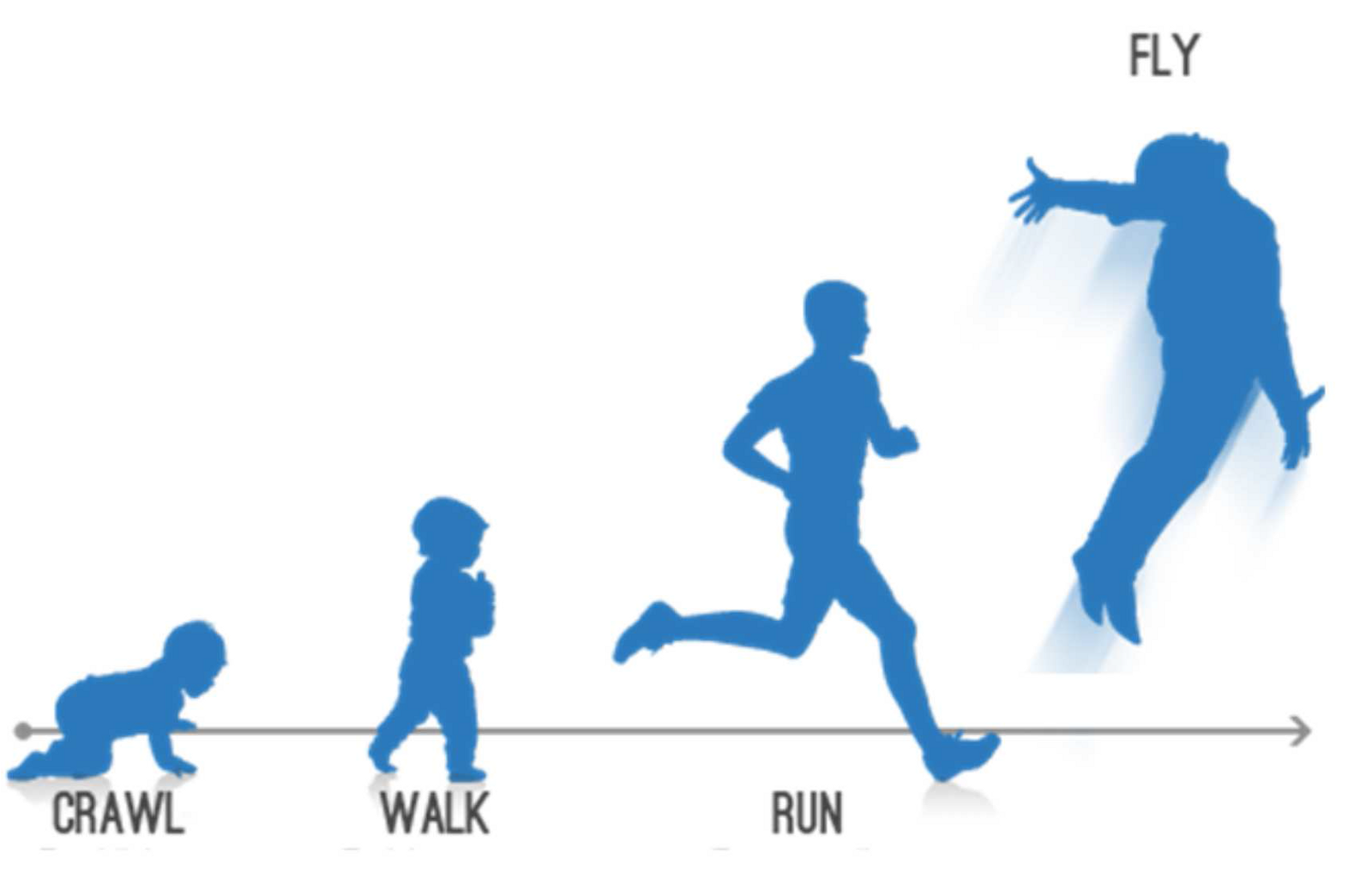
Session 1 - Introduction.Rmd

Table of Contents

# The purpose of this course

* An introduction to basic techniques in R 
* An interdisciplinary approach to R, e.g. regression modelling for psychologists, and text analysis for digital humanities

# Why R?

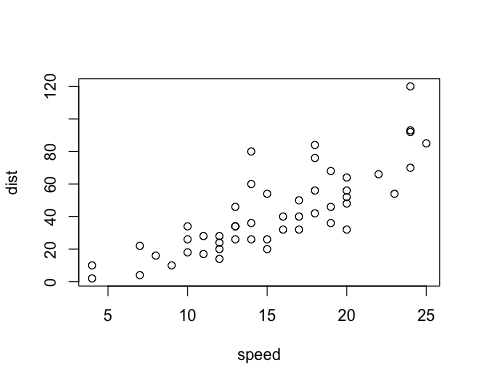
* *Open Source*
  + means that analyses are (a) cutting edge and (b) accurate
* *Strong emphasis on reproducible research*
  + data are (a) accurately reported (b) shareable

# How to use an R Markdown file

This is an [R Markdown](http://rmarkdown.rstudio.com) Notebook. When you execute code within the notebook, the results appear beneath the code.

Try executing this chunk by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Ctrl+Shift+Enter*.

plot(cars)



Add a new chunk by clicking the *Insert* button on the toolbar or by pressing *Ctrl+Alt+I*.

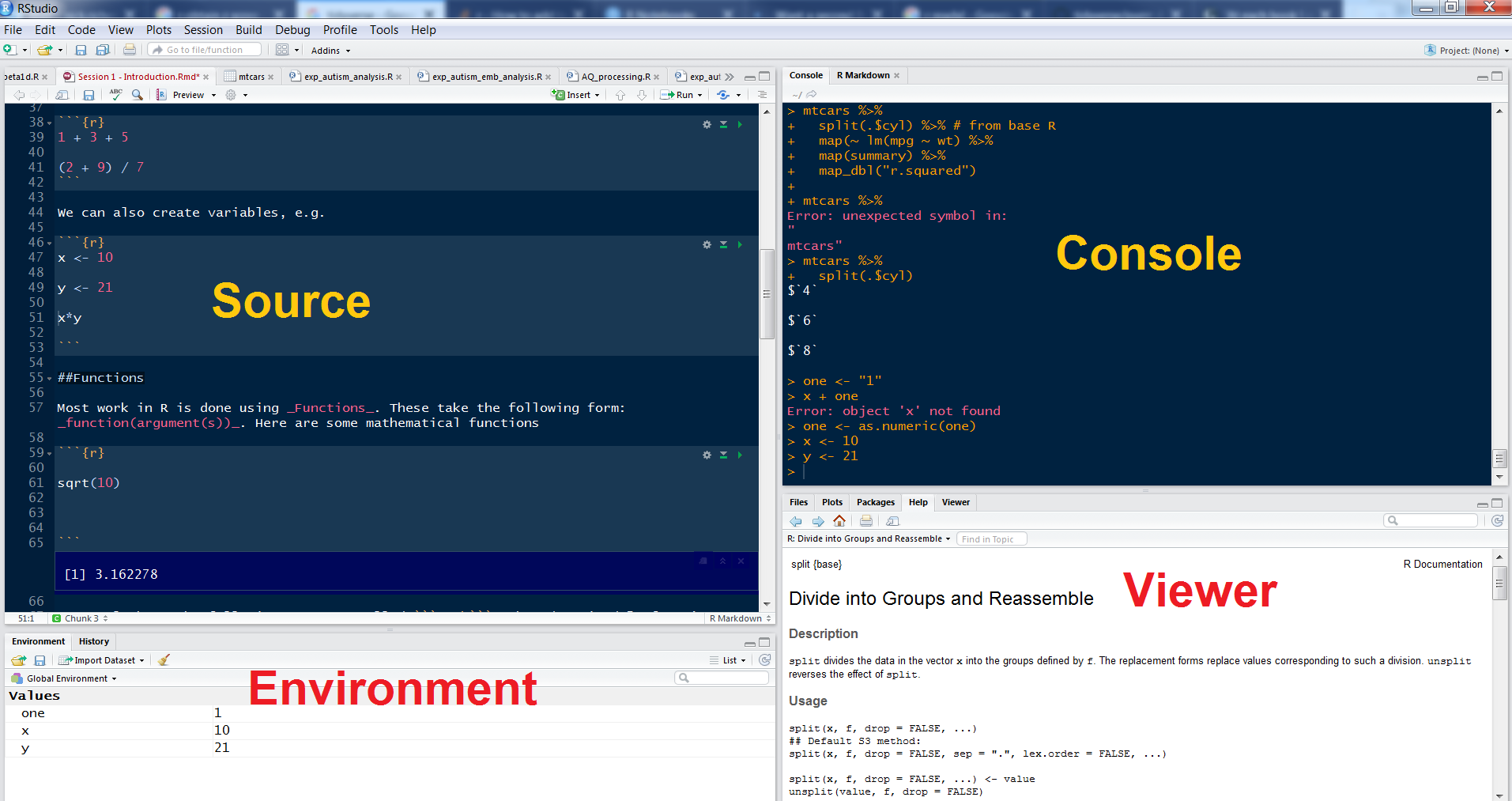
When you save the notebook, an HTML file containing the code and output will be saved alongside it (click the *Preview* button or press *Ctrl+Shift+K* to preview the HTML file).

# RStudio breakdown

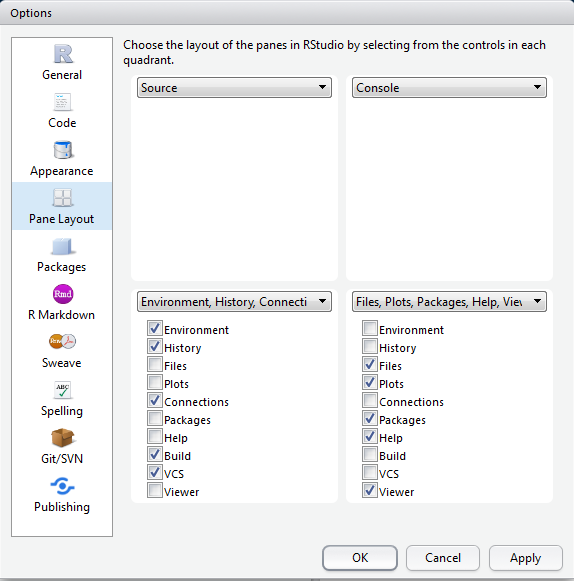
## Panes

RStudio shows you four panes:

1. The ‘Source’ pane: the file where you write your code
2. The ‘Console’ where actual code is run
3. The ‘Environment’ pane, which shows you variables / datasets
4. The ‘Viewer’ pane, which shows you plots and help files



You can arrange these in any order using Tools > Global Options.



## Autocomplete

RStudio has fantastic autocomplete capabilites. To autocomplete just press TAB. This is especially useful when loading files as using autocomplete will help you to identify relevant ones.

# R basics

## Setting the ‘working directory’

At the very beginning of an R session you MUST set a ‘working directory’. This tells R where to look for and save files. To do this type

setwd("path/to/directory")

Unfortunately, if you are on a windows machine you will need to change all backslashes \ to forward slashes /. This is because R follows UNIX conventions which are native to Linux and Mac computers.

If you are not sure what your working directory is type

getwd()

## Using R as a calculator

We can use the console for general arithmetic

1 + 3 + 5

## [1] 9

(2 + 9) / 7

## [1] 1.571429

We can also create variables, e.g.

x <- 10  
y <- 21  
x\*y

## [1] 210

## Comments

If you’d like to comment on any code you write (i.e. you do not wish R to try to ‘run’ this code) just add a hash (#) or series of hashes in front of it, e.g.

df <- read.csv("csv\_file.csv") # This reads in the main file for the experiment

## Functions

Most work in R is done using *Functions*. These take the following form: *function(argument(s))*. Here are some functions

sqrt(10)

## [1] 3.162278

seq(1,10,2)

## [1] 1 3 5 7 9

EX1: What do the arguments of seq do? To find out more search for the relevant help file in the console by typing ?seq

EX2: Have a look at the following arguments called gsub and grepl. What do they do? Clue: if you’re stuck, search the help file using ?

gsub("R-studio", "Rstudio", "R-studio is a great piece of software")

## [1] "Rstudio is a great piece of software"

grepl("chocolate", "Mary likes chocolate cookies")

## [1] TRUE

## DIY functions

It’s possible to **create your own functions**. This makes R extremely powerful and extendible. We’re not going to cover making your own functions in this course, but it’s important to be aware of this capability. There are plenty of good resources online for learning how to do this, including [this one](https://www.statmethods.net/management/userfunctions.html)

## Getting help

As we have seen above, to find out about a particular function just type ? and the name of the function into the console, e.g. ?grepl. This accesses the help files on your computer. If you’d like to search more broadly type ??grepl and your computer will look online for relevant materials on CRAN (the main R website)

Help files in R are quite densely written and not particularly aimed at beginners. Fortunately there are loads of excellent resources on the internet. Here are some really good sites:

1. <https://www.tidyverse.org/> - A brilliant set of of resources on all things related to the tidyverse, Hadley Wickham’s brilliant suite of packages
2. <https://www.statmethods.net/index.html> - a quick way of looking up basic R techniques
3. <https://stats.idre.ucla.edu/r/modules/>
4. <https://rseek.org/> - a search engine for all things related to R (because the word ‘R’ brings up a whole load of irrelevant stuff in Google)
5. <http://www.cookbook-r.com/> - this has lots of tips on how to do graphics.

# Packages

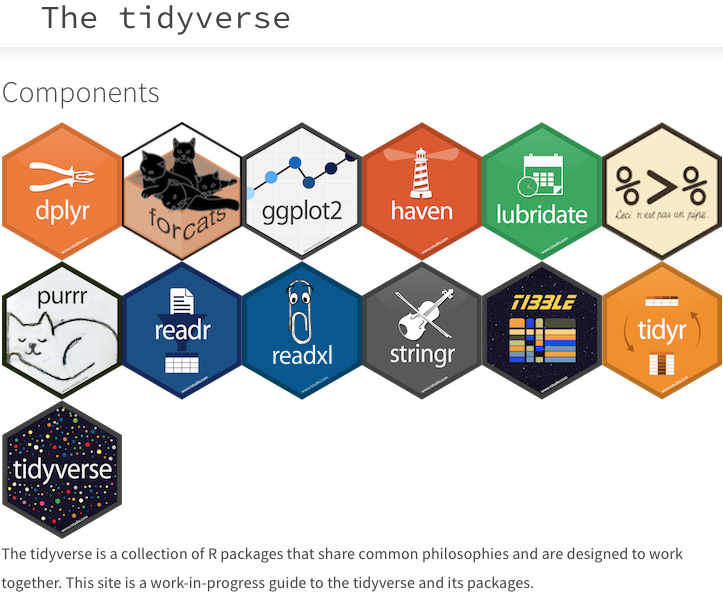
## Installation

To enhance the basic capabilities of R, we need to load packages/libraries. Most of the time, we download these from ‘CRAN’ Tools > Install packages or install.packages(). Once the package/library is installed (i.e. it is sitting somewhere on your computer), we then need to *load* it to the current R session using the library()function.

Remember using a package/library is a two-stage process. We

1. Install the package/library onto your computer (from the internet)
2. Load the package/library using the library command.

One of the most useful packages is called ‘tidyverse’.



It contains a number of useful commands for plots, and data manipulation.

Install the ‘tidyverse’ package, and then load it with the following function:

library(tidyverse)

## Registered S3 methods overwritten by 'ggplot2':  
## method from   
## [.quosures rlang  
## c.quosures rlang  
## print.quosures rlang

## ── Attaching packages ────────────── tidyverse 1.2.1 ──

## ✔ ggplot2 3.1.1 ✔ purrr 0.3.2  
## ✔ tibble 2.1.1 ✔ dplyr 0.8.1  
## ✔ tidyr 0.8.3 ✔ stringr 1.4.0  
## ✔ readr 1.3.1 ✔ forcats 0.4.0

## ── Conflicts ───────────────── tidyverse\_conflicts() ──  
## ✖ dplyr::filter() masks stats::filter()  
## ✖ dplyr::lag() masks stats::lag()

I find that a particularly easy way to load packages is via the pacman library. This contains the incredibly useful function p\_load. This

1. Checks if a particular package/library is installed and up-to-date
2. If it is not installed, or not up-to-date, it will download the latest version of the package/library
3. It will then load the package/library into the current session.

## Obtaining help

To find out more about a package type ?package\_name in the console. Alternatively you can look for the package documentation on [CRAN](https://cran.r-project.org/).

## Using functions from packages

Most of the functions loaded in a package should work ‘out of the box’. However occasionally you need to refer to the package first, and then the function using the format package\_name::function\_from\_that\_package. This is useful for a variety of reasons:

1. It allows you to use a function from a package without having to load that package
2. It helps in cases where you load two packages which contain two different functions which happen to have the same name.
3. Sometimes, even when a package is loaded, you need to precede a function by the package name. However, most of the time this is not necessary.

# Objects, data frames and indices

## Objects

A variable is a type of ‘object’ which R stores in memory. R is capable of creating and storing a wide range of objects. To see what type of object we have created, we use the function class(), e.g.

class(x)

## [1] "numeric"

z <- "hello"  
  
class(z)

## [1] "character"

*class* is one of the most useful functions in R as errors are often due to misassignment of class, e.g.

x + z

## Error in x + z: non-numeric argument to binary operator

Here we have tried to add a number to a string which is clearly impossible. It’s possible to change the class of an object using commands such as as.character, as.integer, as.numeric, as.factor, e.g.

one <- "1"  
x + one

## Error in x + one: non-numeric argument to binary operator

one <- as.numeric(one)  
x + one

## [1] 11

Here is a list of the main object classes in R:

1. Numeric - a number with decimal places
2. Integer - a number without decimal places
3. Character - a string of letters/numbers
4. Vector - an ordered list of numbers or characters, or multi-character strings. NB each number, character, or character string is also an object. So you can have objects within objects!
5. Dataframe - a 2 x 2 array in which each column has a name
6. List - this is like a vector, except it is capable of storing multiple object classes, e.g. it can contain both numbers and strings.

In order to create a vector we need to use the c function. (c = ‘combine’), e.g.

list.of.numbers <- c(1,4,54,22,43,9,0,0,21)  
  
mean(list.of.numbers)

## [1] 17.11111

sd(list.of.numbers)

## [1] 19.85223

a.character.vector <- c("Mary", "Jane", "Ali", "Chen")  
  
a.list <- as.list(c(1, 2, "Mary", "Jane"))

## Creating a data frame from scratch

A data frame is a two-dimensional object containing variables and row numbers. It’s basically a spreadsheet.

The following code creates a data frame programmatically. It creates two variables, and combines them together to make a data frame. Note that to do this we need to use the functions as.data.frame and cbind.

list.of.movies <- c("Independence Day", "Pretty Woman", "The Godfather Part  
Two", "Planet of the Apes (original)")  
  
rotten.tomatoes.variable <- c(62, 61, 97, 89)  
  
df <- as.data.frame(cbind(list.of.movies, rotten.tomatoes.variable)) # 'cbind' binds columns together

## Viewing the contents of a data frame

To glimpse the top few rows type head(name\_of\_data\_frame) in the console, e.g.

head(df)

## list.of.movies rotten.tomatoes.variable  
## 1 Independence Day 62  
## 2 Pretty Woman 61  
## 3 The Godfather Part\nTwo 97  
## 4 Planet of the Apes (original) 89

To view the data frame in the ‘source’ window, type View(name\_of\_data\_frame) in the console, .e.g.

View(df) #NB first letter is a capital letter.

## Referring to variables

To refer to variables, use the following syntax data\_frame\_name$variable\_name, e.g.

df$list.of.movies

## [1] Independence Day Pretty Woman   
## [3] The Godfather Part\nTwo Planet of the Apes (original)  
## 4 Levels: Independence Day Planet of the Apes (original) ... The Godfather Part\nTwo

When naming variables we can use dots and underscores, e.g. df$list.of.movies and df$list\_of\_movies. We can use numbers as long as they don’t come at the beginning, e.g. df$list\_of\_movies.v3.

If you use this convention, then the names for variables can get very long. However, it’s generally useful, as in R you often have multiple data frames loaded into memory. By specifiying both the name of the data frame and the variable, this avoids confusion.

Try to be consistent with your naming conventions. I tend to use underscores to name variables, e.g. data.frame.x$variable\_y. This is also what Hadley Wickham recommends (Have a look at the [Tidyverse Style Guide](https://style.tidyverse.org/))

If you’d like to see all the variable names in a data frame type names(data\_frame), e.g.

names(df)

## [1] "list.of.movies" "rotten.tomatoes.variable"

## Indices

Whenever you wish to access the contents of an object with multiple values (e.g. a data frame) you use indexes. These are placed inside square brackets, e.g. [1]. Have a look at the following example:

df[1,2]

## [1] 62  
## Levels: 61 62 89 97

df[1,] # here the second number is blank

## list.of.movies rotten.tomatoes.variable  
## 1 Independence Day 62

df[,2] # here the first number is blank

## [1] 62 61 97 89  
## Levels: 61 62 89 97

EX3: What does each number refer to? What happens when we leave a blank cell?

## Reading data frames from files using menus

We can use the menu in Rstudio: File > Import dataset. You can do this to import Excel, SPSS, SAS and STATA files.

## Reading data frames from files using code

However, rather than use the menu, it’s much better to use actual code, as this will automate the process. Let’s import a dataset on World Happiness Ratings, by country. The files are <WHR_2017.xlsx>, <WHR_2017.sav>, and <WHR_2017.csv>. Alternatively you can actually download the data set straight from the URL (below)

pacman::p\_load(tidyverse)  
pacman::p\_load(haven)  
  
df <- readxl::read\_excel("WHR\_2017.xlsx") # Read an excel file  
  
df <- haven::read\_spss("WHR\_2017.sav") # Read from an SPSS file  
  
df <- read.csv("WHR\_2017.csv") # Read from a .csv file  
  
#Or to download straight from the URL!!  
  
df <- read.csv("https://verbingnouns.github.io/AdventuresInR/docs/WHR\_2017.csv")

Possibly the best data format to work in is the .csv data format. This is good because it is readable in Excel, small, simple, and not easily-corrupted.

To read .csv files we use the read.csv() function from base R, e.g.

# Subsetting a data set using (a) base R and (d) dplyr

## Subsetting with base R

We’re going to *subset* the WHR dataset (i.e. choose only those cases/observations which fulfil a specific criterion). To do this we’re going to use the which() function. When you apply which to a variable in a dataset, it will produce indices of the rows which fulfil a certain criterio, e.g. which(df$var\_name == 2) will give you the indices of all rows where the value of the variable is 2.

EX4: Armed with this knowledge, your task is to subset the data frame so that it only contains information from African countries.

If you’re stuck have a look at the answer below.

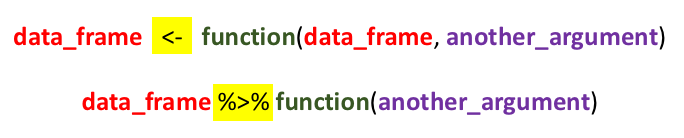
df.Africa <- df[which(df$region == "Africa"), ]

## Piping

Okay, the above code is pretty horrible to look at, so we’re going to explore an alternative using the package dplyr which is from the tidyverse. But before we can use dplyr we have to learn how to ‘pipe’.



Pipes are written in R as %>% (note you must use a percentage sign before and after the pipe). To demonstrate what pipes do, I have a look at the following pseudocode.



All pipes do is enable us to ‘pass’ a data frame (or another object) to a new function without having to keep on specifying the data frame. In addition, we can *chain* pipes together indefinitely.

Here’s how we would subset the data frame using piping:

df %>% filter(region == "Africa") -> df.Africa

Note that to create a new data frame, we need a solid arrow at the end. If we don’t include that solid arrow, the results are shown in the console, but no new data frame is created. This is an incredibly useful feature of pipes. You can try before you buy!

And here is an example where we *chain* a series of pipes together:

df %>%   
 group\_by(region) %>%  
 summarise(mean.happiness = mean(happiness\_score)) ->  
 df.mean.happiness.by.region

NB When piping the code becomes more readable when the line ends with the pipe.

There are a couple of important points to note.

1. We can refer to variables without specifying the data frame
2. If we wish to store the results we must output them using and arrow ->. If we don’t store the results they will merely be displayed in the console.

Piping is a key technique in R and once you’ve learnt it you will write much more powerful and readable code.

As well as using pipes to create data frame, you can also insert pipes into both analyses and figures! Here are some examples

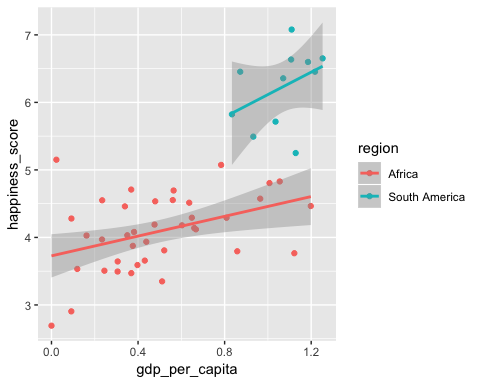
# An ANOVA without a pipe. NB we are using the base function "aov". If you would like to conduct SPSS-style ANOVAs, the best package is called "afex".  
  
mod <- aov(happiness\_rank ~ region, data = df)  
  
pacman::p\_load(broom) # To load the "tidy" function.  
  
tidy(mod)

## # A tibble: 2 x 6  
## term df sumsq meansq statistic p.value  
## <chr> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 region 9 178515. 19835. 21.8 5.06e-23  
## 2 Residuals 145 131795. 909. NA NA

# Here we use a pipe inside the analysis  
mod <- aov(happiness\_rank ~ region, # NB note we can break the line after a comma  
 data = df %>% filter(region == "Africa" | region == "South America"))  
  
tidy(mod)

## # A tibble: 2 x 6  
## term df sumsq meansq statistic p.value  
## <chr> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 region 1 67174. 67174. 189. 2.84e-18  
## 2 Residuals 48 17042. 355. NA NA

g <- ggplot(aes(x = gdp\_per\_capita, y = happiness\_score, colour = region), # NB note we can break the line after a comma  
 data = df %>% filter(region == "Africa" | region == "South America"))  
g <- g + geom\_point()  
g <- g + geom\_smooth(method = "lm")  
g



Note how I have broken some of the lines after a comma. This makes the code more readable. Generally we can break a line when it ends in some kind of symbol, e.g. a pipe, an arrow, or a comma.

# Loops and if-then statements

Loops and if-then statements are useful programming tools which have the same structure: FUNCTION (STATEMENT) {.....}.

## Loops

To demonstrate a loop we’re going to look at the WHR data set. We’re going to ask the question ’for different regions of the world, what is the relationship between GDP per capita nd happiness?

Here’s how we would do it

# This code drops regions where number of observations are less than 3 (we can't do correlations if there are less than 3 observations)  
df %>%  
 group\_by(region) %>%  
 summarise(num = n()) %>%  
 filter(num > 3) ->  
 df.region  
  
# Here is the code with the loop  
for (i in 1:length(df.region$region)){ # We loop through the list  
 df %>% filter(region == df.region$region[i]) -> temp.df # we subset the data according to the region. This contains a temporary dataset "temp.df"  
 model <- cor.test(temp.df$gdp\_per\_capita, temp.df$happiness\_score) # We do the analysis  
 print(paste("Region: ", df.region$region[i])) # We print the results  
 print(model)  
}

## [1] "Region: Africa"  
##   
## Pearson's product-moment correlation  
##   
## data: temp.df$gdp\_per\_capita and temp.df$happiness\_score  
## t = 2.6915, df = 37, p-value = 0.01062  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.1021586 0.6386187  
## sample estimates:  
## cor   
## 0.4046332   
##   
## [1] "Region: Central America"  
##   
## Pearson's product-moment correlation  
##   
## data: temp.df$gdp\_per\_capita and temp.df$happiness\_score  
## t = 3.9995, df = 10, p-value = 0.002521  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.3828945 0.9366585  
## sample estimates:  
## cor   
## 0.7844238   
##   
## [1] "Region: Central Asia"  
##   
## Pearson's product-moment correlation  
##   
## data: temp.df$gdp\_per\_capita and temp.df$happiness\_score  
## t = 2.6962, df = 12, p-value = 0.01944  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.1240646 0.8634151  
## sample estimates:  
## cor   
## 0.6142128   
##   
## [1] "Region: Europe"  
##   
## Pearson's product-moment correlation  
##   
## data: temp.df$gdp\_per\_capita and temp.df$happiness\_score  
## t = 8.2978, df = 40, p-value = 3.134e-10  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.6480351 0.8852640  
## sample estimates:  
## cor   
## 0.7953214   
##   
## [1] "Region: Middle East"  
##   
## Pearson's product-moment correlation  
##   
## data: temp.df$gdp\_per\_capita and temp.df$happiness\_score  
## t = 5.6137, df = 15, p-value = 4.938e-05  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.5666445 0.9341741  
## sample estimates:  
## cor   
## 0.8231111   
##   
## [1] "Region: South America"  
##   
## Pearson's product-moment correlation  
##   
## data: temp.df$gdp\_per\_capita and temp.df$happiness\_score  
## t = 1.3148, df = 9, p-value = 0.2211  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## -0.2614220 0.8069662  
## sample estimates:  
## cor   
## 0.4014009   
##   
## [1] "Region: South Asia"  
##   
## Pearson's product-moment correlation  
##   
## data: temp.df$gdp\_per\_capita and temp.df$happiness\_score  
## t = 0.48989, df = 4, p-value = 0.6499  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## -0.7109133 0.8796331  
## sample estimates:  
## cor   
## 0.2379102   
##   
## [1] "Region: South East Asia"  
##   
## Pearson's product-moment correlation  
##   
## data: temp.df$gdp\_per\_capita and temp.df$happiness\_score  
## t = 3.7425, df = 9, p-value = 0.004608  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.3390982 0.9401079  
## sample estimates:  
## cor   
## 0.7802562

EX5: The code below creates a sequence ranging from 0 to 30 going up in steps of 0.25. Try to achieve the same result using a loop

seq(0,30,2.5)

## [1] 0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0 22.5 25.0 27.5 30.0

## If-then statements

To demonstrate if-then statements, we are going to create a new variable which shows if the happiness index is above the mean

df$happiness\_above\_mean <- 0 # Set variable to 0  
mean\_happiness <- mean(df$happiness\_score) # Calculate mean mpg  
for (i in 1:nrow(df)){  
 if(df$happiness\_score[i] > mean\_happiness){df$happiness\_above\_mean[i] <- 1}  
}

And here is the same process using dplyr, which avoids the loop adn the if-then statement.

df %>%  
 transmute(happiness\_above\_mean = as.numeric(happiness\_score > mean(happiness\_score))) ->  
 df

Note loops and if-then statements are quite verbose, and there is almost always a neater and much shorter alternatives. However, I think they are useful procedures for the relative beginner.

Here is a much easier way to create the same variable

df <- read\_csv("WHR\_2017.csv")

## Parsed with column specification:  
## cols(  
## country = col\_character(),  
## region = col\_character(),  
## happiness\_rank = col\_double(),  
## happiness\_score = col\_double(),  
## whisker\_high = col\_double(),  
## whisker\_low = col\_double(),  
## gdp\_per\_capita = col\_double(),  
## family = col\_double(),  
## life\_expectancy = col\_double(),  
## freedom = col\_double(),  
## generosity = col\_double(),  
## trust\_in\_government = col\_double(),  
## dystopia\_residual = col\_double()  
## )

df$happiness\_above\_mean2 <- as.numeric(df$happiness\_score > mean(df$happiness\_score))

So how does this work? The statement in brackets evaluates to TRUE / FALSE. We then turn this into a number using as.numeric. TRUE evaluates to 1, while FALSE evaluates to 0.

It can be quite useful to chain statements. For example, if we wish to identify countries where both the happiness score and life expectancy are above the mean, we could do this….

df$happiness\_and\_LE\_above\_mean <- as.numeric((df$happiness\_score > mean(df$happiness\_score)) & (df$life\_expectancy > mean(df$life\_expectancy)))

EX6: Try to identify countries where both the GDP per capita and trust in the government are above the mean.

# Stored results

Whenever you run an analysis in R and save that to an object, the object has an internal structure. To demonstrate this, let’s do a simple regression using the mtcars dataset:

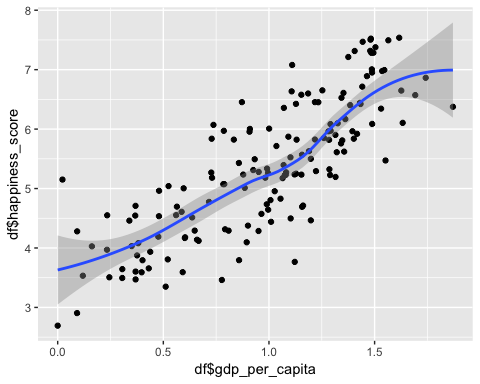
df <- read.csv("WHR\_2017.csv")  
  
head(df)

## country region happiness\_rank happiness\_score whisker\_high  
## 1 Norway Europe 1 7.537 7.594445  
## 2 Denmark Europe 2 7.522 7.581728  
## 3 Iceland Europe 3 7.504 7.622030  
## 4 Switzerland Europe 4 7.494 7.561772  
## 5 Finland Europe 5 7.469 7.527542  
## 6 Netherlands Europe 6 7.377 7.427426  
## whisker\_low gdp\_per\_capita family life\_expectancy freedom generosity  
## 1 7.479556 1.616463 1.533524 0.7966665 0.6354226 0.3620122  
## 2 7.462272 1.482383 1.551122 0.7925655 0.6260067 0.3552805  
## 3 7.385970 1.480633 1.610574 0.8335521 0.6271626 0.4755402  
## 4 7.426227 1.564980 1.516912 0.8581313 0.6200706 0.2905493  
## 5 7.410458 1.443572 1.540247 0.8091577 0.6179509 0.2454828  
## 6 7.326574 1.503945 1.428939 0.8106961 0.5853845 0.4704898  
## trust\_in\_government dystopia\_residual  
## 1 0.3159638 2.277027  
## 2 0.4007701 2.313707  
## 3 0.1535266 2.322715  
## 4 0.3670073 2.276716  
## 5 0.3826115 2.430182  
## 6 0.2826618 2.294804

Let’s draw a plot looking at the relationship between GDP per capita and Happiness Score. We’re not going to focus on the code, which will be covered in the next session.

g <- ggplot(aes(x = df$gdp\_per\_capita, y = df$happiness\_score), data = df)  
g <- g + geom\_point()  
g <- g + geom\_smooth()  
g

## `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



Now let’s run a regression

mod <- lm(happiness\_score ~ gdp\_per\_capita, data = df) # mod = "model"  
  
pacman::p\_load(broom) # Broom is a package which produces neat tables of results  
  
tidy(mod) # This is a broom function which tidies up the statistical results for reporting

## # A tibble: 2 x 5  
## term estimate std.error statistic p.value  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 (Intercept) 3.20 0.136 23.6 6.73e-53  
## 2 gdp\_per\_capita 2.18 0.127 17.2 1.11e-37

Now, let’s have a look at the structure of this model. There are two ways to do this:

1. Use the str function, e.g. str(mod)
2. Type mod$, and then use autocomplete.

We can see that the $ symbol has a dual function in R: firstly, to specify variables within dataframes, and secondly to specify subcomponents of an object.

It is useful to be able to refer to subcomponents of an object so that we can integrate into our report, e.g. the regression yielded a value of 0.6601055