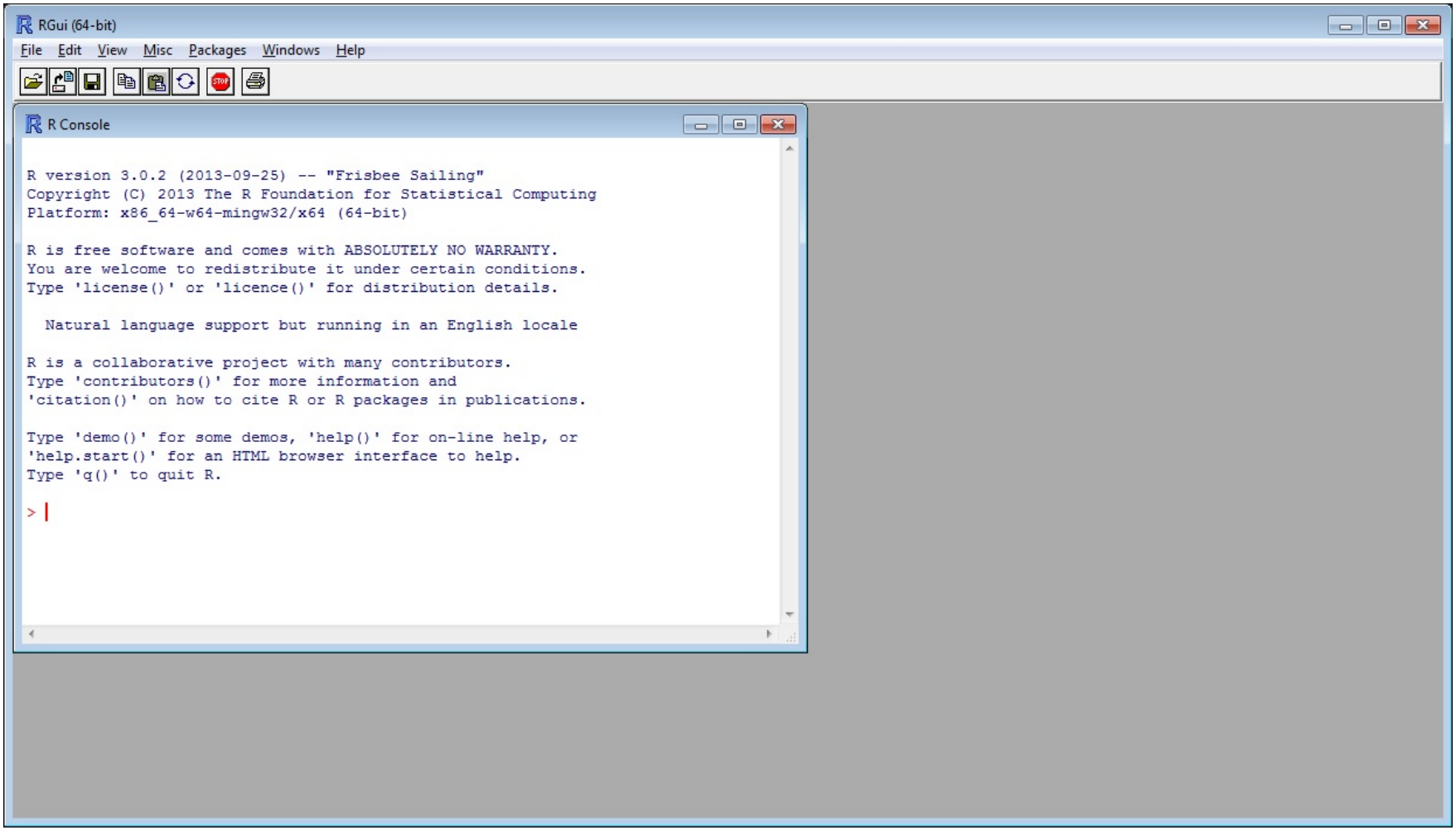
Expectations

- You will not learn R in a 90 minute workshop
- You must use R to learn R

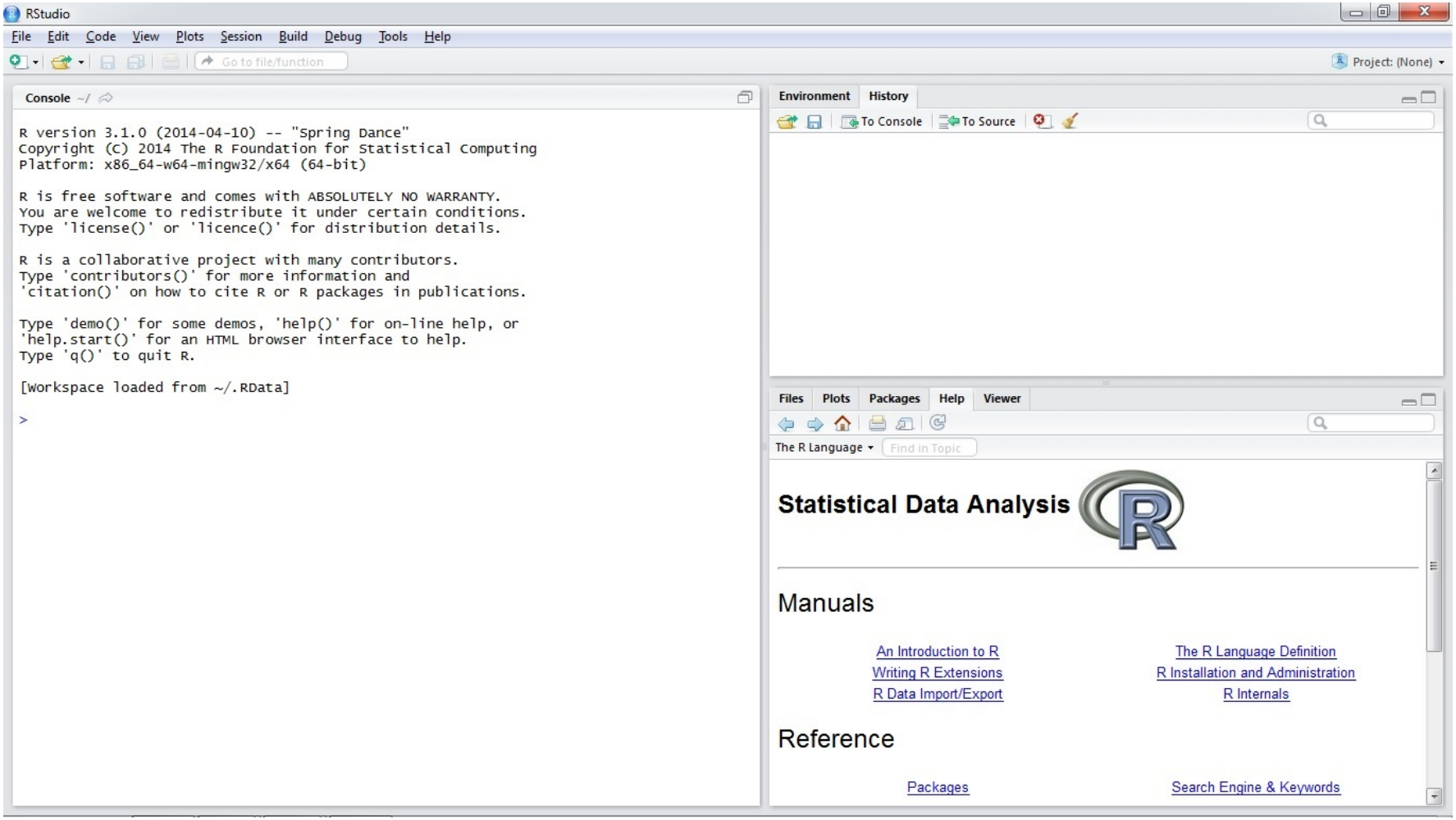
Section 1

The R Environment

R Startup

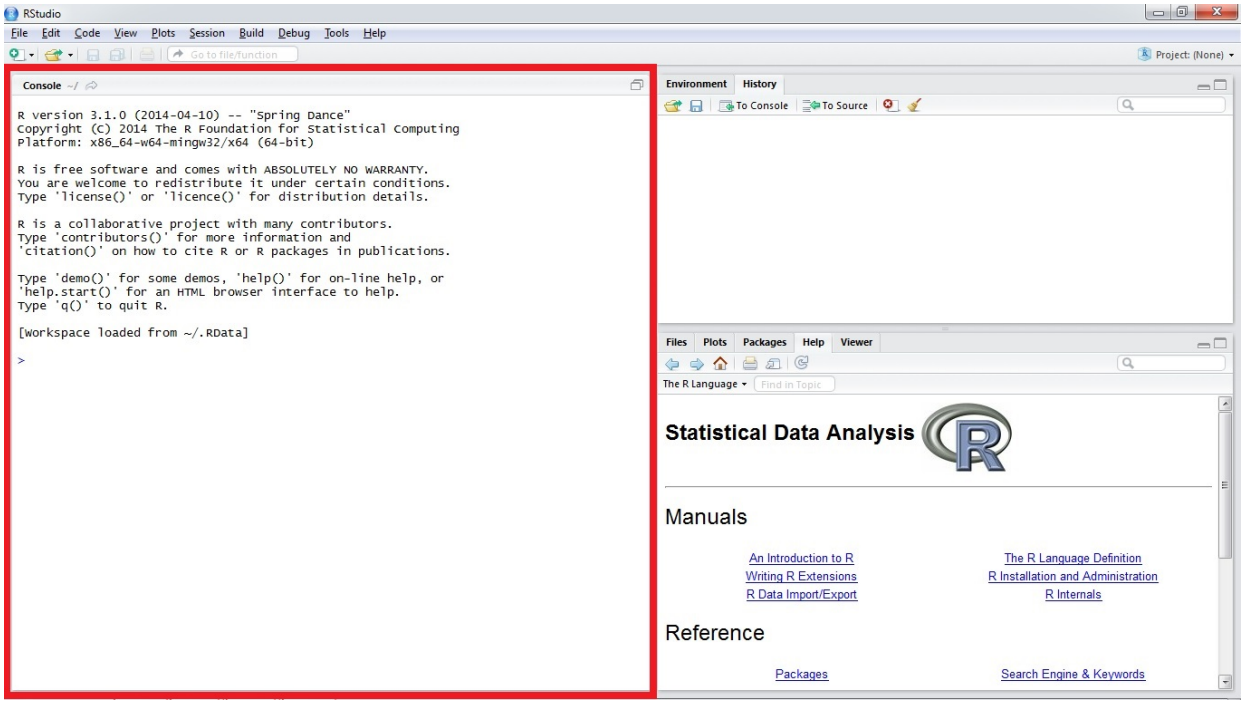

RGui (64-bit)
File Edit View Misc Packages Windows Help
R Console
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>

RStudio Startup



Console

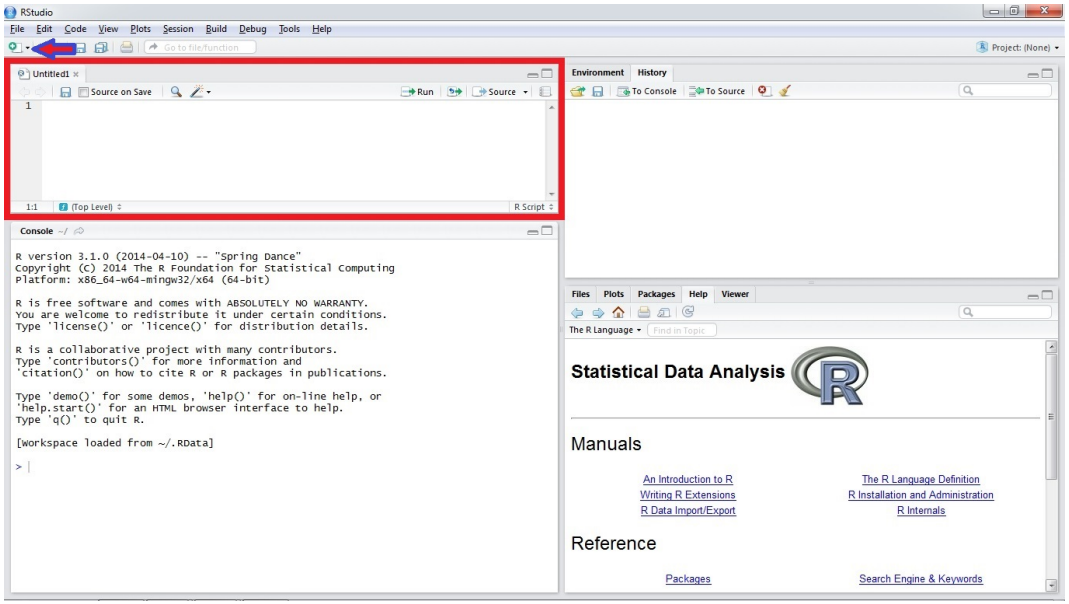
- Run code here
- Type commands after the prompt, ">"
- Hit **Return/Enter** to run



R Script

Although you can type commands directly into the console, you will more likely want to put the commands in a script, so that you can reuse them.

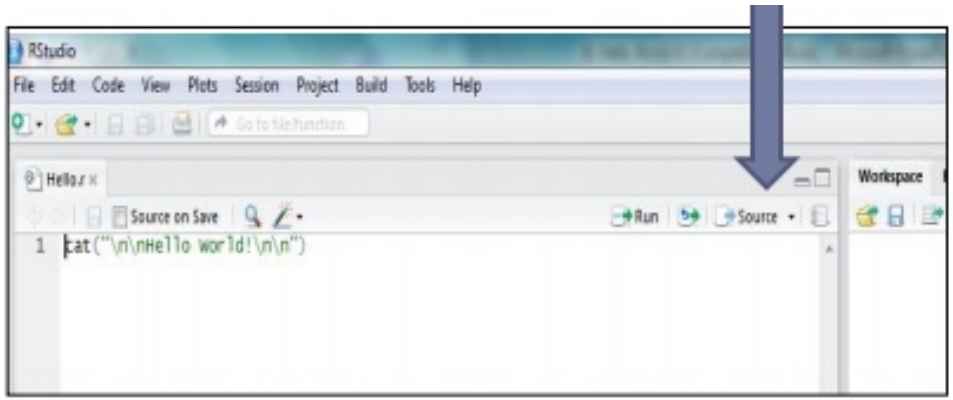
To create a new script, click on button or `File | New | R Script` or `Ctrl + Shift + N`.



Running lines of code from a script

There are several different ways one can run a line from the script in the Console pane

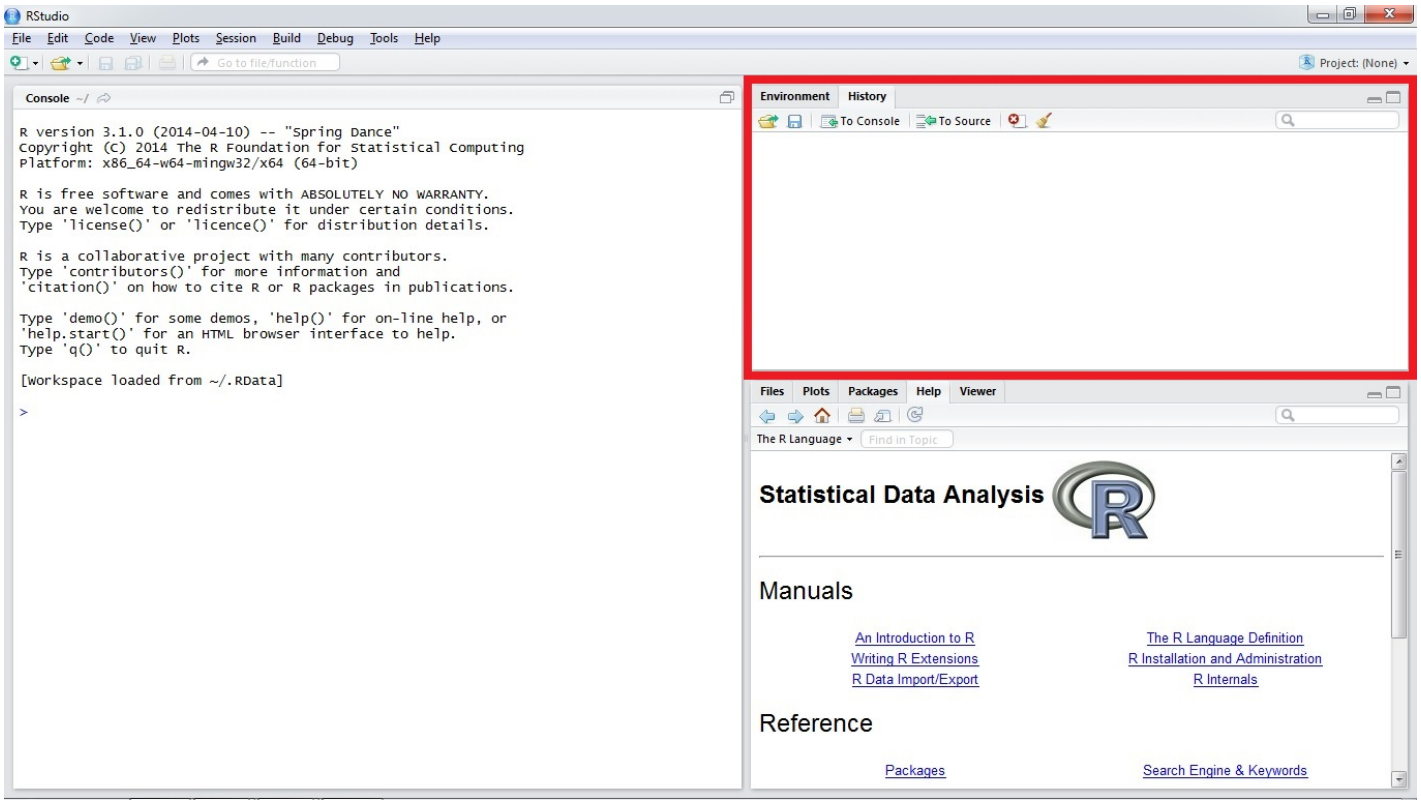
- Individual lines can be run
  - by copying and pasting them from one window to the next
  - running the line of script where the cursor is by typing `Ctrl R`
  - running the line of script where the cursor is by clicking the Run button at the top of the script pane
- Running the whole script
  - type in the console pane `source("scriptname")`
  - clicking the Source button in the upper right corner of the script pane



Workspace Pane

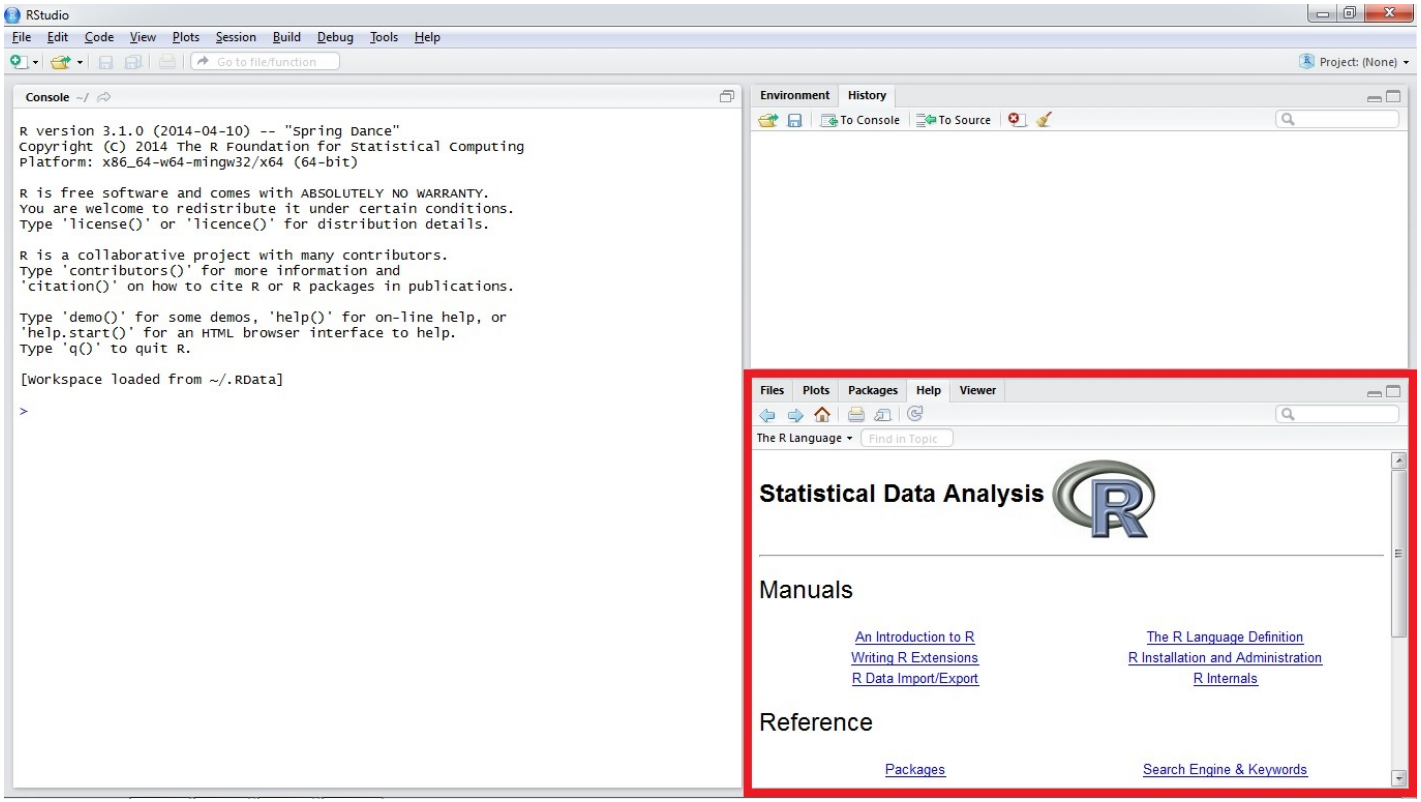
History Tab Keeps record of latest commands ran

Environment Tab List of objects active in memory



Multipurpose Pane

- Files in current workspace
- View plots
- List of installed packages
- R Help



Section 2

Importing Data

Reading in text files

`read.table()` is the easiest and most reliable method of entering data into R. It reads can recognize both numeric and character values but all values will be read in as numeric.

```
> read.table("file", header = FALSE, sep = " ")
```

Note that `read.table()` fails if there are any spaces in any of the variable names in the first row. You need to manually go into the file and remove all blank spaces in names or replace the blanks with a period.

Reading excel datasets

R won't import default excel files without loading a package because .xlsx files could contain graphs or multiple tabs containing tables.

We can read in datasets from excel using two different methods.

1. by using the `xlsx` package and `read.xlsx("file")`
2. save the excel file (.xlsx) as a .csv and then use `read.csv("file",header=TRUE,sep=",")`

```
> install.packages(xlsx)
> library(xlsx)
> data.xlsx <- read.xlsx("file.xlsx", sheetIndex = 1)
```

Setting Working Directory I

Say you have some data, `data.txt`, which is stored in `"C:/Users/MyName/Documents/R Stuff/"`

To read the data, I'll have to type:

```
> mydata <- read.table("C:/Users/MyName/Documents/R Stuff/data.txt",
+     header = TRUE)
```

Often it is easier to set a working directory so importing and saving files is easier in the workspace.

```
> mydata <- read.table("data.txt", header = TRUE)
```

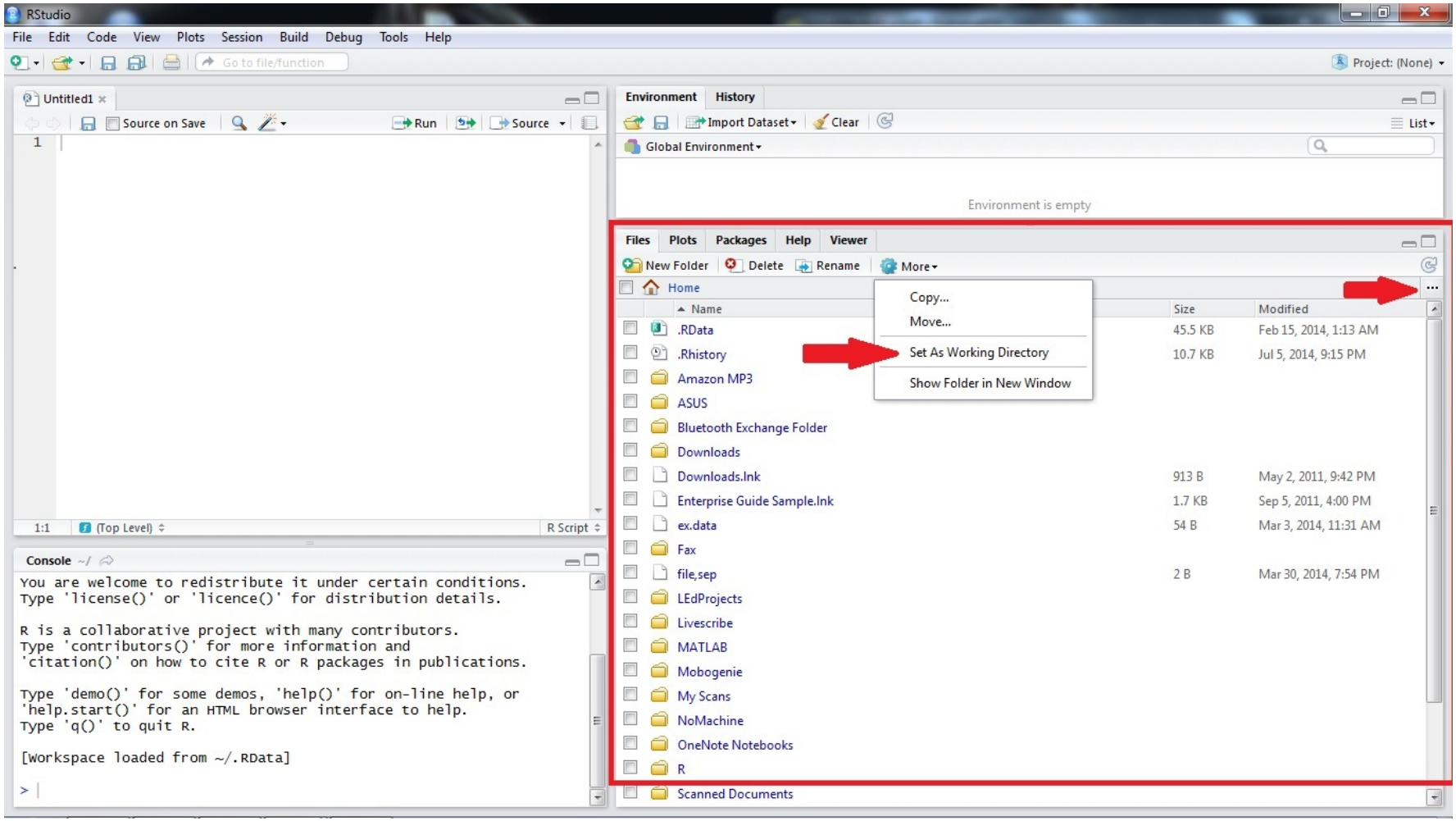
Setting Working Directory II

To set the working directory:

- `setwd("path")`, where the path is the folder you want as your directory.
  - Must use forward slash: `/` or double backslash: `\\`
  - Check if set correctly: `getwd()`
- Select the appropriate folder/directory by
  - `Ctrl + Shift + k`
  - `Session | Choose Directory ...` (Windows)
    or
    `Tools | Change working directory` (Mac)

Setting Working Directory III

- `File` tab in the Multipurpose pane



Packages I

- Think of R/RStudio as an Operating System (e.g. Windows, Mac)
- Packages are the "applications" (e.g. MS Word, Chrome)
- We need to click on the "applications" to use them, aka install and load in R as needed
  - Initial R download includes; `base`, `graphics`, `stats`, `utils`

Packages II

- In order to install packages you must have an internet connection and you need to know what the name of the package you want to install.
  - Packages only need to be installed once on a computer, until you update your R version

```
> # installs package
> install.packages("MASS")
```

- You also need to load the package everytime you wish to use the package.
  - No need to re-load package within the same R session
  - Need to load the package again after terminating a R session

```
> # loads package
> library("MASS")
> # or
> require("MASS")
```

Section 3

Writting Commands

Basics I

- If incomplete command in console, will prompt "+" instead of ">"
  - either complete command or abort by hitting ESC
- R is case sensitive, e.g. x not same as X
- Commands are separated by a ; or by a newline
- R ignores spaces. So can use white space to organize code. Thus can use spacing around all infix operators (`=`, `+`, `-`, `<-`, `etc.`) and after commas

Basics II

- Comments
  - start with "#" sign
  - Keyboard shortcut is Ctrl+Shift+C

```
> # This is a comment. When a # is placed somewhere on a
> # line, everything after the sign is considered a
> # comment (including other #s)
```

R as a calculator

R works like a calculator, you type an expression and you get the answer. The standard arithmetic operators are `+`, `-`, `*`, `/`, `^`.

```
> 15 + 10

[1] 25

> 12/6

[1] 2

> 0.7 * 5

[1] 3.5
```

Just like any calculator, these operators have the standard precedence (e.g. exponentiation highest and addition/subtraction lowest), but you can always control the order of evaluation with parentheses.

Variables/Objects I

- A variable is used to store information, whether data, a computation result, a word, or an image.
- Technically, a variable represents a memory location, but you can think of it like a label for some object.
- Do not need to be declared before (like some other computer programs), but are created as soon as an object is assigned to it.

Variables/Objects II

- To assign a name to a variable one must use the assignment operator <- or =.

```
> value1 <- 22  #stores 22 in value1
> value2 <- 22/7  #stores the result of 22/7 in value2
```

- To recall the value stored in a variable, type out the variable name in the console to have the value stored print out.

```
> value1

[1] 22

> value2

[1] 3.143
```

Variables/Objects III

- Can be made out of letters, numbers, and the dot (`.`) or underline (`_`) characters (no blank spaces).
  - `3value <- 100` will not work (can't start with number)
  - `.value <- 100`
  - `.3value <- 100` will not work (can't have a number after a .)
  - `value <- 100`
  - `value 3 <- 100` will not work (can't have spaces)
- Just like everything else in R, variables are case sensitive

```
> value3 <- 100
> value3

[1] 100

> Value3

Error:  object 'Value3' not found
```

Variables/Objects IV

- stored objects can be used in subsequent operations

```
> value1

[1] 22

> value1/2

[1] 11
```

- results can be stored in a new object

```
> new.value <- value1/2
> new.value

[1] 11
```

Variables/Objects V

- objects of the same name will be overwritten

```
> value4 <- 5^2
> value4

[1] 25

> value4 <- sqrt(100)
> value4

[1] 10
```

R Data Structures

R's base data structures can be organized by their dimensionality (0d, 1d, or 2d) and whether they're homogeneous (all contents must be the same type) or heterogeneous (contents can be of different types).

0d	Scalar	
1d	Vector	List
2d	Matrix	Dataframe

Section 4

Vectors

Vectors, $x_1, x_2, \ldots, x_n$

A vector is a collection of values (1-dimensional array), all of the same type, ordered in a certain way.

Five common types (or class) of vectors

1. logical (TRUE, FALSE)
2. integer $(1, 2, 3, \ldots)$
3. double/numeric $(1.0, 1.5, \ldots)$
4. character/string ("a", "b", ...)
5. factor (a categorical variable with predefined levels)

By default, R treats all variables as vectors, unless otherwise specified, even if the variable holds only one value.

Combine/Concatenate I

- Vectors are usually made with `c()`, short for combine or concatenate, in which the arguments are seperated by a ",".
- The order in which the entries are inputed is very important, as it is how R stores the entries inside the vector.

```
> dbl_var <- c(1, 2.5, 4.5)
> dbl_var

[1] 1.0 2.5 4.5
```

Combine/Concatenate II

```
> # With the L suffix, you get an integer rather than a
> # double
> int_var <- c(1L, 6L, 10L)
> int_var

[1]  1  6 10

> # Use TRUE and FALSE (or T and F) to create logical
> # vectors
> log_var <- c(TRUE, FALSE, T, F)
> log_var

[1]  TRUE FALSE  TRUE FALSE

> # character or strings are enclosed in ' '
> chr_var <- c("Rob", "Tess", "Megan", "Kevin", "Liam")
> chr_var

[1] "Rob"   "Tess"  "Megan" "Kevin" "Liam"
```

Class of Object

We can find the type/class of any R object using `class()`.

```
> class(dbl_var)

[1] "numeric"

> class(log_var)

[1] "logical"
```

Numeric Sequences

To make a sequence of numbers, one can use one of two methods (both allow both positive and negative numbers)

1. `a:b` creates a sequence of numbers starting at a and increasing/decreasing by 1 integer value until it gets to b

```
> 1:3

[1] 1 2 3

> 3:1

[1] 3 2 1
```

2. `seq(from, to, by=1)` creates a sequence of numbers starting at the from value and increasing/decreasing by 1 interger value until it gets to the to value

```
> seq(1, 3, by = 0.5)

[1] 1.0 1.5 2.0 2.5 3.0
```

Repeating I

One can also create a vector using the `rep()` function, short for repeat/replicate.

- Given a vector one can also repeat the vector `n` times `rep(vector, times)`

```
> rep(1:3, 2)

[1] 1 2 3 1 2 3

> rep(c(1, 2, 3), times = c(1, 2, 3))

[1] 1 2 2 3 3 3
```

Repeating II

- Or one can also repeat each element in the vector `n` times in a row `rep(vector, each)` or `rep(vector, times)`

```
> rep(1:3, c(2, 2, 2))

[1] 1 1 2 2 3 3

> rep(c(1, 2, 3), each = 3)

[1] 1 1 1 2 2 2 3 3 3
```

Combining vectors

One can also combine vectors into a new vector by using the concantenate function, `c()`. This function takes not only individual elements but can take vectors or even a combination of both to make a new vector.

```
> girls <- c("Juliet", "Sierra")
> boys <- c("Romeo", "Oscar")
>
> everyone <- c(girls, boys)
> everyone

[1] "Juliet" "Sierra" "Romeo"  "Oscar"

> everyone <- c(boys, girls)
> everyone

[1] "Romeo"  "Oscar"  "Juliet" "Sierra"
```

Subsetting Vectors

- The typical notation used is square brackets specifying what index value should be extracted or removed.
- One can also remove or extract several elements from a vector by using a vector of indexes as the argument insides the brackets.
- Again the order of the argument values do affect the way that the information is printed/stored.
- Neither of these actions permanently change the data, unless you assign it as such.

Subset a Vector I

1. Extract values - using positive integers return elements at the specified positions

```
> x <- c(2.1, 4.2, 3.3, 5.4)
> x[c(3, 1)]

[1] 3.3 2.1

> # Duplicated indices yield duplicated values
> x[c(1, 1)]

[1] 2.1 2.1

> # Real numbers are silently truncated to integers
> x[c(2.1, 2.9)]

[1] 4.2 4.2
```

Subset a Vector II

2. Omit Values - negative integers omit elements at the specified positions

```
> x

[1] 2.1 4.2 3.3 5.4

> x[-c(3, 1)]

[1] 4.2 5.4

> x[-3]

[1] 2.1 4.2 5.4
```

Replacing a value

Sometimes, data is inputed wrong and a value of a vector or other data structure needs to be replaced with the correct value.

```
> x <- c(2, 4, 6, 8)
> x[1] <- 1  #replaces the first index value with the value 1
> x

[1] 1 4 6 8

> x[c(2, 3)] <- c(2, 3)
> x

[1] 1 2 3 8
```

Built in Vector Functions I

R operations are vectorized, so can perform actions without having to use a loop. If x is a vector, then `log(x)` is a vector with the logs of the elements of x.

- Arithmetic and relational operators also work element by element. If x and y are vectors of the same length, then x+y is a vector with elements equal to the sum of the corresponding elements of x and y.
- If x and y are vectors of different lengths, the shorter one is recycled as needed.

Built in Vector Functions II

```
> x <- c(2, 4, 6, 8)
> y <- c(7, 8, 3, 1)
> 2 * x  #multiplication

[1]  4  8 12 16

> y + 3  #addition

[1] 10 11  6  4

> x + y  #adds 1st entries, adds 2nd entries, etc.

[1]  9 12  9  9

> x * y

[1] 14 32 18  8
```

Built in Vector Functions III

```
> x

[1] 2 4 6 8

> y

[1] 7 8 3 1

> sum(x)  #summation

[1] 20

> mean(y)  #average

[1] 4.75
```

Logical operations I

Logical expressions use logical operators to evaluate whether something is TRUE or FALSE

- `>, >=, <, <=`

```
> x <- rep(1:3, 2)
> x

[1] 1 2 3 1 2 3

> x >= 3

[1] FALSE FALSE  TRUE FALSE FALSE  TRUE

> x < 2

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

Logical operations II

- Equal: ==; Not equal: !=

```
> x == 1
[1]  TRUE FALSE FALSE  TRUE FALSE FALSE
> x != 1
[1] FALSE  TRUE  TRUE FALSE  TRUE  TRUE
```

- And: &; Or: |

```
> x
[1] 1 2 3 1 2 3
> x > 1 & x < 3
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE
> x == 1 | x == 2
[1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE
```

Section 5

Dataframes

Dataframes I

- Dataframes are used for storing data tables (similar to a matrix).
  - Most built-in data sets and any read in data are dataframe objects
  - The rows contain different observations from the study
  - The columns contain the values of different variables of which contain different data types and usually labeled

Dataframes II

- Create dataframes with `data.frame(vectors)` (and specify column names)

```
> my.score <- c(1, 3, 2, 2, 4)
> my.names <- c("Rob", "Tess", "Megan", "Kevin", "Liam")
> my.gender <- c("M", "F", "F", "M", "M")
> my.dat <- data.frame(Score = my.score, Names = my.names,
+     Gender = my.gender)
> my.dat

  Score Names Gender
1     1   Rob      M
2     3  Tess      F
3     2 Megan      F
4     2 Kevin      M
5     4  Liam      M
```

Dataframes III

```
> # install.packages('car')
> library(car)
>
> # Load the Vocab data from the car package
> data(Vocab)
>
> # check if dataframe
> class(Vocab)

[1] "data.frame"
```

Obtaining General Information I

- What are the dimensions (rows x columns)?

```
> dim(Vocab)
[1] 21638     4
> nrow(Vocab)
[1] 21638
> ncol(Vocab)
[1] 4
```

- What are the variables names in the dataset?

```
> names(Vocab)
[1] "year"       "sex"        "education"  "vocabulary"
```

Obtaining General Information II

- What are the first 6 observations of the dataset?

```
> # default head(data, n=6)
> head(Vocab)

         year    sex education vocabulary
20040001 2004 Female         9          3
20040002 2004 Female        14          6
20040003 2004   Male        14          9
20040005 2004 Female        17          8
20040008 2004   Male        14          1
20040010 2004   Male        14          7
```

Obtaining General Information III

- What are the last 6 observations of the dataset?

```
> # default tail(data, n=6)
> tail(Vocab)

         year    sex education vocabulary
19982818 1998   Male        14          4
19982819 1998 Female        12          4
19982821 1998   Male        12          5
19982822 1998   Male        12          5
19982825 1998   Male        13          8
19982828 1998 Female        14          6
```

Attaching Dataset

- We can make the columns of a dataframe into individual vectors accessible by their name within the R session using `attach()`
  Note - These vectors won't show up in the workspace window
- We can remove the vector variables created by `attach()` using `detach()`

```
> attach(Vocab)
> year[1:10]

 [1] 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004

> detach(Vocab)
> year

Error:  object 'year' not found
```

Subsetting Datasets I

One can also subset matrices and dataframes using the general bracket notation introduced with vectors. Two arguments are required within the square brakets, one to reference the row(s) and one to reference the column(s)

```
> # first row, first column
> Vocab[1, 1]

[1] 2004

> # first two entries in row 1 and row 2
> Vocab[1:2, 1:2]

         year    sex
20040001 2004 Female
20040002 2004 Female
```

Subsetting Datasets II

```
> # entire first row
> Vocab[1, ]
> # entire first column
> Vocab[, 1]
```

If the dataset (dataframe) contains headers for the columns, we can also grab a specific column by using the $ and/or specific column name/header.

```
> Vocab$year[1:5]

[1] 2004 2004 2004 2004 2004

> Vocab[1:5, "year"]

[1] 2004 2004 2004 2004 2004
```

Subsetting Datasets III

We can remove column(s) or row(s) from a dataframe by using a scalar or vector negative value(s) within the square brackets

```
> head(Vocab, n = 3)

         year    sex education vocabulary
20040001 2004 Female         9          3
20040002 2004 Female        14          6
20040003 2004   Male        14          9

> temp <- Vocab[, -1]
> head(temp, n = 3)

            sex education vocabulary
20040001 Female         9          3
20040002 Female        14          6
20040003   Male        14          9
```

Subsetting Datasets IV

We can add column(s) or row(s) from a dataframe by using the functions `cbind()` and `rbind()`, respectively.

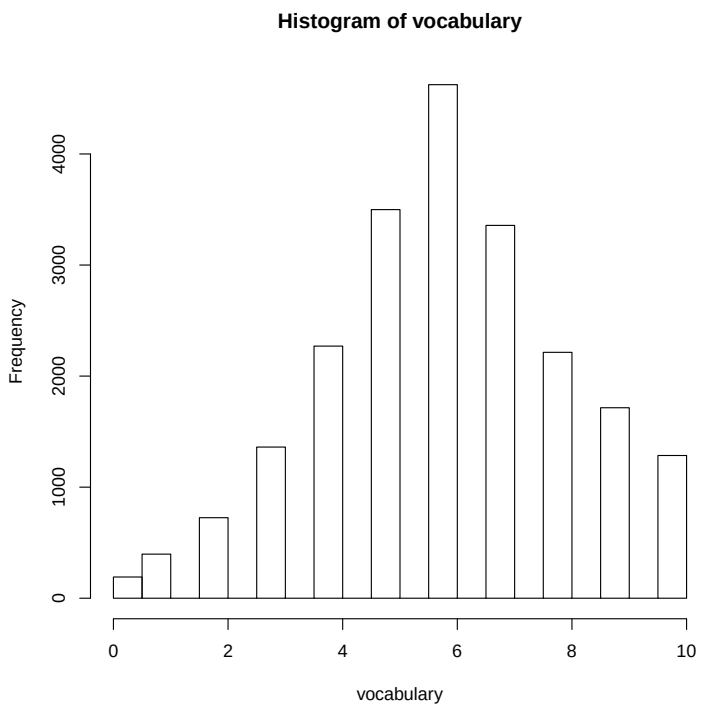
```
> attach(Vocab)
> temp <- cbind(Vocab, log(vocabulary))
> head(temp, n = 3)

         year    sex education vocabulary
20040001 2004 Female         9          3
20040002 2004 Female        14          6
20040003 2004   Male        14          9
         log(vocabulary)
20040001           1.099
20040002           1.792
20040003           2.197
```

Section 6

Graphics

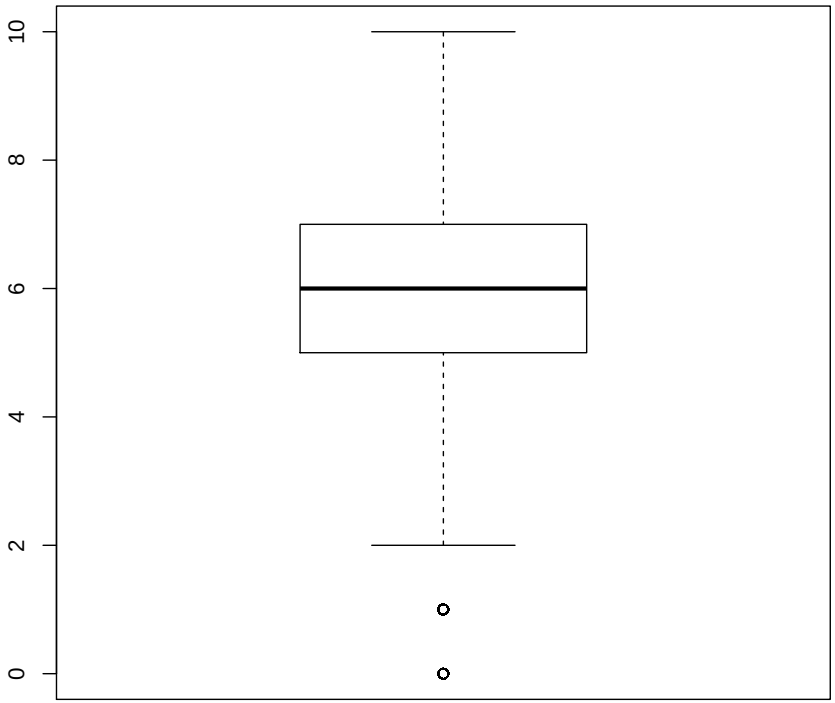
Histograms

```
> names(Vocab)

[1] "year"       "sex"        "education"  "vocabulary"

> hist(vocabulary)
```



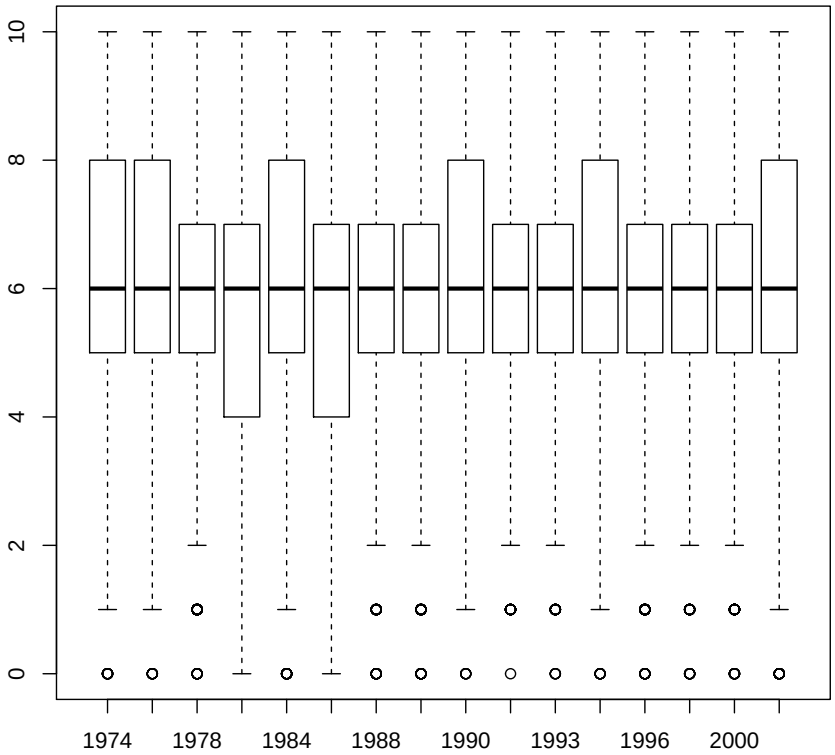
Boxplots I

```
> boxplot(vocabulary)
```



Boxplots II

```
> boxplot(vocabulary ~ year)
```



Scatterplots I

The `plot()` function draws axes and adds a scatterplot of points. Two extra functions exist `points()` and `lines()` which add extra points or lines to an existing plot.

Two ways to write the parameters for any of these functions

- Cartesian, `plot(x,y)`
- Formula, `plot(y~x)`

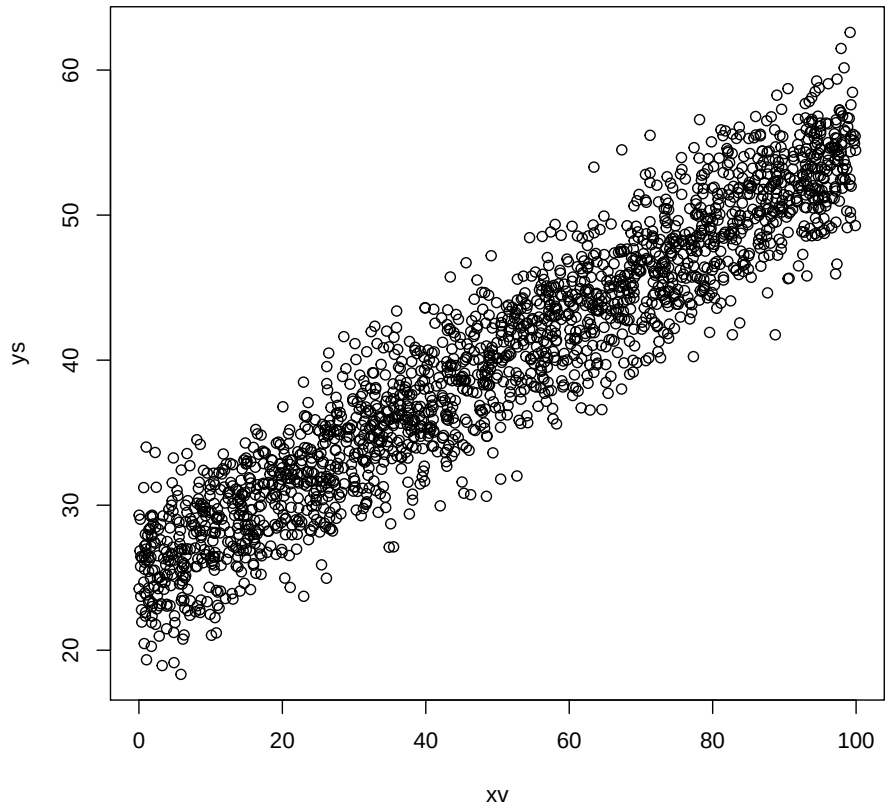
There are 256 different plotting symbols available, in which the default is `pch=1`

Scatterplots II

```
> data1 <- read.table("scatter1.txt", header = TRUE)
> attach(data1)
> names(data1)

[1] "xv" "ys"

> data2 <- read.table("scatter2.txt", header = TRUE)
> attach(data2)
> names(data2)

[1] "xv2" "ys2"
```

Scatterplots III

```
> # making a scatterplot
> plot(xv, ys)
```



Scatterplots IV

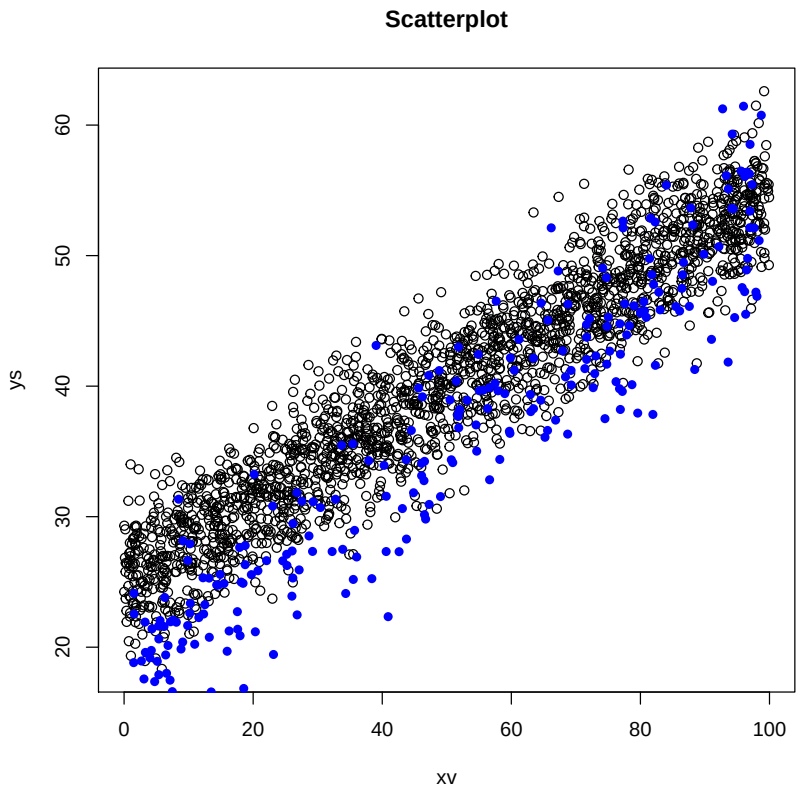
- The `plot()` function by default plots points, so to create lines you need to include the parameter `type="l"`, which will draw by default a solid black line.
- While most plots in R are given a default title, this title is not often representative of what the plot represents so often you want to relabel the title.
  1. `plot(x,y, main="Title")`
  2. after your graphic code use the `title()` function, `title(main="Title")`

Scatterplots V

- The characteristic of this line can be changed by defining the line type, `lty=`, and the line width, `lwd=`. Both of these take a positive number as their input value.
  - For `lwd=` the larger the number the thicker the line gets
  - For `lty=`
    - 1 – "solid line"
    - 2 – "dashed line"
    - 3 – "dotted line"
- Colors of points or lines can be changed by using the parameter `col=` inside of the `plot()`,`lines()` or `points()` functions.

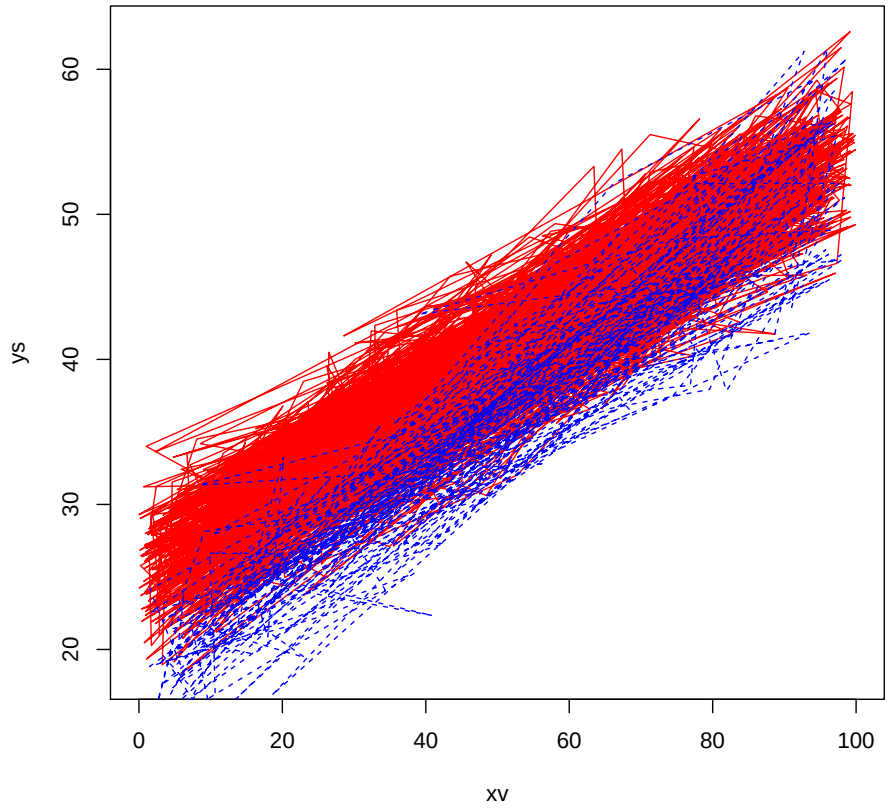
Scatterplots VI

```
> plot(xv, ys)
> points(xv2, ys2, col = "blue", pch = 16)
> title("Scatterplot")
```



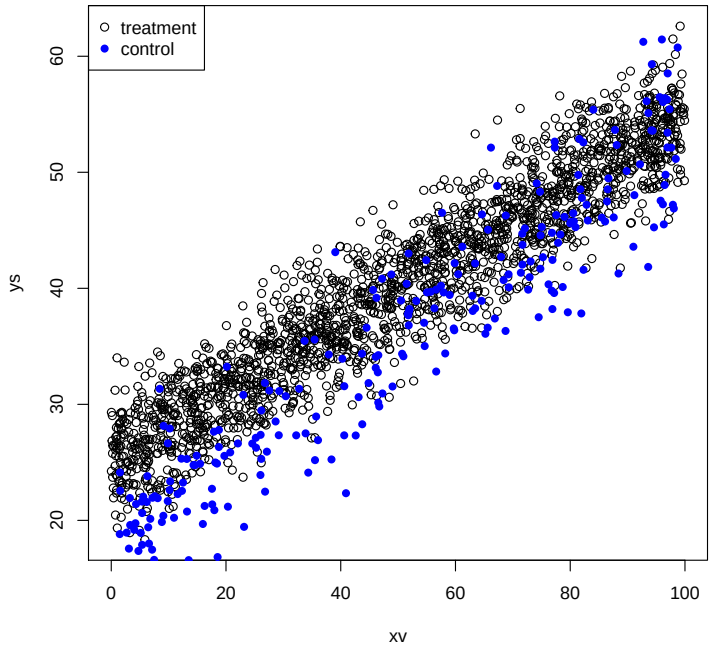
Scatterplots VII

```
> plot(xv, ys, type = "l", col = "red")
> lines(xv2, ys2, lty = 2, col = "blue")
```



Scatterplots VIII

```
> plot(xv, ys)
> points(xv2, ys2, col = "blue", pch = 16)
> # adding legend
> legend("topleft", c("treatment", "control"), pch = c(1,
+     16), col = c("black", "blue"))
```



Section 7

Hypothesis testing

T-test I

- One Sample, `t.test(x,mu= , alt="two.sided")`
  - Assume data is i.i.d. sequence from a $N(\mu, \sigma)$ distribution (both μ and σ are unknown) or that n is large enough for the CLT to apply

$$H_0 : \mu = \mu_0 \quad \textit{versus} \quad H_1 : \mu < \mu_0, \mu > \mu_0, \text{ or } \mu \neq \mu_0$$

- Two Sample, `t.test(x,y,alt="two.sided",var.equal=FALSE)`
  - Assume data $X_1, \ldots, X_{n_x}$ and $Y_1, \ldots, Y_{n_y}$ are independent random samples from a $N(\mu_x, \sigma_x)$ and $N(\mu_y, \sigma_y)$ distributions (both μ's and σ's are unknown)

$$H_0 : \mu_x = \mu_y \quad \textit{versus} \quad H_1 : \mu_x < \mu_y, \mu_x > \mu_y, \text{ or } \mu_x \neq \mu_y$$

T-test II

(Using the scatter1.txt and scatter2.txt)

```
> t.test(ys, mu = 40, alt = "greater")


One Sample t-test

data:  ys
t = 0.1426, df = 1999, p-value = 0.4433
alternative hypothesis: true mean is greater than 40
95 percent confidence interval:
 39.69   Inf
sample estimates:
mean of x
    40.03
```

T-test III

```
> t.test(ys, ys2)


	Welch Two Sample t-test

data:  ys and ys2
t = 5.678, df = 289.4, p-value = 3.311e-08
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 2.833 5.839
sample estimates:
mean of x mean of y
    40.03     35.69
```

Resources

Quick-R
`http://www.statmethods.net/`

Cookbook for R
`http://www.cookbook-r.com/`

R reference card
`http://cran.r-project.org/doc/contrib/Short-refcard.pdf`