

# Introduction to R and Studio

Andrew Zieffler

---



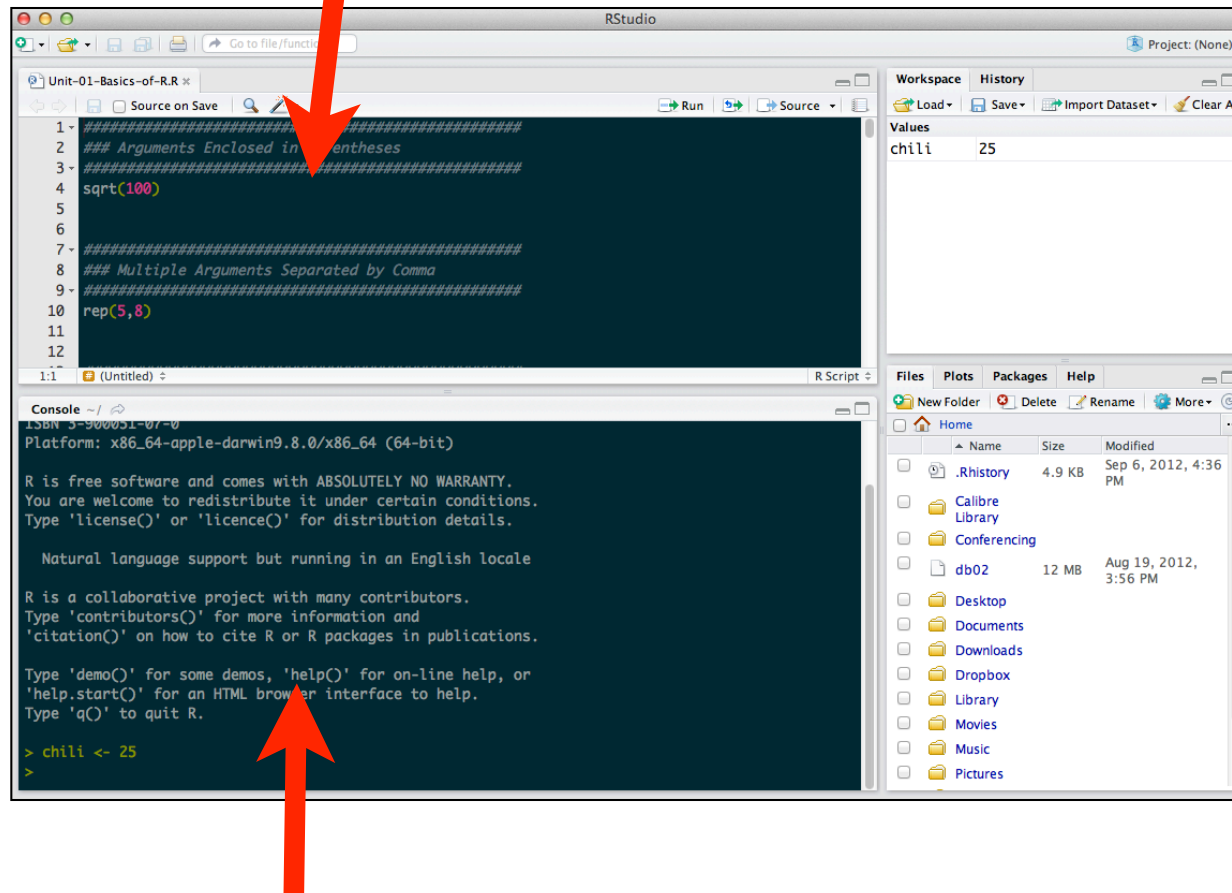
This work is licensed under a  
[Creative Commons Attribution  
4.0 International License](https://creativecommons.org/licenses/by/4.0/).

# Some Things to Remember...

- R is case-sensitive
  - `anova` is different than `Anova` is different than `ANOVA`
- R does exactly what the user codes
  - The user and R often don't see "eye-to-eye"
- R has a "long-term" memory...
  - ...but only in the current session

# RStudio

**Script pane:** non-interactive typing



**Console pane:** Interactive typing

# Console Pane (Interactive)

>

**R prompt...**R is waiting for a command

> 3 + 2

After inputting a command hit **<enter>**

[1] 5

**R returned value...**the brackets indicate the *i*th returned element. Here they indicate the first returned element. (They can be ignored for now (we are more interested in the returned value of 5.)

> 3 +

+

**Continuation prompt...** The previous command was not properly ended prior to hitting **<enter>**

- Finish command
- Hit **<esc>** until you get the command prompt again. Then re-try

} **Two options**

```
> 4 * 5  
[1] 20
```

```
> 10 ^ 3  
[1] 1000
```

```
> 1 / 0  
[1] Inf
```

Space is for humans.

```
> 4 * 5
```

```
> 4*5
```

```
> 4 *5
```

are all the same to R

# Computations in R using Functions

```
> sqrt(100)
[1] 10

> log(7)
[1] 1.94591

> sin(50)
[1] -0.2623749

> exp(3)
[1] 20.08554

> log(100, 10)
[1] 2

> log(100, base = 10)
[1] 2
```

## Three components

- Function
- Argument
- Returned value

### Arguments

Enclosed in parentheses

Multiple arguments  
separated by commas

Can be named or unnamed

```
> log(100, 10)
[1] 2
```

Order matters for unnamed arguments

```
> log(10, 100)
[1] 0.5
```

```
> log(x = 100, base = 10)
[1] 2
```

Order does not matter for named arguments

```
> log(base = 10, x = 100)
[1] 2
```

```
> log(100, base = 10)
[1] 2
```

Conventionally, the first argument is often unnamed and the remainder are named

# Connecting Computations in R

## Two methods of connecting computations

- Chaining
- Assignment

```
> sqrt(log(100, base = 10))
```

```
[1] 1.414214
```

```
> chili <- log(100, base = 10)
```

```
> chili = log(100, base = 10)
```

```
> sqrt(chili)
```

```
[1] 1.414214
```

Chaining uses a complete computation as the argument for another computation.

The assignment operator is either <- or =

Assignment stores the returned value from a computation as a named object.



# Objects

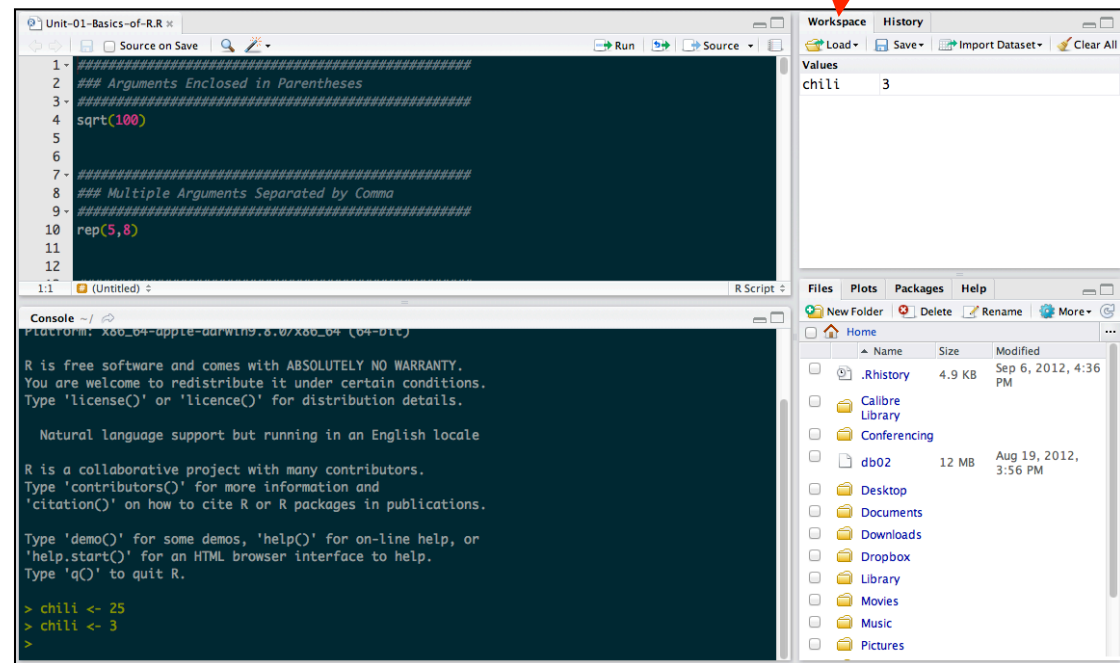
```
> chili = 3  
  
> sqrt(chili)  
[1] 1.732051  
  
> chili  
[1] 3
```

When an object name is re-used, the previous value of the object is lost.

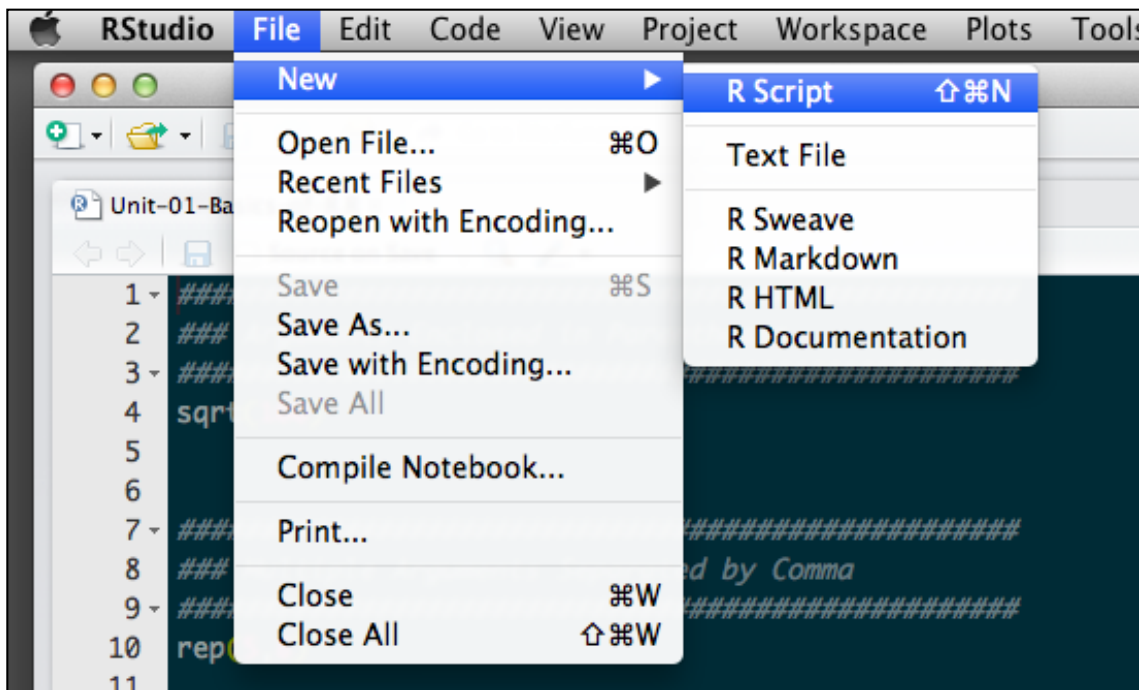
Workspace: List of current objects



- Pretty much any name can be used
- Descriptive is better
  - chili is not a good object name
- Names cannot include hyphens or spaces
- Names cannot begin with a digit
- Conventions in CS
  - myData (bumpy / camel-case)
  - my\_data (use underscore / period)



# Saving Your Work: Script Files



Record syntax in a **script file**

- Includes syntax (R commands) only
  - No prompts
  - No output
- Should also include comments
  - # indicates a comment

```
# Arithmetic computations
```

```
3 + 2
```

```
4 * 5
```

```
10 ^ 3
```

good script files have  
comments

```
# Assignment
```

```
chili = log(100, base = 10)
```

```
# Compute things
```

```
sqrt(chili)    # Find the square root
```

```
log(chili)     # Find the natural logarithm
```

Comments can also be  
placed on an existing line

Syntax in the script file can be  
executed by highlighting it and  
pressing the **Run** button



# Data Structures: Vectors

Vectors are collections of data (univariate; a single variable). They are typically assigned to an object. To create a vector use the `c()` function.

```
> age = c(40, 37, 9, 2, 10)
```

```
> age
```

```
[1] 40 37 9 2 10
```

```
> age + 5
```

```
[1] 45 42 14 7 15
```

Some computations are  
element-wise

```
> age ^ 2
```

```
[1] 1600 1369 81 4 100
```

```
> mean(age)
```

```
[1] 19.6
```

Some computations are  
vector-wise

```
> sum(age)
```

```
[1] 98
```

# Character Vectors

Not all vectors are numeric. R has six different types of vectors. Use `class()` to find out what type you have.

In a **character vector** (a.k.a. strings, literals) each element is a character string. Character strings are delimited by quotation marks.

```
> family = c("Andy", "Lauren", "Chili", "Sadie", "Einstein")

> family
[1] "Andy"      "Lauren"    "Chili"     "Sadie"     "Einstein"

> family + 5
Error in family + 5 : non-numeric argument to binary operator

> length(family)
[1] 5

> family2 = c("Andy", "Lauren", "Chili", 1, 5)

> family2
[1] "Andy"      "Lauren"    "Chili"     "1"         "5"
```

All elements in a vector have to be of the same type...if not they will be coerced to the same type

# Logical Vectors

In a **logical vector** each element is TRUE or FALSE. Logical elements are *not* delimited by quotation marks.

```
> myLogical = c(TRUE, FALSE, FALSE)
```

```
> age
```

```
[1] 40 37 9 2 10
```

```
> age > 10
```

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

Logical vectors are often generated through **conditional** statements

```
> people = (age > 10)
```

```
> length(people)
```

```
[1] 5
```

```
> people + 1
```

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

Logical elements have numeric values associated with them, namely 0 (FALSE) or 1 (TRUE).

# Data Structures: Data Frames

Data frames have a tabular (rectangular) structure made up of rows (cases) and columns (variables).

- Columns need to have the same length
- Columns can be of different vector types
- Columns have names

family	age	people
Andy	40	TRUE
Lauren	37	TRUE
Chili	9	FALSE
Sadie	2	FALSE
Einstein	10	FALSE

Names

Same length

# Creating Data Frames

You can create data frames by binding together existing vectors.

```
> myData = data.frame(family, age, people)
```

```
> myData
```

	family	age	people
1	Andy	1	FALSE
2	Lauren	6	TRUE
3	Chili	7	TRUE
4	Mojo	7	TRUE
5	Einstein	10	TRUE



```
> myData = data.frame(  
  family = c("Andy", "Lauren", "Chili", "Sadie", "Einstein"),  
  age     = c(40, 37, 9, 2, 10)  
)
```

Data frames can also be created from scratch.

```
> myData  
  family age  
1   Andy  40  
2  Lauren 37  
3   Chili  9  
4   Sadie  2  
5 Einstein 10
```

```
> myData$species = c("Person", "Person", "Dog", "Dog", "Dog")
```

```
> myData  
  family age species  
1   Andy  40  Person  
2  Lauren 37  Person  
3   Chili  9    Dog  
4   Sadie  2    Dog  
5 Einstein 10    Dog
```

New variables can be appended to existing data frames using the \$ operator.

## Script File: Part II

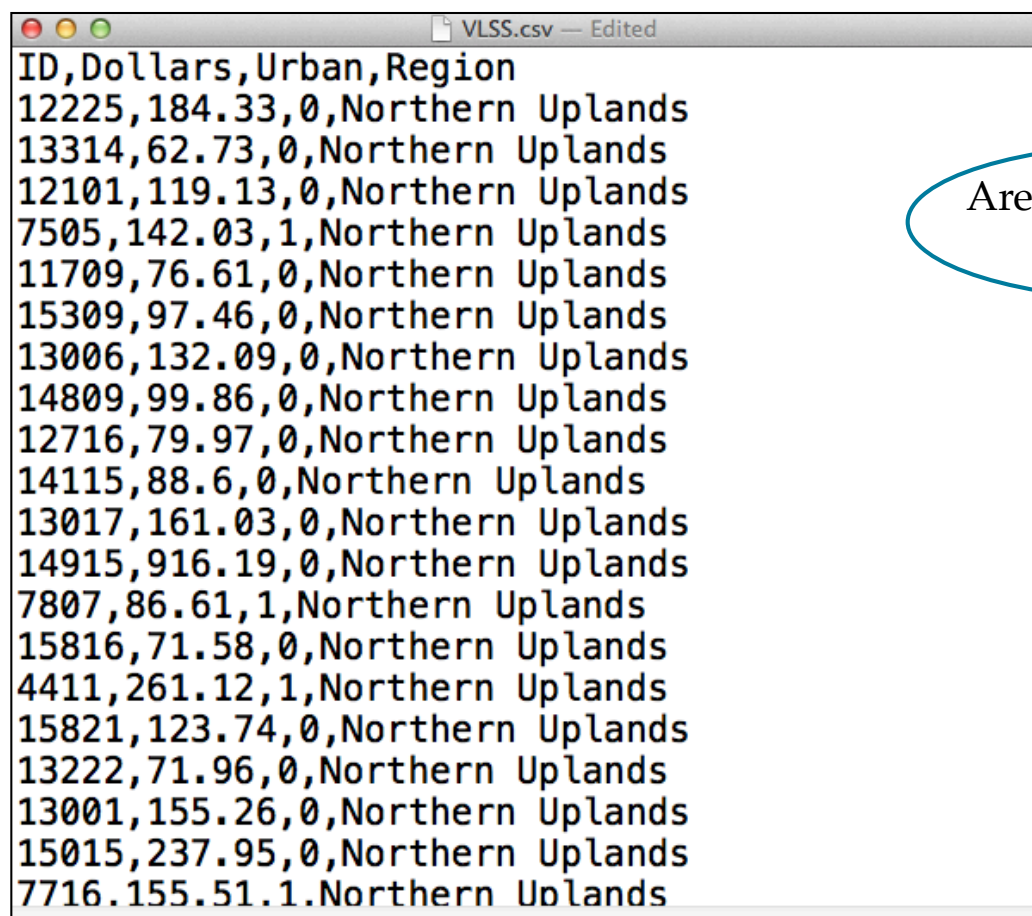
```
# Create data frame
myData = data.frame(
  family = c("Andy", "Lauren", "Chili", "Sadie", "Einstein"),
  age     = c(40, 37, 9, 2, 10)
)
```

good script files use  
indentation to ease readability!

```
# Assignment
myData$species = c("Person", "Person", "Dog", "Dog", "Dog")
```

# Importing External Data into RStudio

If you didn't enter the data, generally a good idea to examine the data in a text editor or browser.



```
ID,Dollars,Urban,Region
12225,184.33,0,Northern Uplands
13314,62.73,0,Northern Uplands
12101,119.13,0,Northern Uplands
7505,142.03,1,Northern Uplands
11709,76.61,0,Northern Uplands
15309,97.46,0,Northern Uplands
13006,132.09,0,Northern Uplands
14809,99.86,0,Northern Uplands
12716,79.97,0,Northern Uplands
14115,88.6,0,Northern Uplands
13017,161.03,0,Northern Uplands
14915,916.19,0,Northern Uplands
7807,86.61,1,Northern Uplands
15816,71.58,0,Northern Uplands
4411,261.12,1,Northern Uplands
15821,123.74,0,Northern Uplands
13222,71.96,0,Northern Uplands
13001,155.26,0,Northern Uplands
15015,237.95,0,Northern Uplands
7716.155.51.1.Northern Uplands
```

## Two Questions

Are variable names in the first row?

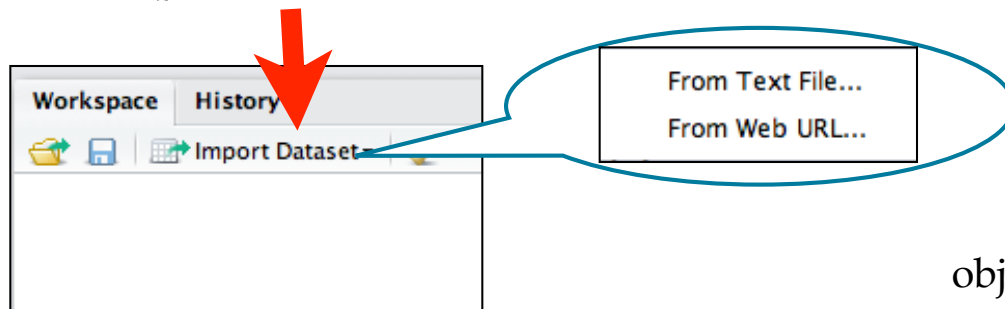
What is the delimiter?

### Tip

Enter data in a spreadsheet program (e.g., Excel). Leave a blank cell for missing data.

Save as a CSV file

Workspace pane:  
Import Dataset



object name

Are variable names in  
the first row?

What is the  
delimiter?

How your data  
will look in R

When you are happy,  
click **Import**

Import Dataset

Name: vlss

Input File

Heading: ☒ Yes ☐ No

Separator: Comma

Decimal: Period

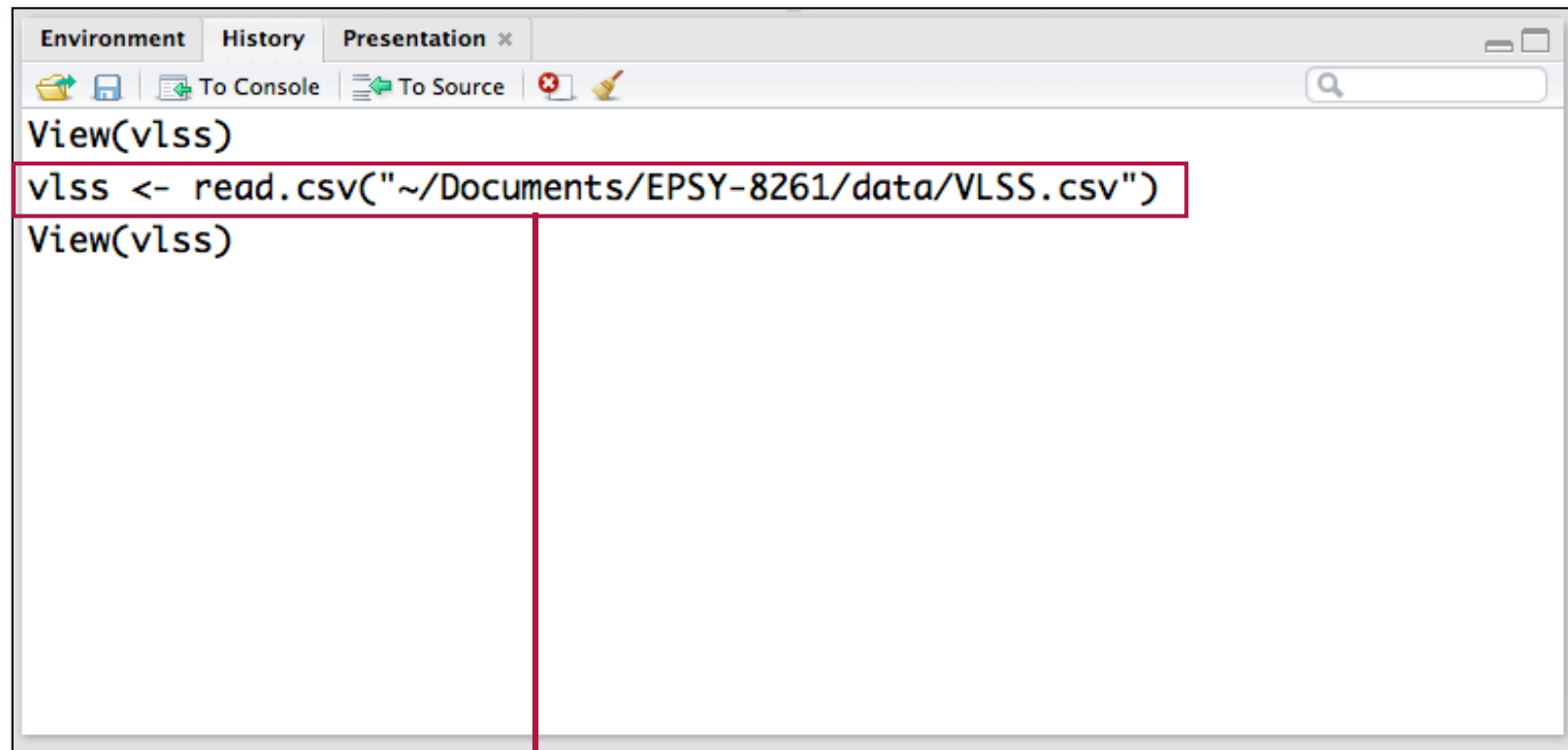
Quote: Double quote (")

Data Frame

ID	Dollars	Urban	Region
12225	184.33	0	Northern Uplands
13314	62.73	0	Northern Uplands
12101	119.13	0	Northern Uplands
7505	142.03	1	Northern Uplands
11709	76.61	0	Northern Uplands
15309	97.46	0	Northern Uplands
13006	132.09	0	Northern Uplands
14809	99.86	0	Northern Uplands
12716	79.97	0	Northern Uplands
14115	88.60	0	Northern Uplands
13017	161.03	0	Northern Uplands
14915	916.19	0	Northern Uplands
7807	86.61	1	Northern Uplands

Import Cancel

Immediately copy the syntax from  
the console pane into your script file  
and comment it...remember no  
prompts; no output




```
Environment History Presentation x
To Console To Source
View(vlss)
vlss <- read.csv("~/Documents/EPsy-8261/Data/VLSS.csv")
View(vlss)
```

```
# Read in VLSS data
vlss = read.csv("~/Documents/EPsy-8261/Data/VLSS.csv")
```

The import feature in RStudio is using the `read.csv()` function to read in an external file. The argument for this function is the file's path name given as a character string.

```
vlss = read.csv("~/Documents/EPsy-8261/Data/VLSS.csv")
```



Path name

The path name is the syntactic "address" for a file on your computer.

- Go into your home directory (~/)
- Go into the "Documents" folder (Documents/)
- Go into the "EPsy-8261" folder (EPsy-8261/)
- Go into the "Data" folder (Data/)
- Open the file called "VLSS.csv" (VLSS.csv)

Then, assign this file into the object `vlss`, which will be a data frame.

After reading in external data examine it.

```
> head(vlss)
      ID Dollars Urban      Region
1 12225   184.33     0 Northern Uplands
2 13314    62.73     0 Northern Uplands
3 12101   119.13     0 Northern Uplands
4  7505   142.03     1 Northern Uplands
5 11709    76.61     0 Northern Uplands
6 15309    97.46     0 Northern Uplands
```

```
> tail(vlss)
      ID Dollars Urban      Region
5994 10608   652.42     1 Mekong Delta
5995 35320   361.92     0 Mekong Delta
5996 35617   190.60     0 Mekong Delta
5997 33811   115.87     0 Mekong Delta
5998 34620   123.01     0 Mekong Delta
5999 38820   152.50     0 Mekong Delta
```

# Accessing Data Frame Elements: Indexing

Think of the data frames as a rectangle (matrix) made up of rows and columns.

This data frame is a 5999 x 4 rectangle.

ID	Dollars	Urban	Region
12225	184.33	0	Northern Uplands
13314	62.73	0	Northern Uplands
12101	119.13	0	Northern Uplands
7505	142.03	1	Northern Uplands
11709	76.61	0	Northern Uplands
:	:	:	:
35320	361.92	0	Mekong Delta
35617	190.60	0	Mekong Delta
33811	115.87	0	Mekong Delta
34620	123.01	0	Mekong Delta
38820	152.50	0	Mekong Delta



To access elements we give the “address” of the element within the rectangle, [*row*, *column*]. This is called **indexing**.

```
> vlss[3, 1]  
[1] 12101
```

This is the ID for the 3rd subject.  
(3rd row, 1st column)

```
> vlss[1, 3]  
[1] 0
```

This is Urban value for the 1st  
subject.  
(1st row, 3rd column)

```
> vlss[1, ]  
child age toddler  
1 Andy 1 FALSE
```

```
> myData[1, ]  
ID Dollars Urban Region  
1 12225 184.33 0 Northern Uplands
```

Omitting the row or column gives  
all the elements of the omitted part  
of the “address”. In this case all  
columns in the 1st row

# Accessing Variables (columns)

```
> vlss[ , 2]  
[Output not shown]
```

```
> vlss$Dollars  
[Output not shown]
```

```
> myData[[2]]  
[Output not shown]
```

```
> mean(vlss$Dollars)  
[1] 212.5778
```

You can also access columns by using the variable names and \$ operator.  
*dataframe\$variable*

You can also use list indexing, double square brackets.

# Indexing a Vector

Indexing also works on vectors.

```
> age  
[1] 40 37 9 2 10
```

```
> age[2]  
[1] 37
```

```
> myData$age[2]  
[1] 37
```

```
> myData[[2]][2]  
[1] 37
```

Since vectors only have one dimension, we only need to provide a single value in the “address”

# Adding Functionality: Packages (Libraries)

R functions are stored in packages (or libraries).  
There over 5,000 different packages available on  
CRAN (and more on gitHub, webpages, etc.)

## Base

c  
data.frame  
mean  
⋮


## graphics

boxplot  
plot  
legend  
⋮

## MASS

dropterm  
glmPQL  
⋮

...

Available CRAN Packages By Name		
ABCDEFGHIJKLMNOPQRSTUVWXYZ		
	<a href="#">abc</a>	Tools for Approximate Bayesian Computation (ABC)
<a href="#">CRAN</a>	<a href="#">abcdeFBA</a>	ABCDE_FBA: A-Biologist-Can-Do-Everything of Flux Balance Analysis with this package
<a href="#">Mirrors</a>	<a href="#">abd</a>	The Analysis of Biological Data
<a href="#">What's new?</a>	<a href="#">abind</a>	Combine multi-dimensional arrays
<a href="#">Task Views</a>	<a href="#">abn</a>	Data Modelling with Additive Bayesian Networks
<a href="#">Search</a>	<a href="#">AcceptanceSampling</a>	Creation and evaluation of Acceptance Sampling Plans
<a href="#">About R</a>	<a href="#">ACCLMA</a>	ACC & LMA Graph Plotting
<a href="#">R Homepage</a>	<a href="#">Ace</a>	Assay-based Cross-sectional Estimation of incidence rates
<a href="#">The R Journal</a>	<a href="#">acepack</a>	ace() and avas() for selecting regression transformations
<a href="#">Software</a>	<a href="#">acer</a>	The ACER Method for Extreme Value Estimation
<a href="#">R Sources</a>	<a href="#">aCGH.Spline</a>	Robust spline interpolation for dual color array comparative genomic hybridisation data
<a href="#">R Binaries</a>	<a href="#">ACNE</a>	Affymetrix SNP probe-summarization using non-negative matrix factorization
<a href="#">Packages</a>	<a href="#">acs</a>	Download and manipulate data from the US Census American Community Survey
<a href="#">Other</a>	<a href="#">Actigraphy</a>	Actigraphy Data Analysis
<a href="#">Documentation</a>	<a href="#">actuar</a>	Actuarial functions
<a href="#">Manuals</a>	<a href="#">ActuDistns</a>	Functions for actuarial scientists
<a href="#">FAQs</a>	<a href="#">ada</a>	ada: an R package for stochastic boosting
<a href="#">Contributed</a>	<a href="#">adabag</a>	Applies multiclass AdaBoost.M1, AdaBoost-SAMME and Bagging
	<a href="#">adagio</a>	Discrete and Global Optimization Routines
	<a href="#">AdaptFit</a>	Adaptive Semiparametric Regression
	<a href="#">AdaptFitOS</a>	Adaptive Semiparametric Regression with Simultaneous Confidence Bands
	<a href="#">adaptivetau</a>	Tau-leaping stochastic simulation

# Packages You (probably) Already Have

**Base**

**boot**

**class**

**cluster**

**codetools**

**compiler**

**datasets**

**foreign**

**graphics**

**grDevices**

**grid**

**KernSmooth**

**lattice**

**MASS**

**Matrix**

**methods**

**mgcv**

**nlme**

**nnet**

**parallel**

**rpart**

**splines**

**spatial**

**stats**

**stats4**

**survival**

**tcltk**

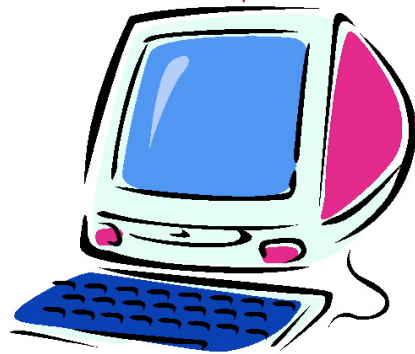
**tools**


**utils**

There are two distinct things you need to do to use the functions available in a package...**install** the package *and* **load** the package.



**Installing** the package takes it from the internet and puts it on your computer.



```
Console ~/   
  
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"  
Copyright (C) 2013 The R Foundation for Statistical Computing  
Platform: x86_64-apple-darwin10.8.0 (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.  
  
> |
```

**Loading** the package makes it useable during your R session

# Loading Packages that are Installed

The `library()` function is used to load packages that have previously been installed.

```
> library(MASS)
> library(survival)
Loading required package: splines
```

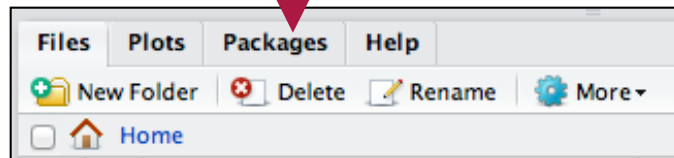
Some packages requires other packages (dependencies) to work. For example, the **survival** package is dependent on the **splines** package.

Once the package is loaded, all of the functions, data sets, etc. in that package are available to you.

Packages will need to be loaded every time you launch a new R session.

# Installing New Packages

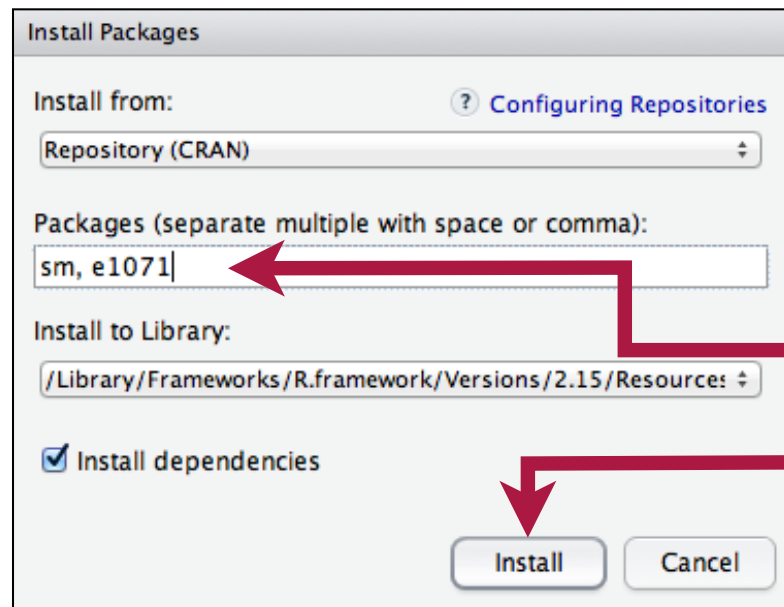
1. Click the **Package** tab in the **Files/Plots/Packages/Help** pane. This will bring up a list of all of the packages *installed* on your computer



2. To install a new package, click **Install Packages**



3. Enter the name of the package you would like to install in the text box. (Note that you can install more than one package at a time.) Click **Install**.



Packages will only need to be installed once.



# To-Do List

Install the following packages:

- arm
- devtools
- dplyr
- ggplot2
- ggthemes
- sm

Download the **cehd.csv** data set from the course website to your computer. Import the data into RStudio. If all goes well, the data frame should have six columns (variables) and 166 rows (cases).

Open a new script file and record and comment on the syntax for each of the following:

- Compute the mean annual base salary for the CEHD faculty.
- Compute the standard deviation for the 166 salaries.
- Load the sm package and use the `sm.density()` function to plot the annual base salaries for all 166 faculty members. Export the plot, and import it into a word-processing program.
- Use indexing to view the 100th row.
- Use the `table()` function to count the number of faculty of each title.