# Spoon-Fed R

author: Matt Shirley date: October 24 2013

# Overview

1. interacting with R
2. using R as a calculator
3. variables
4. data structures
5. summarizing data
6. loops, flow-control
7. apply
8. basic stats in R
9. reading and writing delimited data
10. plotting with base R graphics
11. loading and installing packages
12. plotting with ggplot2

# interacting with R

- command-line interpreter
- GUI interpreter: RStudio

# command-line interpreter

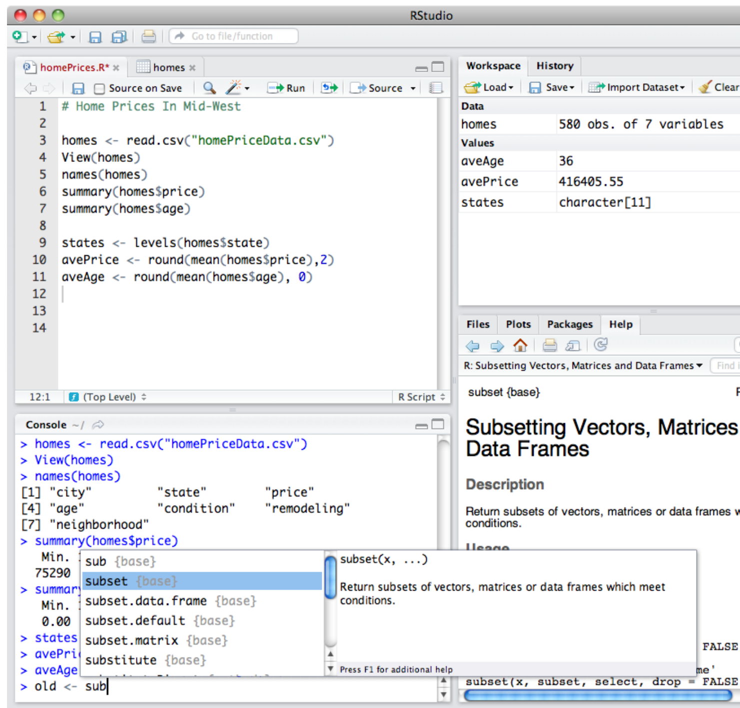
- everyone has one
- just type `R` at your command-line shell:

"' R version 3.0.2 – "Frisbee Sailing" Platform: x86_64-apple-darwin13.0.0 (64-bit) . . .

Type 'q()' to quit R.

> "`- The carat (`>') is your prompt for entering commands - I will omit the carat for the rest of the presentation

## GUI interpreter: RStudio




- Download from http://www.rstudio.com/

## GUI interpreter: RStudio

RStudio is an integrated development environment including: - interpreter with code completion - text editor with syntax highlighting and completion - file browser - version control manager - visual object workspace - command history

## the R interpreter

`{r} # This is a comment, which is ignored` `{r} # functions are applied with () print("hello")` - anything in quotes is a "string" - anything else is either a number or: - function - class - operator (`+-/?%&=<>|!^*`)

## using R as a calculator

Addition `{r} 2 + 2` Subtraction `{r} 5 - 2`

## using R as a calculator

Division `{r} 2 * 2` Multiplication `{r} 5 / 2`

## using R as a calculator

Exponents `{r} 2^4` Logorithms `{r} log10(100) log2(4)`

## using R as a calculator

Order of operations `{r} 10 / 2 - 1 10 - 5 / 5 (10 - 5) / 5` Be careful. Evaluation of operators occurs left to right.

## variables

`{r} x <- 1 x` Variables can be assigned (`<-`) a value

## variables

`{r} x <- 1 y <- 2 x <- y x y` But be **careful** because they can be re-assigned

## data structures: types of data

`{r} typeof(1) typeof("A") typeof(TRUE)`

## data structures: types of data

`{r} as.numeric("1") as.character(1) as.logical(1)`

## data structures: comparisons

`{r} x <- 0` `{r} x > 1 ## x is greater than 1 x < 1 ## x is greater than 1`

## data structures: comparisons

`{r} x == 1 x == 0 x != 0` Comparisons result in *logical* values

## data structures: vectors

`{r} x <- 3 y <- c(1,2,x) y` Vectors can hold elements of the *same type*.

## data structures: vectors

`{r} names(y) <- c("one", "two", "three") y` Vectors can also have *names* for each element.

## data structures: vectors

`{r} z <- y * 3 z sum(z)` Arithmetic can be performed on a vector, which applies that operation to every element and returns a *new vector*.

## data structures: vector indexing

`{r echo=F} z` `{r} z[1] z["one"]` Vectors can be indexed using a *1-based* position, as well as *name*.

## data structures: vector slicing

`{r} z z[2:3]` *Slicing* a vector is as easy as specifying `start:end`.

## data structures: vector slicing

`{r} z[-1] z[-2:-3]` Remove elements from a vector using negative indices.

## data structures: lists

`{r} q <- list(y, z) q` Lists can contain vectors.

## data structures: list indexing

`{r} q[[1]] q[[1]][1]` You can index a list in the same way as a vector.

## data structures: sequences

`{r} v <- seq(1,9) ## or 1:9 v` Let's construct a sequence of 9 numbers.

## data structures: sequences

`{r} c(v,v) rep(v, times=3)` We can *concatonate* or *repeat* a vector as well.

## data structures: matrices

`{r} mt <- matrix(v, nrow=3) mt matrix(v, nrow=3, byrow=T)` Matrices, created from vectors, are row or column oriented.

## data structures: matrix indexing

`{r echo=F} mt {r} mt[1,1] mt[3,3]` Matrices are indexed as `[row,col]`

## data structures: dimension

`{r} dim(mt) nrow(mt) ncol(mt)` Dimensionality, number of rows and columns can computed using these functions.

## data structures: dataframes

`{r} df <- data.frame(y, z) colnames(df) <- c("first","second") df` Dataframes are like matrices, but contain more structure.

## data structures: dataframe indexing

`{r echo=F} df {r} df$first` Dataframes can be indexed by name to return a vector.

## data structures: dataframe indexing

`{r echo=F} df` `{r} df["first"]` Dataframes can be indexed by name to return another dataframe

## data structures: dataframe indexing

`{r echo=F} df` `{r} df$first[1]` Dataframes can be further indexed to return individual elements

## data structures: logical indexing

`{r echo=F} df` `{r} df > 3` Dataframes, just like other structures, can be compared, resulting a *logical* values.

## data structures: logical indexing

`{r echo=F} df > 3` `{r} df[df > 3]` Passing the logical result of comparison as an index returns only elements where the comparison was `TRUE`.

## data structures: logical indexing

`{r echo=F} df > 3` `{r} which(df > 3)` The `which` function converts a boolean index to a numeric index.

## data structures: dataframe binding

`{r echo=F} df` `{r} cbind(df, data.frame("third"=c(9,18,27)))` Dataframe columns can be bound to form a new dataframe.

## data structures: dataframe binding

`{r echo=F} df` `{r} rbind(df, data.frame("first"=4, "second"=12, row.names="four"))` Dataframe rows can be bound to form a new dataframe.

# summarizing data

`{r} library(datasets) dim(cars) head(cars)`

# summarizing data

`{r} mean(cars$speed) median(cars$speed) sd(cars$speed)` Mean, median and standard deviation.

# summarizing data

`{r} summary(cars)` Summarizing a dataframe returns percentiles and mean.

# loops, flow-control: for loops

`{r} for (x in 1:10){ print(x) }` Use *for loops* to repeat a task a certain number of times.

# loops, flow-control: if/else

`{r} x <- 0 if (x == 0) { print("yes") } if (x > 1) { print("yes") } else { print("no") }` - If statements only execute code if the condition evaluates to `TRUE`. - Else statements execute when the condition is not satisfied.

# loops, flow-control: while loops

`{r} x <- 0 while (x < 5){ print(x) x <- x + 1 }` Use *while loops* to repeat a task *while* a condition (`x<5`) is true.

# apply: functional application

`{r echo=F} df` `{r} apply(df, 1, sum) apply(df, 2, sum)` *Apply* a function over array columns (1) or rows (2).

## sapply: simpler apply

`{r echo=F} df` `{r} sapply(df, sqrt)` *Simple apply* a function to every element, returning the same type of data structure.

## reading and writing delimited data

`{r} write.table(df, file = "example.txt") write.table(df, file = "example.tsv", sep = "\t") write.csv(df, file = "example.csv")`
Write 1) space-delimited, 2) tab-delimited, 3) comma-delimited files containing dataframe `df`.

## reading and writing delimited data

`{r} df1 = read.table("example.txt", header=T) df2 = read.delim("example.tsv", sep = "\t") df3 = read.csv("example.csv", row.names = 1)` `{r} identical(df1,df2) identical(df2,df3)` All three files result in equivalent dataframes.

## reading and writing delimited data

Issues to consider when reading and writing delimited files:

1. Do I want/have column names (header)?
2. Do I want/have row names?
3. What is my delimiter?
4. Do I want/have quotes surrounding each value?

Check the **default behavior** of the reading/writing function first.

## plotting with base R graphics

`{r} head(cars)`

## plotting with base R graphics: scatterplot

`{r} plot(cars)` * - `plot` accepts a dataframe with two columns - column 1 = x axis - column 2 = y axis

## plotting with base R graphics: line plot

`{r} plot(cars, type="l")` * - valid plot types: - "p" for points - "l" for lines - "b" for both ("o" for overplotted) - "h" for 'histogram'-like lines - "s" for stair steps ("S" for other) - "n" for no plotting.

## plotting with base R graphics: linear regression

`{r} lmcars <- lm(dist ~ speed, cars) lmcars` - `lm` fits a linear model: `response ~ terms` - in this case the response is distance traveled at speed

## plotting with base R graphics: linear regression

`{r} plot(cars) abline(lmcars)` * - `abline` draws a line from slope and intercept

## plotting with base R graphics: graphics parameters

`{r} plot(cars, title="Speed vs. Distance", xlab="Speed", ylab="Distance", ylim=c(0,100)) abline(lmcars)`

## plotting with base R graphics: graphics parameters

`{r} plot(cars, col="red", pch=16, cex=2) abline(lmcars, col="blue")`

## plotting with base R graphics: histograms

`{r} hist(cars$speed)`

## plotting with base R graphics: boxplots

`{r} boxplot(cars)` ** - *Outliers are defined as outside 1.5*IQR - IQR = interquartile range

## plotting with base R graphics: PCA

`{r} pcars <- prcomp(cars) biplot(pcars)` * - principal components analysis of variance - biplot of the first (and therefore largest) components - vector arrows represent magnitude of contribution of each variable

## loading and installing packages

`{r eval=FALSE} library('stats') - library()` loads an R *package* into your current session - This will import all *functions* from that package for your use - If you don't have the package installed, R will complain: `{r eval=FALSE} library('foo') Error in library("foo") : there is no package called 'foo'` So we must install the package…

## loading and installing packages

Install from CRAN (Comprehensive R Archive Network) repositories `{r eval=FALSE} install.packages('ggplot2')` Install from Bioconductor `{r eval=FALSE} source("http://bioconductor.org/biocLite.R") biocLite('GRanges')` Install from GitHub `{r eval=FALSE} install.packages('devtools') library(devtools) install_github('ballgown')`

## plotting with ggplot2

- ggplot = *grammar* of *graphics*
- combines statistical and graphical models
- can create very concise, detailed plots using few keystrokes `{r} library("ggplot2")`

# plotting with ggplot2

`{r} head(mtcars[c("wt","mpg","cyl","disp")]) p <- ggplot(mtcars, aes(wt, mpg))` - building a plot starts with a dataframe - aestetics (aes) are columns of the dataframe - usually corresponts to x & y axis

# plotting with ggplot2: add a geometry

`{r} p + geom_point()` * - now we add a geometry - calling the plot produces the graphics

# plotting with ggplot2: geometry aes

`{r} p + geom_point(aes(colour = factor(cyl)))` * - we can add aestetics to the geometry - in this case, color the points by number of cylinders

# plotting with ggplot2: geometry aes

`{r} p + geom_point(aes(shape = factor(cyl)), size=6, alpha=I(0.5))`

# plotting with ggplot2: boxplots

`{r} p <- ggplot(mtcars, aes(factor(cyl), mpg)) p + geom_boxplot()`

# plotting with ggplot2: barplots

`{r} ggplot(diamonds, aes(clarity, fill=cut)) + geom_bar()`