

Introduction to R

Matthias Haber























01 March 2017

What is R?

R is an open source programming language with a particular focus on statistical programming

- Originally developed as 'S' by Bell Labs in 1993

R in comparison

Language Rank	Types	Spectrum Ranking
1. C	  	100.0
2. Java	  	98.1
3. Python	 	98.0
4. C++	  	95.9
5. R		87.9
6. C#	  	86.7
7. PHP		82.8
8. JavaScript	 	82.2
9. Ruby	 	74.5
10. Go	 	71.9

(source: IEEE Spectrum)

R interface (console-only)

- Command line, unlike SPSS/Stata
- An interpreted programming language
- Purist's interface: Text-editor & copy paste

RStudio is an Integrated Developer Environment (IDE) and serves as:

- Code editor
 - Code highlighting/completion, indentation, ...
 - Feed code from editor to R-console
- Project manager
- Workspace viewer
- Data browser
- Enhanced output viewer
- Help browser

RStudio windows at startup

- Source editor
- Console
- Workspace
- Multi-purpose-panel

Projects

- A working directory for each project
- Code: .r files
- Dataset/Workspace: .Rdata files

Using RStudio (II)

Basic Workflow

- Edit in code editor (.r-file)
- Paste to console
- Save Workspace/Datasets (.Rdata-file)
- Save code routinely (no auto-save!)

Shortcuts

- Run code from editor: Select line and ctrl+Enter
- Switch between source and console: ctrl+1, ctrl+2
- Clear console: ctrl+L
- 'Arrow up' gives you the last line of code in the console
- Press Alt+Shift+K to see all keyboard shortcuts

Fundamentals of the R language

- Use `#` to comment code (will not be run)
- Case-sensitivity: `data` vs `Data`
- Assigning objects: `<-` and `=`

Assign the number 5 to an object called number

```
number <- 5
```

```
number
```

```
## [1] 5
```

Assign the character string Hello World

```
string <- "Hello World"
```

```
string
```

```
## [1] "Hello World"
```


Functions

Functions perform operations on the input given and end in ()

e.g. find the class of number

```
class(number)
```

```
## [1] "numeric"
```

Operations on scalars

You can use R as a calculator:

```
2 + 3
```

```
2 - 3
```

```
2 * 3
```

```
2 / 3
```

Functions on scalars:

```
a <- 5
```

```
factorial(a)
```

```
## [1] 120
```

Special values in R

- NA: not available, missing
- NULL: does not exist, is undefined
- TRUE, T: logical true
- FALSE, F: logical false

Finding special values

Function	Meaning
<code>is.na</code>	Is the value NA
<code>is.null</code>	Is the value NULL
<code>isTRUE</code>	Is the value TRUE
<code>!isTRUE</code>	Is the value FALSE

```
absent <- NA  
is.na(absent)
```

```
## [1] TRUE
```

Operator	Meaning
<	less than
>	greater than
==	equal to
<=	less than or equal to
>=	greater than or equal to
!=	not equal to
a b	a or b
a & b	a and b

Naming objects

- Object names cannot have spaces
 - Use `CamelCase`, `name_underscore`, or `name.period`
- Avoid creating an object with the same name as a function (e.g. `c` and `t`) or special value (`NA`, `NULL`, `TRUE`, `FALSE`).
- Use descriptive object names
- Each object name must be unique in an environment.
 - Assigning something to an object name that is already in use will overwrite the object's previous contents.

R is object-oriented

Objects are R's nouns and include (not exhaustive):

- character strings
- numbers
- vectors of numbers or character strings
- matrices
- data frames
- lists

Vectors

A vector is a container of objects put together in an order.

Define a vector

```
a <- c(1,4,5)
```

```
b <- c(3,6,7)
```

Join multiple vectors

```
ab <- c(a,b)
```

Find vector length (number of its elements)

```
length(a)
```

Reference vector components

```
ab[4] # Index one element in vector
```

```
ab[4:6] # Index several elements in a vector
```

```
ab[ab==6] # Index with condition
```


Operations on vectors

Operation	Meaning
<code>sort(x)</code>	sort a vector
<code>sum(x)</code>	sum of vector elements
<code>mean(x)</code>	arithmetic mean
<code>median(x)</code>	median value
<code>var(x)</code>	variance
<code>sd(x)</code>	standard deviation
<code>factorial(x)</code>	factorial of a number

Task 1

Calculate the mean of the vector 1,99,3,4,5,6,8. What's the mean if you out the 'outlier'?

Task 1 (solution)

Calculate the mean of the vector 1,99,3,4,5,6,8. What's the mean if you out the 'outlier'?

```
# Defining vector using sequence between 3 and 6
```

```
a <- c(1,99,3:6,8)
```

```
mean(a)
```

```
## [1] 18
```

```
# Calculate the mean of a but exclude the highest number
```

```
mean(a[a!=max(a)])
```

```
## [1] 4.5
```

A Matrix is a square 2 dimensional container, i.e. vectors combined by row or column

- Must specify number of rows and columns

`matrix(x,nrow,ncol,byrow)`

- `x`: vector of length `nrow*ncol`
- `nrow`: number of rows
- `ncol`: number of columns
- `byrow`: TRUE or FALSE, specifies direction of input

Task 2

Assign a 6×10 matrix with $1, 2, 3, \dots, 60$ as the data.

Task 2 (solution)

Assign a 6×10 matrix with 1,2,3,...,60 as the data.

```
m <- matrix(1:60, nrow=6, ncol=10)
```

```
m
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    7   13   19   25   31   37   43   49   55
## [2,]    2    8   14   20   26   32   38   44   50   56
## [3,]    3    9   15   21   27   33   39   45   51   57
## [4,]    4   10   16   22   28   34   40   46   52   58
## [5,]    5   11   17   23   29   35   41   47   53   59
## [6,]    6   12   18   24   30   36   42   48   54   60
```

Referencing matrices

- Like vectors, you can reference matrices by elements
 - `a[i,j]` refers to the *i*th row, *j*th column element of object `a`
- Can also reference rows/columns, these are vectors
 - `a[i,]` is *i*th row, `a[,j]` is *j*th column

Task 3

Extract the 9th column of the matrix from the previous problem.
How can you find the 4th element in the 9th column?

Task 3 (solution)

Extract the 9th column of the matrix from the previous problem.
How can you find the 4th element in the 9th column?

```
m[,9]
```

```
## [1] 49 50 51 52 53 54
```

```
m[4,9]
```

```
## [1] 52
```

```
m[,9][4]
```

```
## [1] 52
```

Data frames

Data frames are a two-dimensional container of vectors with the same length. Each column (vector) can be of a different class and can be indexed with `[,]` or `$`. You can use functions like `nrow()`, `ncol()`, `dim()`, `colnames()`, or `rownames()` on your df.

```
# Combine two vectors into a data frame
```

```
number <- c(1, 2, 3, 4)
```

```
name <- c('John', 'Paul', 'George', 'Ringo')
```

```
df <- data.frame(number, name, stringsAsFactors = FALSE)
df
```

```
##   number  name
## 1      1  John
## 2      2  Paul
## 3      3 George
## 4      4  Ringo
```

Lists

A list is an object containing other objects that can have different lengths and classes.

```
# Create a list with three objects of different lengths
list1 <- list(beatles = c('John', 'Paul', 'George', 'Ringo'
                        alive = c('Paul', 'Ringo'), albums = 1:13)

list1

## $beatles
## [1] "John"    "Paul"    "George"  "Ringo"
##
## $alive
## [1] "Paul"    "Ringo"
##
## $albums
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13
```

R's build-in data sets

There are a number of example data sets available within R.

```
# List internal data sets:
```

```
data()
```

```
# Load swiss data set:
```

```
data(swiss)
```

```
# Find data description:
```

```
?swiss
```

Importing data

You can read data and assign it to an object. The most frequently used functions to read data include:

- `load()`: To open `.RData` files
- `read.csv()`: Reads csv file
- `read.table()`: Is more general and allows to set separators
- `read.dta()`: From `foreign` library, used to read Stata files

You can export your data in various formats:

- `save()`: Only readable in R, but can store multiple objects
- `write.csv()`: Writes matrix/dataframe to csv
- `write.table`: Writes matrix to a tab delimited text file
- `write.dta()`: Writing in Stata format, requires foreign library

For() loops

For() loops are used to loop around a vector/matrix to do something.

```
m <- matrix(1:5, nrow=1, ncol=5)
```

```
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
```

```
## [1,]    1    2    3    4    5
```

```
for (j in 1:3){
```

```
  m[,j]=0
```

```
}
```

```
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
```

```
## [1,]    0    0    0    4    5
```

For() loops (II)

You can also 'nest' a for() loop in another for() loop

```
m <- matrix(1:15, nrow=3, ncol=5)
for (i in 1:2){
  for (j in 1:4){
    m[i,j]=0
  }
}
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0   13
## [2,]    0    0    0    0   14
## [3,]    3    6    9   12   15
```


Task 4

Create a matrix with `matrix(1:20,nrow=4,ncol=5)` and another with `matrix(NA,nrow=4,ncol=5)`. Adapt the second to the first matrix using `for()`

- hint: define a 'counter' variable that increments by 1 each time you move to the next cell

Task 4 (solution)

Create a matrix with `matrix(1:20,nrow=4,ncol=5)` and another with `matrix(NA,nrow=4,ncol=5)`. Adapt the second to the first matrix using `for()`

```
counter=1
m1 <- matrix(1:20,nrow=4,ncol=5)
m2 <- matrix(NA,nrow=4,ncol=5)
for (j in 1:5){
  for (i in 1:4){
    m2[i,j]=counter
    counter=counter+1
  }
}
```

Task 4 (alternative solution)

Create a matrix with `matrix(1:20,nrow=4,ncol=5)` and another with `matrix(NA,nrow=4,ncol=5)`. Adapt the second to the first matrix using `for()`

```
m1 <- matrix(1:20,nrow=4,ncol=5)
m2 <- matrix(NA,nrow=4,ncol=5)
for (j in 1:5){
  for (i in 1:4){
    m2[i,j]=m1[i,j]
  }
}
```

If() statements

If() statements are used to make conditions on executing some code.
If condition is true, action is done.

```
a <- 3
b <- 4
number <- 0
if(a<b){
  number=number+1
}
number
```

```
## [1] 1
```

Tests for conditions: ==; >; <; >; <; !=

Task 5

Create the two objects `a <- sample(c(4,0),20,replace=TRUE)` and `m <- matrix(a,nrow=4,ncol=5)`. Now recode all the 4s into 1s using `if()` and `for()` statements.

Task 5 (solution)

Create the objects `a <- sample(c(4,0),20,replace=TRUE)` and `b <- matrix(a,nrow=4,ncol=5)`. Now recode all the 4s into 1s in `b` using `if()` and `for()` statements.

```
a <- sample(c(4,0),20,replace=TRUE)
b <- matrix(a,nrow=4,ncol=5)

for (j in 1:5){
  for (i in 1:4){
    if (b[i,j]==4){
      b[i,j]=1
    }
  }
}
```

Creating Functions

If you want to repeat an operation it is often useful to create your own Function. Functions have names, inputs and outputs and simplify your code.

Find the sample mean of a vector

```
fun_mean <- function(x){  
  sum(x) / length(x)  
}
```

```
## Find the mean
```

```
data(swiss)
```

```
fun_mean(x = swiss$Infant.Mortality)
```

```
## [1] 19.94255
```

Task 6

Write a function that takes a number and doubles it.

Task 6 (solution)

Write a function that takes a number and doubles it.

```
double <- function(x){  
  output <- x * 2  
  output  
}  
double(8)
```

```
## [1] 16
```

Apply function

Apply allows you to apply a function to every row or every column. This can be done with a `for()` loop, but `apply()` is usually much faster and simpler. `apply()` takes the following form: `apply(X, MARGIN, FUN, ...)`.

```
m <- matrix(c(1:10, 11:20), nrow = 10, ncol = 2)
# mean of the rows
apply(m, 1, mean)
```

```
## [1] 6 7 8 9 10 11 12 13 14 15
```

```
# mean of the columns
apply(m, 2, mean)
```

```
## [1] 5.5 15.5
```

Task 7

Load up the build-in R dataset 'iris' and use `apply()` to get the mean of the first 4 variables.

Task 7 (solution)

Load up the build-in R dataset 'iris' and use `apply()` to get the mean of the first 4 variables.

```
attach(iris)
apply(iris[,1:4], 2, mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##      5.843333      3.057333      3.758000      1.199333
```

Packages

You can greatly expand the number of functions by installing and loading user-created packages.

```
# Install dplyr package  
install.packages('dplyr')
```

```
# Load dplyr package  
library(dplyr)
```

You can also call a function directly from a specific package with the double colon operator (`::`).

```
Grouped <- dplyr::group_by(combined_df, character_vector)
```

Tidy data

Most of the time data sets have to be cleaned before you can run statistical analyses on them. To help streamline this process Hadley Wickham laid out principles of data tidying which links the physical structure of a data set to its meaning (semantics).

- In tidy data:
 - Each variable is placed in its own column
 - Each observation is placed in its own row
 - Each type of observational unit forms a table

Tidy data

Not tidy:

Person	treatmentA	treatmentB
John Smith		2
Jane Doe	16	11

Tidy:

Person	treatment	result
John Smith	a	
Jane Doe	a	16
John Smith	b	2
Jane Doe	b	11

Messy to tidy data

```
# Create messy data
messy <- data.frame(
  person = c("John Smith", "Jane Doe"),
  a = c(NA, 16), b = c(2, 11))
# Gather the data into tidy format
library(tidyr)
tidy <- gather(messy, treatment, result, a:b)
tidy
```

```
##      person treatment result
## 1 John Smith         a      NA
## 2  Jane Doe         a      16
## 3 John Smith         b        2
## 4  Jane Doe         b      11
```


Merging data

Once you have tidy data frames, you can merge them for analysis. Each observation must have a unique identifier to merge them on.

```
data(swiss)
swiss$ID <- rownames(swiss) # Create ID
df <- merge(swiss[,c(1:3,7)], swiss[,4:7], by = "ID")
```

Appending data

You can also add observations to a data frame.

```
data(swiss)
df <- swiss[1:3,1:3]
df2 <- rbind(df, swiss[4,1:3])
df2
```

##	Fertility	Agriculture	Examination
## Courtelary	80.2	17.0	15
## Delemont	83.1	45.1	6
## Franches-Mnt	92.5	39.7	5
## Moutier	85.8	36.5	12

Using dplyr

The dplyr package has powerful capabilities to manipulate data frames quickly.

```
library(dplyr)
```

```
##
```

```
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
```

```
##
```

```
##      filter, lag
```

```
## The following objects are masked from 'package:base':
```

```
##
```

```
##      intersect, setdiff, setequal, union
```

```
data(ames)
```

Piping

Piping allows to pass a value forward to a function call and produces faster compilation and enhanced code readability. In R use `%>%` from the `dplyr` package.

Not piped:

```
values <- rnorm(1000, mean = 10)
value_mean <- mean(values)
round(value_mean, digits = 2)
```

```
## [1] 9.99
```

Piped:

```
library(dplyr)
rnorm(1000, mean = 10) %>% mean() %>% round(digits = 2)
```

```
## [1] 9.94
```

There are a number of ways to generate tables in R. Two useful tools that are good to know are:

- `kable`: for creating tables from data frames
- `stargazer`: for creating tables of regression model output

In R, models are fit using the adequate functions e.g. `lm()` for OLS. Many models are packaged in one function, e.g. logistic regression is used with the `glm()` (generalized linear model) function by specifying the model type.

```
library(knitr)
kable(head(mtcars[,1:6]), digits = 2)
```

	mpg	cyl	disp	hp	drat	wt
Mazda RX4	21.0	6	160	110	3.90	2.62
Mazda RX4 Wag	21.0	6	160	110	3.90	2.88
Datsun 710	22.8	4	108	93	3.85	2.32
Hornet 4 Drive	21.4	6	258	110	3.08	3.21
Hornet Sportabout	18.7	8	360	175	3.15	3.44
Valiant	18.1	6	225	105	2.76	3.46

`kable` is limited if we want to create regression output tables, especially for multiple models. `stargazer` is good for this. `stargazer` can output tables in various formats.

To export a table to word document use: `type = 'html'`.

stargazer example

```
library(stargazer)
# Run regressions
output <- lm(rating ~ complaints + privileges + learning
              + raises + critical, data=attitude)
output2 <- lm(rating ~ complaints + privileges + learning,
              data=attitude)

# Create table
stargazer(output, output2, type="html",
           out="attitude.htm")
```


stargazer example

	<i>Dependent variable:</i>	
	rating	
	(1)	(2)
complaints	0.692*** (0.149)	0.682*** (0.129)
privileges	-0.104 (0.135)	-0.103 (0.129)
learning	0.249 (0.160)	0.238* (0.139)
raises	-0.033 (0.202)	
critical	0.015 (0.147)	
Constant	11.011 (11.704)	11.258 (7.318)
Observations	30	30
R ²	0.715	0.715
Adjusted R ²	0.656	0.682
Residual Std. Error	7.139 (df = 24)	6.863 (df = 26)
F Statistic	12.063*** (df = 5; 24)	21.743*** (df = 3; 26)
<i>Note:</i> *p<0.1; **p<0.05; ***p<0.01		

R has a powerful graphics engine to produce high quality graphs e.g.:

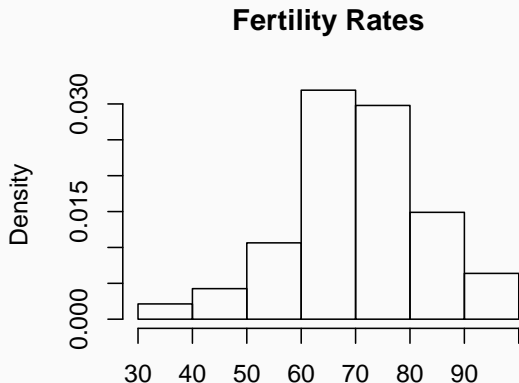
- `plot`: Basic plotting function (e.g. for scatterplots)
- `hist()`: Histograms
- `dotchart()`: Dot plots
- `boxplot()`: Box-and-whisker plots

Histograms

```
# Create a histogram
```

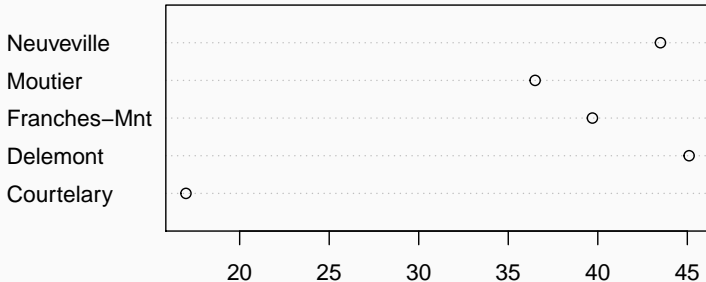
```
data(swiss)
```

```
hist(swiss$Fertility, freq=FALSE, main="Fertility Rates")
```



Dotchart

```
# Create a dot plot  
data(swiss)  
dotchart(swiss[1:5,2], labels=rownames(swiss))
```

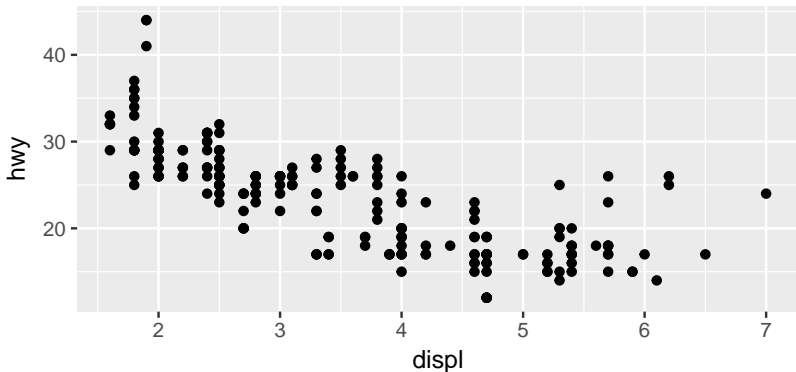


R has several systems for making graphs, but `ggplot2` is one of the most elegant and most versatile. `ggplot2` implements the grammar of graphics, a coherent system for describing and building graphs.

- Each plot is made of layers. Layers include the coordinate system (x-y), points, labels, etc.
- Each layer has aesthetics (`aes`) including x & y, size, shape, and color.
- The main layer types are called geometrics(`geom`) and include lines, points, etc.

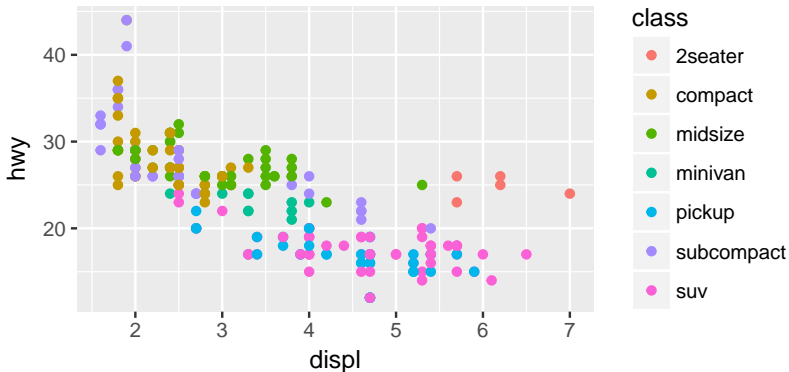
ggplot2 example

```
library(ggplot2)
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```



ggplot2 customization: color

```
library(ggplot2)
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class
```



ggplot2 customization: theme

```
library(ggplot2)
ggplot(data = mpg) + theme_bw() +
  geom_point(mapping = aes(x = displ, y = hwy, color = class
```

