RStudio Lab Codes and Functions

Contents

Loading, saving and viewing data . . . 1
Scraping web data . . . 3
Data manipulation functions . . . 3
Numerical summaries and frequency tables . . . 5
Plotting functions . . . 7
Maps . . . 11
Statistical modeling . . . 12
Sampling and permutation functions . . . 13
Probability functions . . . 14
Symbols and operators . . . 15
Creating functions . . . 15

Loading, saving and viewing data

`data()`: Loads and displays a pre-loaded data file from RStudio.

Example:

```
data(cdc)
```

`read.csv()`: Imports data from a *.csv* formatted file into R.

Example:

```
read.csv("Time Use.csv")
```

`View()`: Displays the data as a spreadsheet in a new tab.

Example:

```
View(cdc)
```

`head()`: Prints the first 6 values or rows of data in the console.

Examples:

```
# Observations of a dataset
head(cdc)
```

```
# Observations of a variable
head(~gender, data = cdc)
```

`tail()`: Prints the last 6 values or rows of data in the console.

Examples:

```
# Observations of a dataset
tail(cdc)
```

```
# Observations of a variable
tail(~gender, data = cdc)
```

`dim()`: Prints the number of rows and columns of a data file.

Example:

```
dim(cdc)
```

`nrow()`: Prints the number of rows (observations) in a data file.

Example:

```
nrow(cdc)
```

`ncol()`: Prints the number of columns (variables) in a data file.

Example:

```
ncol(cdc)
```

`length`: Prints the number of elements in a vector.

Example:

```
length(1:25)
```

`names()`: Prints the variable names of a data file.

Example:

```
names(cdc)
```

`str()`: Prints the structure of a data file.

- Information displayed includes the number of observations and variables, the names of the variables, the type of variables and the first few observations.

Example:

```
str(cdc)
```

`save()`: Saves a data file to the *Files* pane.

- When using the save function, the `file` argument is required to tell R where to save your new data and what to call it.

Example:

```
# Subset the cdc data to include only males and then save
cdc_males <- subset(cdc, gender == "Male")
save(cdc_males, file = "cdc_male.Rda")
```

`load()`: Loads a previously saved R data file from the *Files* pane.

Example:

```
load("cdc_male.Rda")
```

Scraping web data

`readHTMLTable()`: A function used to scrape basic web tables from a URL.

- *Option*: `which`. When many tables are scraped from a page, use `which` to designate exactly which table to extract.
- *Option*: `colClasses`. Specify, for a single table, the class of variables (`"factor"` for categorical, `"numeric"` for numerical).

Examples:

```
# The url of the 200 tallest peaks in the US
mtns_url <- "https://tinyurl.com/usamtns"
# Scrape all tables from a webpage
tables <- readHTMLTables(mtns_url)
# Scrape just the first table from a webpage
table_one <- readHTMLTables(mtns_url, which = 1)
# Scrape the first table and specify which variables are which types
var_types <- c("factor", "factor", "factor", "numeric", "numeric", "numeric", "numeric",
               "numeric", "numeric", "numeric")
mtns <- readHTMLTable(mtns_url, which = 1, colClasses = var_types)
```

Data manipulation functions

`c()`: Used to combine values together.

Example:

```
# Combine values 0, 1 & 20 together into a vector and then compute their mean
new_vector <- c(0, 1, 20)
mean(new_vector)
```

`rep()`: Create a vector of repeated values. - *Option*: `times`. The number of times to repeat. - *Option*: `each`. The number of times to repeat each element of a vector.

Examples:

```
# Basic usage
rep("A", times = 5)
```

```
# Repeat a vector of inputs 5 times. Notice the order of the output differs
# from the example below
rep(c("A", "B"), times = 5)
```

```
# Repeat each element of a vector 5 times. Notice the order of the output differs
# from the example above.
rep(c("A", "B", each = 5))
```

`subset()`: Used to create a smaller dataset where each observation abides but a specific rule.

- Rules can be created using the relationship operators. See the **Symbols** section for more info on them.

Examples:

```
# Subsetting based on a categorical variable
subset(cdc, gender == "Male")
```

```
# Subsetting based on a numerical variable
subset(cdc, height > 1.6)
```

`mutate()`: Used to change or add variables in a data file. Can also be used with the following functions to change the class of the variable:

- `as.numeric()`: Used to change a categorical variable to a numerical variable.
- `as.character()`: Used to change a numerical variable to a categorical variable.
- `revalue()`: Used to change the names of categories for a categorical variable.

Examples:

```
# Convert meters to feet
mutate(cdc, height = height * 3.28084)
```

```
# Create a new variable called ratio
mutate(cdc, ratio = height/weight)
```

```
# Overwrite the cdc data to change height to a categorical variable
cdc <- mutate(cdc, height = as.character(height))
```

```
# Overwrite the cdc data again but changing the height variable back to numerical
cdc <- mutate(cdc, height = as.numeric(height))
```

```
# Change the categories of the asthma variable from Yes/No to 1/0
mutate(cdc, asthma = revalue(asthma, c("Yes"="1", "No" = "0")))
```

`slice()`: Extract rows of data based on row numbers. *Note*: To extract everything EXCEPT some number of rows, use a minus sign. See the examples below.

Examples:

```
# Extract 1st 3 rows of the cdc data
slice(cdc, 1:3)
```

```
# Extact a random sample of 1,000 rows from the cdc data
rows <- sample(1:nrow(cdc), size = 1000, replace = FALSE)
slice(cdc, rows)
```

```
# Extract every row EXCEPT those sampled from the above example
slice(cdc, -rows)
```

`rename()`: Used to rename the variables of a data file.

Example:

```
rename(cdc, HEIGHT = height)
```

`summarize()`: Can be used to display information about specific variables in a data file. The following functions can be used with `summarize()`:

- `levels()`: Used to print out the categories of a categorical variable.

Example:

```
summarize(cdc, levels(age))
```

`timeuse_format()`: Takes the downloaded *Time Use* data and formats it for use in the labs.

Example:

```
timeuse_format(TimeUseData)
```

`merge()`: Merge two data frames together. *Note*: When merging data, it's best if there's a one-to-one or many-to-one connection between the data. Meaning that one data set has an *id* variable where each individual *id* occurs only once in the data.

Example:

```
# Load the Personality Colors data and name it colors.
# Load the Stress data and name it stress.
# NOTICE: Personality Color data has a unique user in each row.
stress_colors <- merge(stress, colors, by = "user.id")
```

Numerical summaries and frequency tables

Note: The numerical summary functions will print a warning when the data they're using contains missing values, denoted as `NA`. The warnings are so the user doesn't mistakenly believe they are calculating a summary of all observations in a data set when in fact they are only summarizing a small subset of the observations.

`favstats()`: Computes the mean, median, Q1, Q3, max, min, sd, number of observatins (n) and number of missing values (missing) for a variable.

Examples:

```
# Basic usage
favstats(~height, data = cdc)
```

```
# Groupwise favstats
favstats(~height | gender, data = cdc)
```

`mean()`: Calculates the average of a numerical variable.

Examples:

```
# Basic usage
mean(~height, data = cdc)
```

```
# Calculate group means
mean(~height | gender, data = cdc)
```

`median()`: Calculates the value that falls in the middle of an ordered set of numbers.

Examples:

```
# Basic usage
median(~height, data = cdc)
```

```
# Calculate group means
median(~height | gender, data = cdc)
```

`diff()`: Calculates the difference between two or more values.

Examples:

```
# Create a vector of values
values <- c(1, 2, 4, 5)
```

```
# Use diff() to calculate pairwise differences between pairs of values.
diff(values)
```

`MAD()`: Calculates the *mean absolute deviation*, meaning the average (positive) distance each observation is from the mean.

Examples:

```
# Basic usage
MAD(~height, data = cdc)
```

```
# Compute the groupwise MAD
MAD(~height | gender, data = cdc)
```

`range()`: Calculates the minimum and maximum values of a variable.

Examples:

```
# Basic usage
range(~height, data = cdc)
```

```
# To compute the statistical range, that is the max minus min value:
rng <- range(~height, data = cdc)
diff(rng)
```

`max()`: The largest value of a numerical variable.

Examples:

```
max(~height, data = cdc)
```

`min()`: The smallest value of a numerical variable.

Examples:

```
min(~height, data = cdc)
```

`quantile()`: Computes the value for which `p` percent of the data are smaller than.

Examples:

```
# Calculate the value for which 25% of the age variable in the cdc data are smaller than.
quantile(~height, data = cdc, p = 0.25)
```

```
# Calculate the value for which 75% of the age variable in the cdc data are smaller than.
quantile(~height, data = cdc, p = 0.75)
```

`iqr()`: Calculate the difference between the 75th and 25th percentiles.

Examples:

```
# Basic usage
iqr(~height, data = cdc)
```

```
# Calculate the groupwise IQRs
iqr(~height | gender, data = cdc)
```

`cor()`: Calculate the correlation between two variables.

Examples

```
cor(audience_rating ~ critics_rating, data = movie)
```

`tally()`: Creates a one-way or two-way frequency table for categorical variables.

- *Option*: `format`. Can be set to `"count"`, `"proportion"`, `"percent"`
- *Option*: `margins`. Can be set to `TRUE`, `FALSE`. Includes column-totals when set to `TRUE`.

Examples:

```
# One-way table
tally(~asthma, data = cdc)
```

```
# Two-way table
tally(gender ~ asthma, data = cdc)
```

```
# Two-way table with options
tally(gender ~ asthma, data = cdc, margins = TRUE, format = "percent")
```

```
# The order of the variables is important as swapping them results in different tables.
tally(asthma ~ gender, data = cdc, margins = TRUE, format = "percent")
```

Plotting functions

`histogram()`: Creates a visual display of a numerical variable.

- *Option*: `type`. Can be set to `"count"`, `"percent"`, `"density"`
- *Option*: `nint`. The number of intervals or bars to use.
- *Option*: `fit`. Overlays a probability curve over the data. Most often used with `"normal"`.
- *Option*: `layout`. Used to arrange the individual plots when faceting. Use with the `c()` function to control the number of columns (first number) and rows (second number).

Examples:

```
# Basic usage
histogram(~height, data = cdc)
```

```
# With faceting and horizontal layout
histogram(~height | gender, data = cdc, layout = c(2, 1))
```

```
# Including options
histogram(~height, data = cdc, type = "percent", nint = 24, fit = "normal")
```

`dotPlot()`: Creates a visual display of a numerical variable.

- *Option*: `nint`. The number of intervals or bars to use.
- *Option*: `fit`. Overlays a probability curve over the data. Most often used with `"normal"`.
- *Option*: `cex`. The *character expansion* option can be used to make the dots larger or smaller.
- *Option*: `layout`. Used to arrange the individual plots when faceting. Use with the `c()` function to control the number of columns (first number) and rows (second number).

Examples:

```
# Having too many data points makes the dots hard to see so we'll take a sample
cdc_sample <- sample(cdc, size = 100)
```

```
# Basic usage
dotPlot(~height, data = cdc_sample)
```

```
# With faceting and stacked layout
dotPlot(~height | gender, data = cdc_sample, layout = c(1, 2))
```

```
# Including options and making dots smaller
dotPlot(~height, data = cdc_sample, nint = 24, fit = "normal", cex = 0.5)
```

```
# Including options and making dots larger
dotPlot(~height, data = cdc_sample, nint = 24, fit = "normal", cex = 1.5)
```

`bargraph()`: Creates a visual display of a categorical variable.

- *Option*: `group`. Used to create a split bargraph based on a categorical variable.
- *Option*: `horizontal`. Set equal to `TRUE` to make bars horizontal.
- *Option*: `layout`. Used to arrange the individual plots when faceting. Use with the `c()` function to control the number of columns (first number) and rows (second number).

Examples:

```
# Basic usage
bargraph(~age, data = cdc)
```

```
# With faceting and 3-column layout
bargraph(~age | asthma, data = cdc, layout = c(3, 1))
```

```r
# With faceting on two variables and 3-column 2-row layout.
bargraph(~age | asthma + gender, data = cdc, layout = c(3, 2))
```

```r
# Including options
bargraph(~age, data = cdc, groups = asthma, horizontal = TRUE)
```

`xyplot()`: Creates a visual display of two numerical variables.

- *Option*: `group`. Plots points with different colors based on a categorical variable.
- *Option*: `layout`. Used to arrange the individual plots when faceting. Use with the `c()` function to control the number of columns (first number) and rows (second number).

Examples:

```r
# Basic usage
xyplot(weight ~ height, data = cdc)
```

```r
# With faceting and 3-column layout
xyplot(weight ~ height | asthma, data = cdc, layout = c(3, 1))
```

```r
# With options
xyplot(weight ~ height, data = cdc, group = gender)
```

`add_line()`: Adds a line to an `xyplot` either by running the command with no arguments and then clicking twice on the plot plane or by supplying a slope and intercept. - *Option*: `slope`. The slope of the line you'd like to have plotted. - *Option*: `intercept`. The intercept of the line you'd like to have plotted.

Examples:

```r
# After making an xyplot, run the following and then click twice on the plot pane
# to add a line
add_line()
```

```r
# Alternatively, after making an xyplot you could use the following to add a line with
# slope = 5 and intercept = -2
add_line(slope = 5, intercept = -2)
```

`add_curve()`: Adds a curve to an `xyplot` based on a linear model (`lm`) with one explanatory and one response variable.

- *Option*: `col`. Draw the line in a different color.

Examples:

```r
# add_curve() can perform the same job as add_line()
model <- lm(height ~ armspan, data = arm_span)
xyplot(height ~ armspan, data = arm_span)
add_curve(model)
```

```r
# add_curve() can also plot curves from linear models using polynomials
model <- lm(height ~ poly(armspan, degree = 2), data = arm_span)
xyplot(height ~ armspan, data = arm_span)
add_curve(model)
```

```r
# Use col to change the line/curve's colors
model <- lm(height ~ poly(armspan, degree = 2), data = arm_span)
```

```
xyplot(height ~ armspan, data = arm_span)
add_curve(model, col = "red")
```

`treeplot()`: Draw the tree of a classification or regression tree.

Examples:

```
# Create a tree model
model <- tree(survived ~ gender + age + class + embarked, data = titanic)
# Then use treeplot() to plot it
treeplot(model)
```

`bwplot()`: Creates a box-and-whisker plot of a variable.

- *Option*: `layout`. Used to arrange the individual plots when faceting. Use with the `c()` function to control the number of columns (first number) and rows (second number).

Examples:

```
# Basic usage
bwplot(~height, data = cdc)
```

```
# Two-ways to facet:
# Method 1
bwplot(~height | gender, data = cdc)

# Method 2
bwplot(gender ~ height, data = cdc)
```

```
# With options
bwplot(~height | gender, data = cdc, layout = c(1, 2))
```

`plotdist()`: Plot the curve of a probability distribution

- *Option*: `dist`. The distribution you want to plot. In the curriculum, we use `"norm"` for the normal distribution. See the examples for a few others.
- *Option*: `mean`. For the normal distribution, the value to center the distribution around.
- *Option*: `sd`. For the normal distribution, the value of the spread for the distribution.

Examples:

```
# Standard normal distribution
plotDist("norm", mean = 0, sd = 1)
```

```
# mean 10, sd 4 normal distribution
plotDist("norm", mean = 10, sd = 4)
```

```
# A binomial distribution with n = 25, p = 0.25
plotDist('binom', size = 25, prob = .25)
```

```
# A beta distribution because, why not?
plotDist('beta', shape1 = 3, shape2 = 10)
```

Maps

`colorize()`: A simple function to add colors to maps

Example:

```
# Load the mountains data and call it mtns_data
# Add the colors for each state by mutating the data
mtns_data <- mutate(mtns_data, state_colors = colorize(state))
```

`leaflet()`: Input data to make maps via the leafet mapping functions.

Example:

```
# Load the mountains data and call it mtns_data
mtns_leaf <- leaflet(mtns_data)
```

`addTiles()`: Add map tiles to a leaflet leaf

Example:

```
# From the leaflet() example
mtns_map <- addTiles(mtns_leaf)
```

`addMarkers()`: Add icons to denote the location of data on a map

- *Option*: `map`. The map to add markers to.
- *Option*: `lng`. The longitude values of points.
- *Option*: `lat`. The latitude values of points.
- *Option*: `popup`. Values that will popup when a marker is clicked

Example:

```
# From the addTiles() example
addMarkers(map = mtns_map, lng = ~long, lat = ~lat)
```

```
# From the the example above, but with the name of the peaks as popups
addMarkers(map = mtns_map, lng = ~long, lat = ~lat, popup = ~peak)
```

`addCircleMarkers()`: Add circles, instead of icons, to denote the location of data on a map

- *Option*: `map`. The map to add markers to.
- *Option*: `lng`. The longitude values of points.
- *Option*: `lat`. The latitude values of points.
- *Option*: `popup`. Values that will popup when a marker is clicked
- *Option*: `color`. Include colors for different circles.

Example:

```
# From the addTiles() example
addMarkers(map = mtns_map, lng = ~long, lat = ~lat)
```

```
# From the the example above, but with the name of the peaks as popups
addMarkers(map = mtns_map, lng = ~long, lat = ~lat, popup = ~peak)
```

```
# From the the example above as well as the example for colorize()
addMarkers(map = mtns_map, lng = ~long, lat = ~lat, popup = ~peak, color = ~state_colors)
```

`addLegend()`: Include a legend for a map. Used with the `unique()` function to determine the unique values of `colors` and `labels`.

- *Option*: `colors`. The value of colors used in the map.
- *Option*: `labels`. The label of variables used in the map.

Example:

```
# From the last example for addCircleMarkers():
mtns_map <- addMarkers(map = mtns_map, lng = ~long, lat = ~lat,
                       popup = ~peak, color = ~state_colors)
# To include a legend
addLegend(mtns_map, colors = ~unique(state_colors), labels = ~unique(state))
```

Statistical modeling

`lm()`: Create a line of best fit, i.e. linear model, using ordinary least squares.

Examples:

```
# For a simple linear regression where we want to explain a variable, height,
# in terms of an expanatory variable, armspan.
lm(height ~ armspan, data = arm_span)
```

```
# For a multiple linear regression, include more variables using a "+" sign.
lm(domest_gross ~ critics_rating + runtime, data = movie)
```

`poly()`: Model a variable as a polynomial instead of a line in a linear model (`lm`). *Note*: The `poly()` function coefficients aren't directly interpretable because it orthogonalizes the polynomial before estimating the coefficients.

- *Option*: `degree`. The degree of the polynomial you want to use.

Examples:

```
# To fit a quadratic polynomial for person's armspan
lm(height ~ poly(armspan, degree = 2), data = arm_span)
```

`predict()`: For a linear model (`lm`) or a tree model (`tree`), give the predicted values for the data used to create the model or the values of a different data set with the same variables (i.e. testing data).

- *Option*: `newdata`. When using a testing data set, specify the data set with this option.

```
# Make a model using lm() or tree() and give the model a name (I'll assume
# you've named it "model"). To give the predicted values for the data used
# to create the model use:
predict(model)
```

```
# We could also predict the values of a testing data set
predict(model, newdata = testing_data_set)
```

`tree()`: Create a tree based model.

- *Option*: `cp`. The complexity parameter to use for the tree. Smaller numbers results in more complex trees, that is, trees with more branches. Default value is `0.01`.
- *Option*: `minsplit`. The minimum number of observations that need to be in a leaf before splitting the leaf into a new branch. Default value is `20`.

Examples:

```
# Create a simple classification tree
tree(survived ~ gender, data = titanic)
```

```
# Create a classification tree with additional explanatory variables
tree(survived ~ gender + age + class + embarked, data = titanic)
```

```
# Include options to create more complex trees
tree(survived ~ gender + age + class + embarked, data = titanic, cp = 0.005, minsplit = 10)
```

`kclusters()`: Cluster 2 variables into `k` clusters using the k-means algorithm.

- *Option*: `k`. The number of clusters to group the data into.

Examples:

```
# Group data into 2 clusters
kclusters(ht_inches ~ wt_lbs, data = futbol, k = 2)
```

Sampling and permutation functions

`sample()`: Sample from a set of values.

- *Option*: `size`. The number of times you want to sample.
- *Option*: `replace`. After a value has been sampled, should the sampled value be replaced so that it can be sampled again.

Examples:

```
# Create a vector to sample from
values <- c("A", "B", "B", "C")
# Basic usage
sample(values, size = 1)
```

```
# Sample 3 values without replacement
sample(values, size = 3)
```

```
# Sample values with or without replacement
sample(values, size = 4, replace = TRUE)
sample(values, size = 4, replace = FALSE)
```

`set.seed()`: Perform random functions in a systematic way, meaning, if two or more users set the same seed, they can generate the same random values.

Examples:

```r
# Any user who runs the following code will generate the same random sample.
set.seed(531)
sample(1:3, size = 10, replace = TRUE)
## [1] 1 2 1 2 1 3 2 2 1 3
```

`do()`: Perform an operation many times, often with using the `shuffle()` function. - *Option*: `shuffle`. Inside of a `do()` call, `shuffle()` will shuffle the values of a given variable so that any relationship between variables is nullified.

Examples:

```r
# Compute the mean of the height of people 5 times BUT shuffle the genders of our
# observations before computing the mean each time.
do(5) * mean(~height | shuffle(gender), data = cdc)
```

`rowSums()`: Count the instances that follow a rule from a `do()` operation.

Examples:

```r
# Create some values to sample from
values <- c("A", "B", "C")

# Use a do() operation to sample 2 letters, 5 times
draws <- do(5) * sample(values, size = 2, replace = FALSE)

# Count the number of times both draws were "A"
draws <- mutate(draws, n_A == rowSums(draws == "A"))

# Subset the data based on the number of rows where n_A is equal to 2.
draws_sub <- subset(draws, n_A == 2)

# Calculate the proportion of two sample draws that were both A's
nrow(draws_sub) / 5
```

Probability functions

`pnrom()`: Calculate the probabilities of values for a normal distribution.

- *Option*: `mean`. The mean of the normal distribution
- *Option*: `sd`. The standard deviation of the normal distribution

Examples:

```r
# The probability of a value less than 0.5 for a standard normal distribution
pnorm(0.5, mean = 0, sd = 1)
```

```r
# The probability of a value greater than 0.5 for a standard normal distribution
pnorm(0.5, mean = 0, sd = 1, lower.tail = FALSE)
# Equivalently:
1 - pnorm(0.5, mean = 0, sd = 1)
```

```r
# The probability of a value less than 0.5 and greater than -0.25 for a
# standard normal distribution
pnorm(0.5, mean = 0, sd = 1) - pnorm(0.25, mean = 0, sd = 1)
```

`qnrom()`: Calculate the values for a normal distribution for a given probability.

- *Option*: `mean`. The mean of the normal distribution
- *Option*: `sd`. The standard deviation of the normal distribution

Examples:

```r
# The value for which 75% of the probability is less than for a standard
# normal distribution
qnorm(0.75, mean = 0, sd = 1)
```

```r
# The value for which 75% of the probability is less than for a mean 66,
# standard deviation 8 normal distribution (The mean and standard deviation
# of men's heights in the United States)
qnorm(0.75, mean = 66, sd = 8)
# [1] 71.39592
# This is interpreted as: About 75% of men in the US are shorter than 71.4 inches.
```

Symbols and operators

`<-` : Assigns values on the right-side of the arrow to the name on the left-side of the arrow.

Example:

```r
new_object <- c("A", "A", "A", "B", "B")
tally(new_object)
```

Relationship operators

The following operators can be used to make comparisons:

- `==` : Are the left and right-sides exactly equal?
- `!=` : Are the left and right-sides not equal?
- `<=` : Is the left-side less than or equal to the right-side?
- `<` : Is the left-side less than the right-side?
- `>=` : Is the left-side greater or equal to the right-side?
- `>` : Is the left-side greater than the right-side?

Creating functions

`function(input_a, input_b)`: Create a function which takes inputs and computes outputs.

Examples:

```r
# Create a function, called mm_diff, that computes the difference between the
# median and mean values of a variable
# The inputs in this function are variable and data
```

```r
mm_diff <- function(variable, data) {
  # The steps taken to compute the output of the function must be between the
  # curly braces, {}.
  mean_val <- mean(variable, data = data)
  med_val <- median(variable, data = data)
  abs(mean_val - med_val)
}
```

```r
# The user will then populate the inputs. The following input methods are equivalent.
mm_diff(variable = ~height, data = cdc)
mm_diff(~height, data = cdc)
mm_diff(~height, cdc)
```