A Gentle Introduction to R

Dr Burak Sonmez

UCL Social Research Institute

02/10/2023

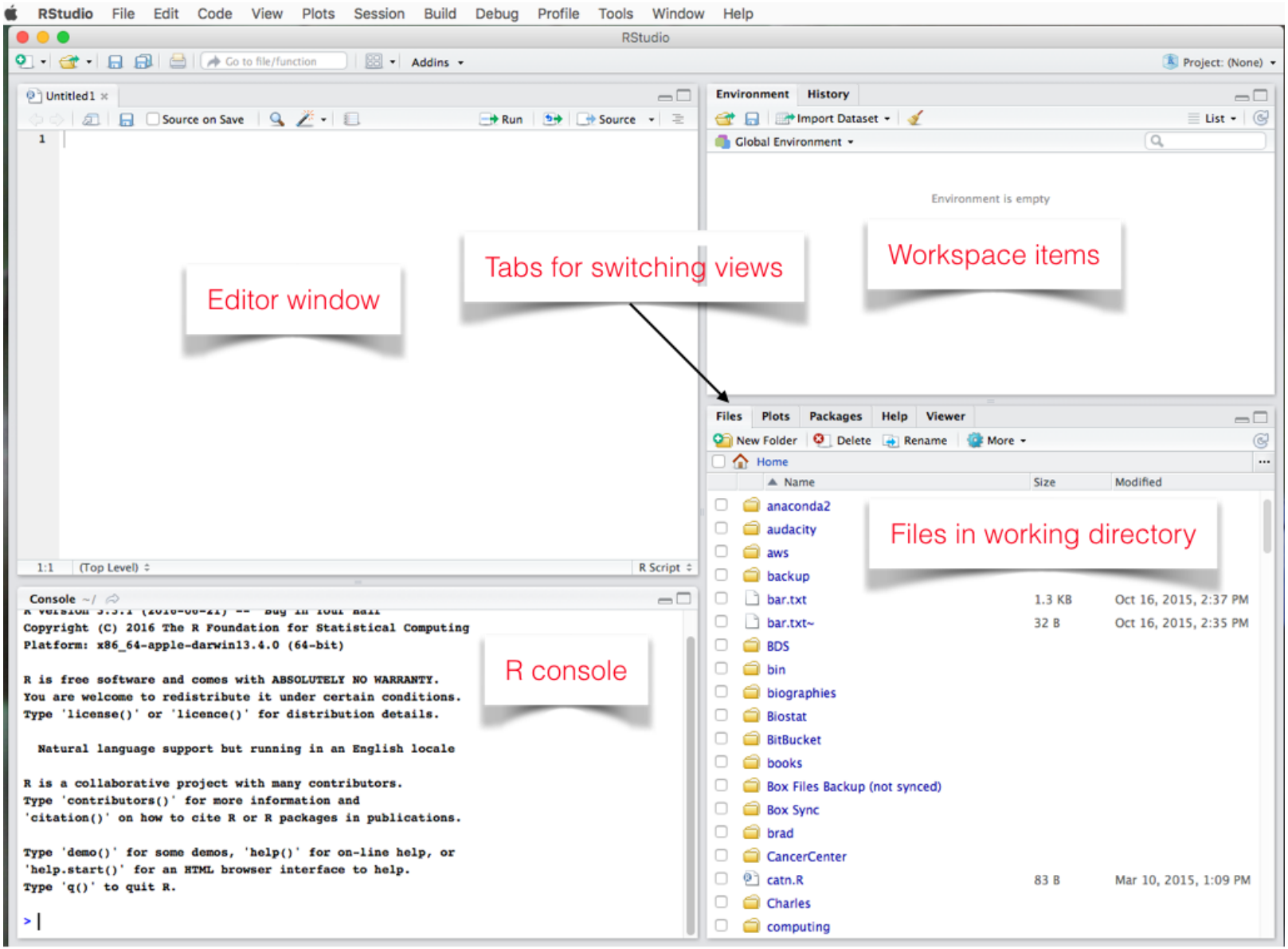
What is R?

- R is a widely known open-source software environment for statistical computing and graphics. Download: https://cran.r-project.org/ for both Windows and (Mac) OS X
- R is more than statistical software; it is essentially a programming language based on statistical programming language S (1976)
- Developed by **R**oss Ihaka & **R**obert Gentleman (1995)
- R may be a little bit challenging at the beginning because R demands precision, and carrying out simple tasks may seem to require a lot of effort
- Once you get acquainted with R, however, you can handle both basic and complex tasks with ease

Let's get started with R Studio

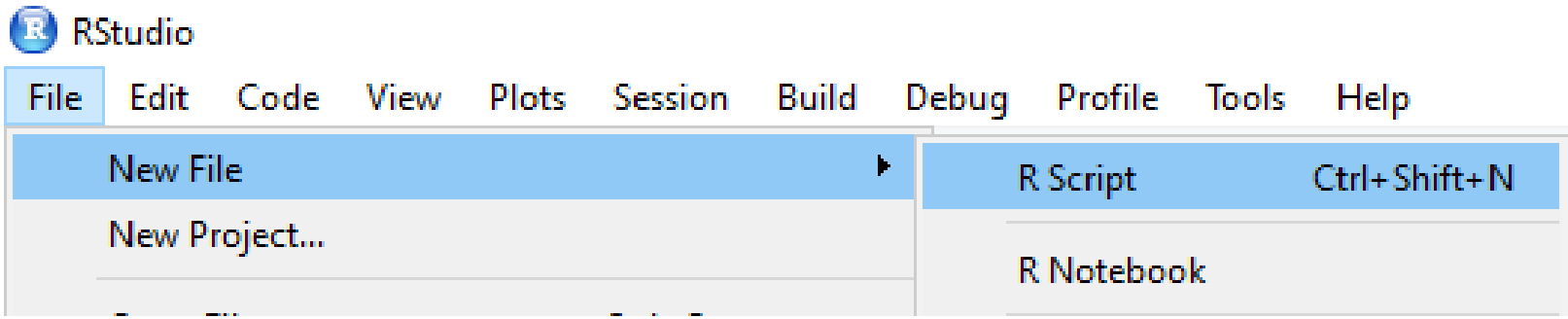
- We are going to use R Studio in this workshop. R Studio is free software that runs R in the background with some brilliant features
- It is an **i**ntegrated **d**evelopment **e**nvironment (IDE) for R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management
- Go to https://www.rstudio.com/products/rstudio/download/
- Select "RSTUDIO-2023.06.2-561.EXE - Windows 10/11 (64-bit)" for Windows users
- Select "RSTUDIO-2023.06.2-561.DMG - macOS 11+ (64-bit)" for Mac users

The RStudio interface

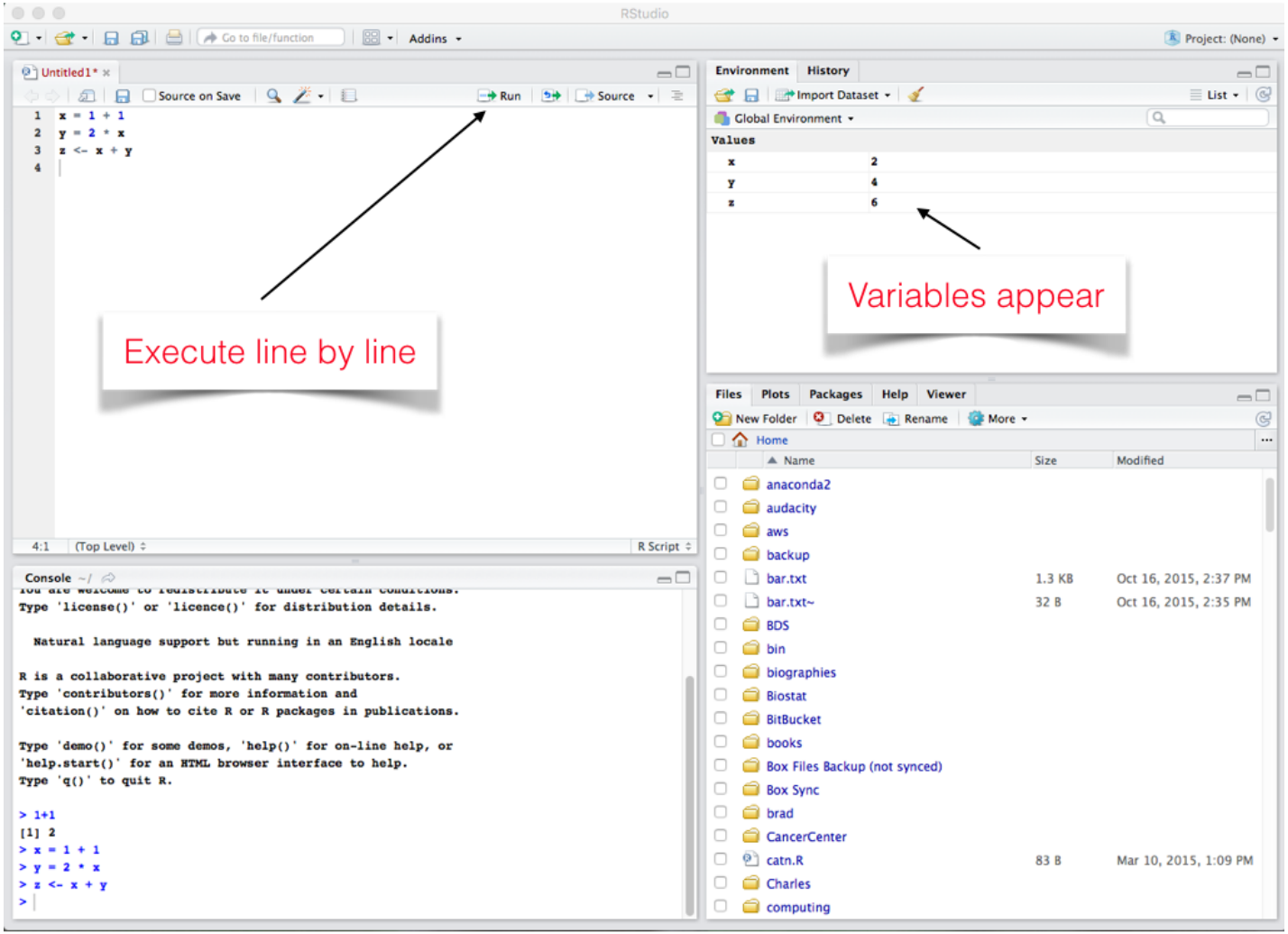


Working with R Script Files

- Rather than typing R commands into the console, we typically write short programs, known as "R scripts" that contain the R commands that we wish to execute
- A file editor tab will open in the source panel. R code can be entered here
- You can use File, then Save to give your script a name and save it in your working directory.

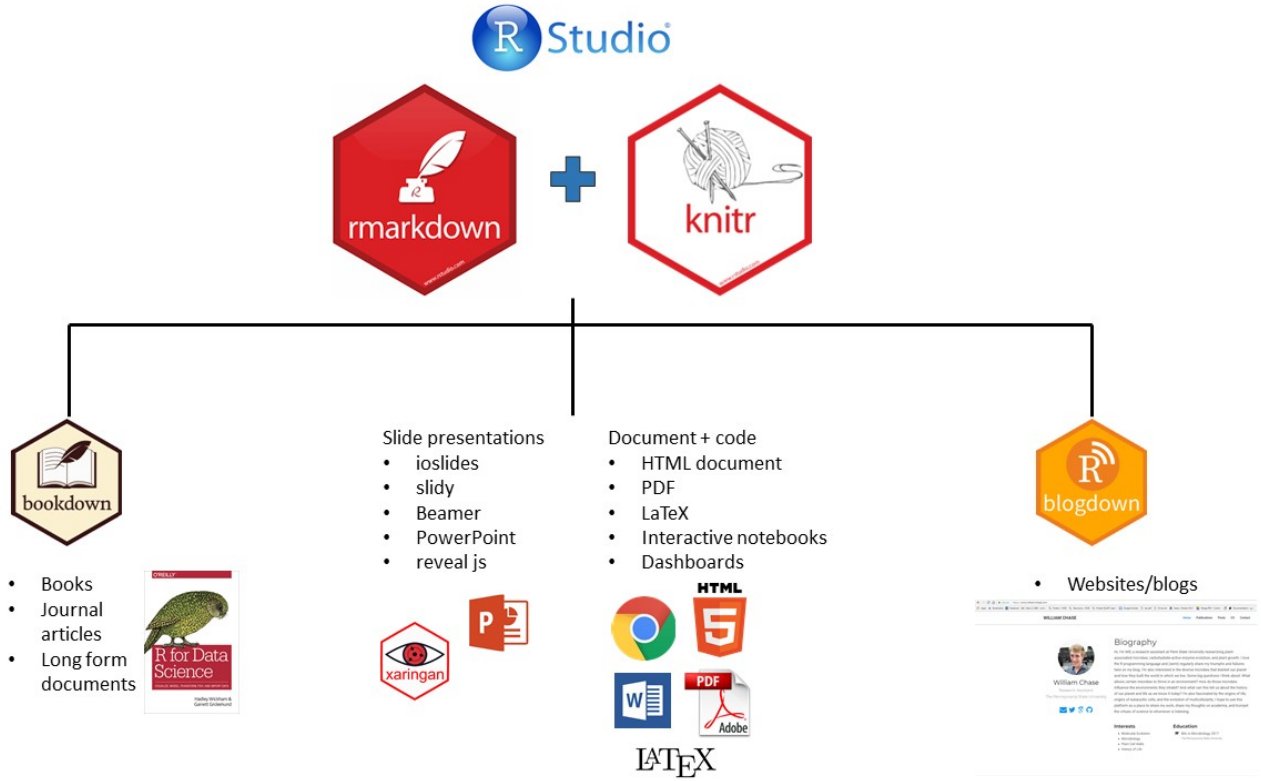


Running R code



- When running multiple lines: select all lines, then press 'Run' or cmd/ctrl+enter

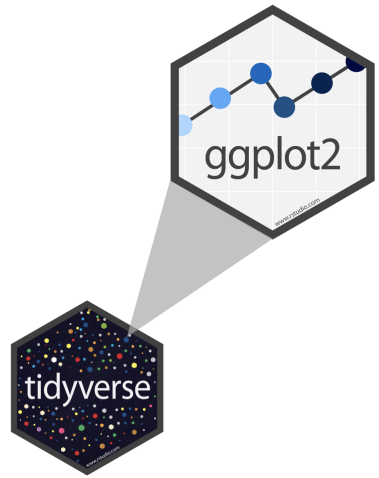
R Markdown/Quarto



- R Markdown/Quarto produces dynamic output formats in html, pdf, MS Word, dashboards, Beamer presentations, etc.
- We will be learning how to effectively use Quarto for reproducible documents in week 5

Base R vs R packages

- There are "default" packages that come with R. Some of these include:
  - mean(), median(), max(), min(), and sd()
  - ls() to see what we have in the environment
  - setwd() to set your working directory
  - print() to display the "things" on the console



- And there are R packages developed and shared by others. Some R packages include:
  - foreign
  - ggplot2
  - tidyverse

Installing and loading R packages

- You only need to install a package once on your computer. To install an R package use install.package() function.

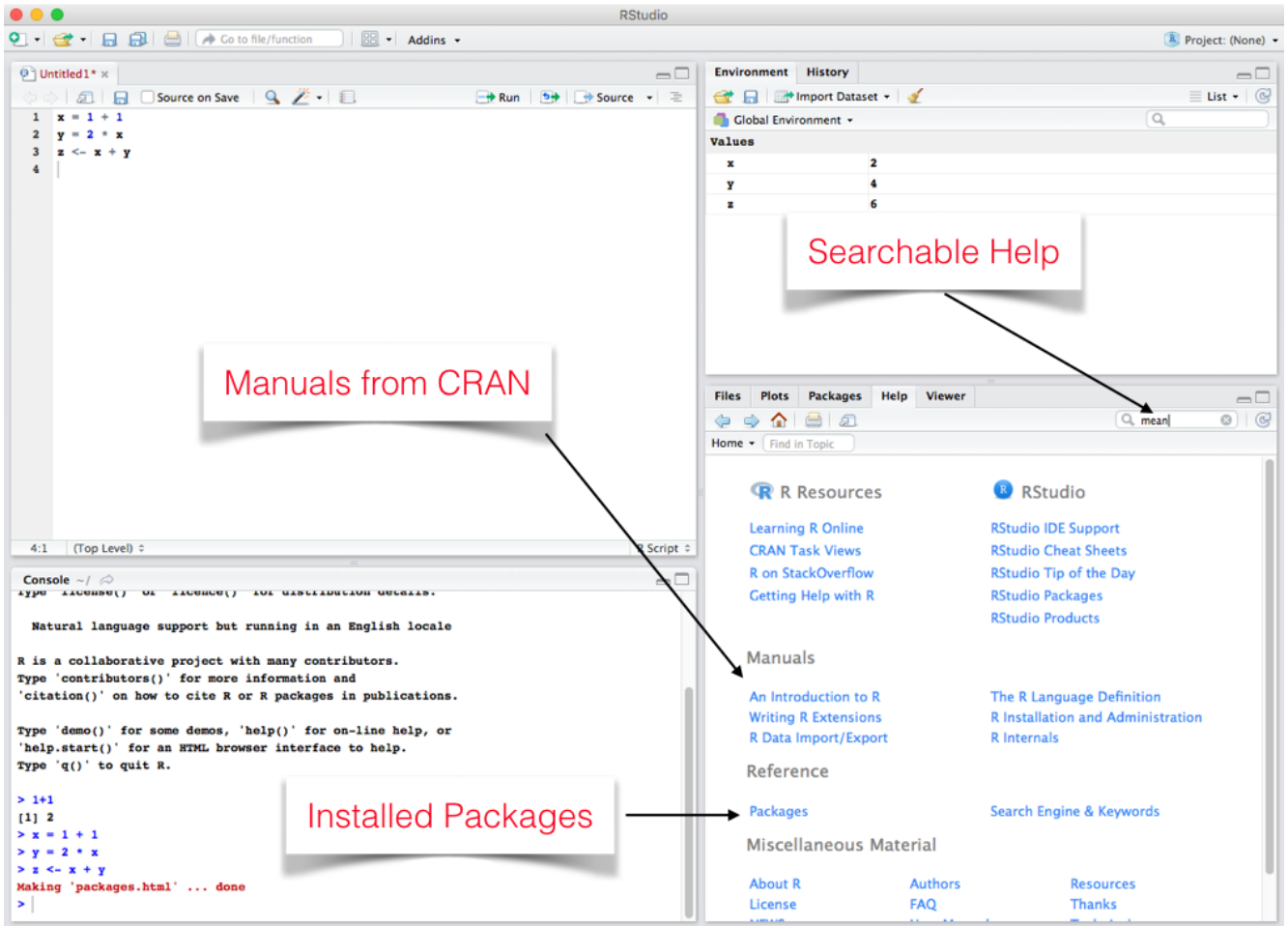
install.packages("tidyverse")

- However, you need to load a package everytime you plan to use it. To load a package use the library() function.

```
library(tidyverse)
```

It's easy to ask for HELP

- You can either use help() function to access R help files or use help section in the bottom right pane to search



Learning outcomes in this online workshop

Able to:

- Read and write lines of R code (even if you do not understand all functions, you know how to look them up)
- Understand what 'tidy' data is, how to generate it, and work with it
- Open, read, manipulate, analyse, visualise, and save a dataset, using some packages
- Use RStudio, and use it to write an R script and an R markdown document

R basics

- R as a calculator

```
5
# [1] 5
5+2
# [1] 7
10*3
# [1] 30
```

Assignment

Assignment means creating a variable or more generally, an "object" and assigning values to it

- <- is the assignment operator
- good practice to put a space before and after assignment operator!

```
# Create an object and assign value
a <- 10
a
# [1] 10
b <- "SOCS0100"
b
# [1] "SOCS0100"

print("Next slide please")
# [1] "Next slide please"
```

- The console has displayed "Next slide please". This is in quotation, which tells R that we are entering a text (string)

Objects

- Most statistical software, such as Stata, operates on datasets including rows of observations and columns of variables
- However, R is an "object-oriented" programming language like Python and JavaScript
- You can consider objects as anything you can assign values to (e.g. data, functions)
- Remember, you can also check what objects you have got in the environment by calling the ls() function

Objects

- Objects can be categorised by "type" and by "class"
- For instance, a date is an object with a numeric type and a date class
- There is no limit to the number of objects R can hold (except RAM memory)

Logicals

- A logical is `True` or `False`, and can also be written as `T` or `F`. Logicals are mostly used as follows:

==	is equal to
!=	is not
>=	larger than or equal to
<	smaller than

```
my.weight <- 65 #defining your weight as an object
my.weight == 65
# [1] TRUE
my.weight != 70
# [1] TRUE
my.weight <= 65
# [1] TRUE
my.weight > 70
# [1] FALSE
```

Vectors

- A vector is a collection of values
- The individual values within a vector are called "elements"
- Values in a vector can be numeric, character (e.g., "SOCS0100"), or some other type
- For instance, you can use the combine function c() to create a numeric vector that contains elements (e.g. your modules at UCL and your grades)

```
courses <- c("SOCS0100", "SOCS0079", "SOCS0081") #create object cal
#courses, which is a vector with three elements (characters)
courses # print object
# [1] "SOCS0100" "SOCS0079" "SOCS0081"
```

```
grades <- c(60, 63, 65) #create object called grades,
#which is a vector with three elements (numbers)
grades # print object
# [1] 60 63 65
```

Practicals (~5 min)

- Using either the R console or the R script file, please do the following exercises:

1. Create a vector called v1 with three elements, where all the elements are numbers. Then print the values.
2. Create a vector called v2 with four elements, where all the elements are characters. Then print the values.
3. Create a vector called v3 with five elements, where some elements are numeric and some elements are characters. Then print the values.

Solutions

```r
v1 <- c(50, 100, 150)

v1
# [1]  50 100 150
```

```r
v2 <- c("s", "o", "c", "i")

v2
# [1] "s" "o" "c" "i"
```

```r
v3 <- c("s", "o", 4, 9, 1)

v3
# [1] "s" "o" "4" "9" "1"
```

[1] The data in a vector must be only one type or mode (numeric, character, or logical) though. You can't mix modes in the same vector

Basic data structures

- There are two broad types of vectors (Grolemund & Wickham, 2016):

1. Atomic vectors: They are objects that contain elements. They are homogeneous. In other words, all elements within atomic vector must be of the same type. There are six types of atomic vectors: logical, **integer, double**, character, complex, and raw.
2. Lists: They are also objects that contain elements. Lists can be heterogeneous though. For example, one element can be an integer and another element can be character.

- These two concepts are not quite intuitive, but they will settle down after a while.

Length of an vector is the number of elements

- You can use `length()` function to examine vector length

```
x <- c(10, 14, 18)
x
# [1] 10 14 18
length(x)
# [1] 3

beatles <- c("Lennon", "McCartney",
             "Harrison","Starr")
beatles
# [1] "Lennon"    "McCartney" "Harrison"  "Starr"
length(beatles)
# [1] 4
```

It's straightforward to identify type of a vector

- You can use `typeof()` function to examine vector type

```
x
# [1] 10 14 18
typeof(x)
# [1] "double"

p <- c(0.5, 1.5)
p
# [1] 0.5 1.5
typeof(p)
# [1] "double"

beatles
# [1] "Lennon"    "McCartney" "Harrison"  "Starr"
typeof(beatles)
# [1] "character"
```

Sequences

- A sequence is a set of numbers in ascending or descending order (e.g., 1, 2, 3)
- It can be created using the colon operator `:` with the notation `start:end`
- You can use `seq()` function to create a series of numbers and assign it to an object.

```
s<- 5:10 #same as this: s<- c(5:10)
s
# [1]  5  6  7  8  9 10
length(s)
# [1] 6
```

```
seq(10,15)
# [1] 10 11 12 13 14 15
seq(from=10,to=15,by=1)
# [1] 10 11 12 13 14 15
seq(from=100,to=150,by=10)
# [1] 100 110 120 130 140 150
```

Vectors can be used in mathematical operations

```
p <- c(3:10)
p
# [1]  3  4  5  6  7  8  9 10
```

```
mean(p)
# [1] 6.5
```

```
p * 2
# [1]  6  8 10 12 14 16 18 20
```

```
c(2,1,1)+c(1,0,2)
# [1] 3 1 3
c(1,1,3)*c(1,0,2)
# [1] 1 0 6
```

Understanding structure of lists using str() function

```
l <- list(1,2,3)
typeof(l)
# [1] "list"
length(l)
# [1] 3
str(l)
# List of 3
#  $ : num 1
#  $ : num 2
#  $ : num 3
```

- Remember that each element of a list can be a vector of different length

```
l <- list(c(1,2),c(-1,0,5))
str(l)
# List of 2
#  $ : num [1:2] 1 2
#  $ : num [1:3] -1 0 5
```

Data types can differ across elements within a list

```r
b <- list(5,6,"beatles", TRUE)
typeof(b)
# [1] "list"
length(b)
# [1] 4
str(b)
# List of 4
#  $ : num 5
#  $ : num 6
#  $ : chr "beatles"
#  $ : logi TRUE
```

Lists can contain other lists

```
l1 <- list(c(5,6), list("beatles", "radiohead"), list(10, 20, 30))
str(l1)
# List of 3
#  $ : num [1:2] 5 6
#  $ :List of 2
#   ..$ : chr "beatles"
#   ..$ : chr "radiohead"
#  $ :List of 3
#   ..$ : num 10
#   ..$ : num 20
#   ..$ : num 30
```

You can also name each element in the list

```
l2 <- list(a=c(5,6), b=list("beatles", "radiohead"),
           c=list(10, 20))
str(l2)
# List of 3
#  $ a: num [1:2] 5 6
#  $ b:List of 2
#   ..$ : chr "beatles"
#   ..$ : chr "radiohead"
#  $ c:List of 2
#   ..$ : num 10
#   ..$ : num 20
```

- You can use `names()` function to show names of elements in the list

```
names(l2) # has names
# [1] "a" "b" "c"
names(l1) # no names
# NULL
```

Accessing individual elements in a list

-You can use the syntax: `list_name$element_name`

```
l2 <- list(a=c(5,6), b=list("beatles", "radiohead"),
           c=list(10, 20))
l2$a
# [1] 5 6
typeof(l2$a)
# [1] "double"
length(l2$a)
# [1] 2

typeof(l2$b)
# [1] "list"
length(l2$b)
# [1] 2
```

Combining the vectors to a unidimensional/multidimensional list with c()

- Let's say you have two vectors: candidate and age

```
candidate <- c("Biden", "Harris", "Trump", "Pence")
age <- c(78, 56, 74, 61)
mean(age)
# [1] 67.25
```

```
c(candidate,age)
# [1] "Biden"  "Harris" "Trump"  "Pence"  "78"     "56"     "74"
list(candidate,age)
# [[1]]
# [1] "Biden"  "Harris" "Trump"  "Pence"
#
# [[2]]
# [1] 78 56 74 61
```

Combine the vectors to a twodimensional data frame, with data.frame()

```
data.frame(candidate,age)
#   candidate age
# 1     Biden  78
# 2    Harris  56
# 3     Trump  74
# 4     Pence  61
df <- data.frame(candidate,age)
summary(df)
#   candidate              age
#  Length:4           Min.   :56.00
#  Class :character   1st Qu.:59.75
#  Mode  :character   Median :67.50
#                     Mean   :67.25
#                     3rd Qu.:75.00
#                     Max.   :78.00
```

Factors – a special type of vector, defined by *levels*

```
sex <- c("Male","Female","Male","Male")
sex
# [1] "Male"   "Female" "Male"   "Male"
factor(sex)
# [1] Male   Female Male   Male
# Levels: Female Male
```

```
df <- data.frame(candidate, age,
                 sex = factor(sex))
df
#   candidate age    sex
# 1     Biden  78   Male
# 2    Harris  56 Female
# 3     Trump  74   Male
# 4     Pence  61   Male
```

Dataframes in R (main takeaways)

- Data in R are held in objects of different types, dimensions and classes
- A data frame is just a list
- Each element in data frame must be a vector, not a list
- Each element (column) is a variable
- The length of an element is the number of observations (rows)
- Each element is also named
- You may have several different datasets with various types and shapes contained in the R environment

R built-in datasets

- You can use the `View()` function to display the data frame like a spreadsheet
- Please type this on the R Console `View(longley)`
- Remember that you can access individual columns of a data frame by using the dollar sign $

```
longley$Year
#  [1] 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1
# [16] 1962
longley$Population
#  [1] 107.608 108.632 109.773 110.929 112.075 113.270 115.094 116.2
# [10] 118.734 120.445 121.950 123.366 125.368 127.852 130.081
```

Accessing certain observations (rows) and/or certain columns (variables)

- You can use square brackets to subset data frames, in which the row coordinate goes first and the column coordinate second.

```
longley[5, ] # brings the 5th row
#      GNP.deflator     GNP Unemployed Armed.Forces Population Year
# 1951         96.2 328.975      209.9        309.9    112.075 1951
longley[,6] # brings the 6th column
#  [1] 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1
# [16] 1962
longley[1:2, ] # brings the first two rows
#      GNP.deflator     GNP Unemployed Armed.Forces Population Year
# 1947         83.0 234.289      235.6        159.0    107.608 1947
# 1948         88.5 259.426      232.5        145.6    108.632 1948
longley[1:2, c(5,6)] # brings the 5th & 6th column of 1st two rows
#      Population Year
# 1947    107.608 1947
# 1948    108.632 1948
```

Missing data

- Let's add a column (variable) to our data:

```
df$tax_return <- factor(c("Yes","Yes","No",NA))

df$tax_return
# [1] Yes  Yes  No   <NA>
# Levels: No Yes
```

- `na.rm` argument asks whether to remove NA values prior to calculation
- For most functions, default value is `na.rm = FALSE`
- If you specify, na.rm = TRUE , NA values removed prior to calculation

```
sum(c(1,2,3,NA))
# [1] NA
sum(c(1,2,3,NA), na.rm = TRUE)
# [1] 6
```

Missing data

- You must realise that `NA` is not a level!
- NA is a special keyword, not the same as the character string "NA"

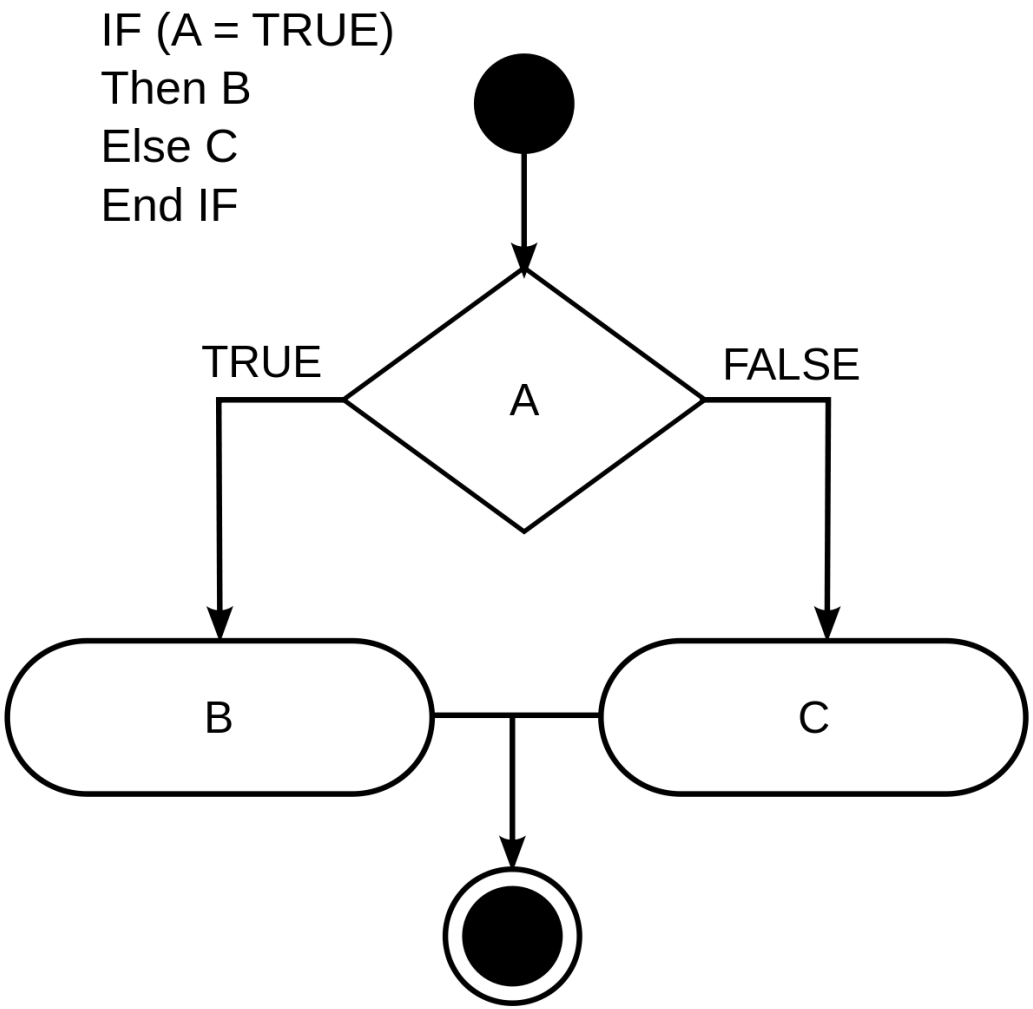
```
df
#   candidate age    sex tax_return
# 1     Biden  78   Male         Yes
# 2    Harris  56 Female         Yes
# 3     Trump  74   Male          No
# 4     Pence  61   Male        <NA>
```

- You can use `is.na()` function to determine if a value is missing

```
is.na(df)
#      candidate   age   sex tax_return
# [1,]     FALSE FALSE FALSE      FALSE
# [2,]     FALSE FALSE FALSE      FALSE
# [3,]     FALSE FALSE FALSE      FALSE
# [4,]     FALSE FALSE FALSE       TRUE
```

Programming: if statements

- A test/condition is this statement true or false?
- If the statement A is true, then do B; if false, then do C (optional)



Conditionals

- You can use `if(){}else{}` function

```
number <- 10
if(number > 9){
print("Bingo")
} else {
print("Nah")
}
# [1] "Bingo"
```

Programming: functions

- We previously use some predefined functions (e.g. `mean()`, `sum()`, `length()`)
- We can customize functions to serve our special needs
- Functions contain multiple instructions that create a cohesive unit:

```
name <- function(argument_1, argument_2, ...){
    commands
    return(value)
}
```

Programming: functions

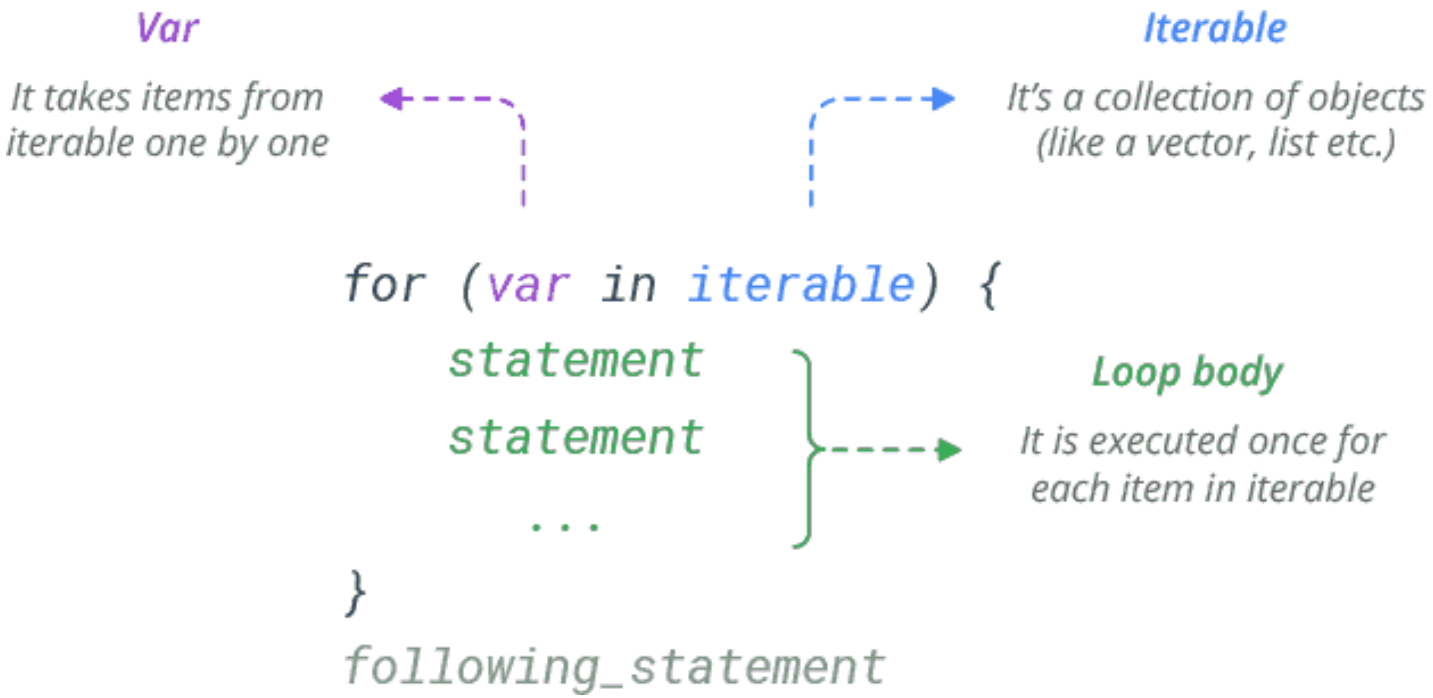
- Let's remember our created dataset
- Arguments, sometimes referred to as parameters, are special variables that are passed into functions, so that they can be used to perform some tasks

```
df
#   candidate age    sex tax_return
# 1     Biden  78   Male         Yes
# 2    Harris  56 Female         Yes
# 3     Trump  74   Male          No
# 4     Pence  61   Male        <NA>
```

```
find_biden_age <- function(data){
  biden_age <- data[data$candidate == "Biden", "age"]
  return(biden_age)
}
find_biden_age(df)
# [1] 78
```

Programming: loops

- Instructions needs to be applied multiple times
- Input is an iterable object (e.g. multiple similar elements)



Programming: loops

```
for (time in c(1:5)) {
print(c("The course will", "finish", "in", time , "minutes" ))
}
# [1] "The course will" "finish"          "in"              "1"
# [5] "minutes"
# [1] "The course will" "finish"          "in"              "2"
# [5] "minutes"
# [1] "The course will" "finish"          "in"              "3"
# [5] "minutes"
# [1] "The course will" "finish"          "in"              "4"
# [5] "minutes"
# [1] "The course will" "finish"          "in"              "5"
# [5] "minutes"
```

Importing data

- Data come in many different file formats such as .csv, .tab, .dta, .sav, etc. Today we will load a dataset which is stored in R's native file format: .RData
- rm() function in R is used to delete objects from the memory

```
rm(list = ls())
```

- Setting the working directory

```
setwd ("~/Desktop/R_Workshop")
```

- Importing data

```
load("gss2016.RData")
```

Importing data

- Inspecting the names of the variables and the dimensions of the dataset (dimension 1 = rows, dimension 2 = columns)

```
names(gss)
#  [1] "id"       "year"     "wtssall"  "vpsu"     "vstrat"   "polv
#  [7] "born"     "adults"   "hompop"   "race"     "region"   "age"
# [13] "sex"      "one"      "gunlaw"   "cappun"   "grass"    "eqwl
# [19] "marital"  "wrkstat"  "income16" "rincom16" "trust"    "soco
# [25] "socrel"   "socfrend" "relig"    "friend"   "degree"   "pres
# [31] "natsci"   "confinan" "conbus"   "conclerg" "coneduc"  "conp
# [37] "contv"    "conjudge" "consci"   "conlegis" "conarmy"  "spen
# [43] "sphlth"   "sppolice" "spschool" "sparms"   "sparts"
dim(gss)
# [1] 2867   47
```

Subsetting Data Frames with [] and $

- Show all obs where respondents' sex is male (1) (1276) and all columns (variables)

```
male <- gss[gss$sex == 1, ]
dim(male)
# [1] 1276   47
```

- Show all obs where respondents' sex is female (2) (1591) and the first three columns (first 3 variables)
- Show all obs where respondents' sex is "female" (2) and race is "black" (2) (283)

```
female <- gss[gss$sex == 2, 1:3]
dim(female)
# [1] 1591    3
black_female <- gss[gss$sex == 2 & gss$race == 2, ]
dim(black_female)
# [1] 283  47
```

Subsetting with Base R function

- The `subset()` is a base R function and easiest way to "filter" observations, which can be combined with `select()` another base R function to select variables

```
trust_inst <- subset(gss, select=c("confinan","conbus",
                                   "conclerg", "coneduc",
                "conpress", "contv", "conjudge", "consci"))
names(trust_inst)
# [1] "confinan" "conbus"   "conclerg" "coneduc"  "conpress" "contv
# [8] "consci"
```

Introducing tidyverse

- Tidyverse has become the most popular way of cleaning and manipulating data in R
- Tidyverse commands can be more efficient with less lines of code

tidyverse	base R	operation
`select()`	`[ ]`+ `c()` **OR** `subset()`	"extract" variables
`filter()`	`[ ]`+ `$` **OR** `subset()`	"extract" observations

```
library(tidyverse)
```

```
trust_inst <- select(gss, confinan, conbus, sex, race)
gss_race <- filter(gss, race == 1)
```

- You can also use %in% operator to further filter

```
gss_filter <- filter(gss, sex == 1, marital %in% 2:4)
```

Rename variables

- `rename()` function renames variables within a data frame object

```
rename(obj_name, new_name = old_name,...)
```

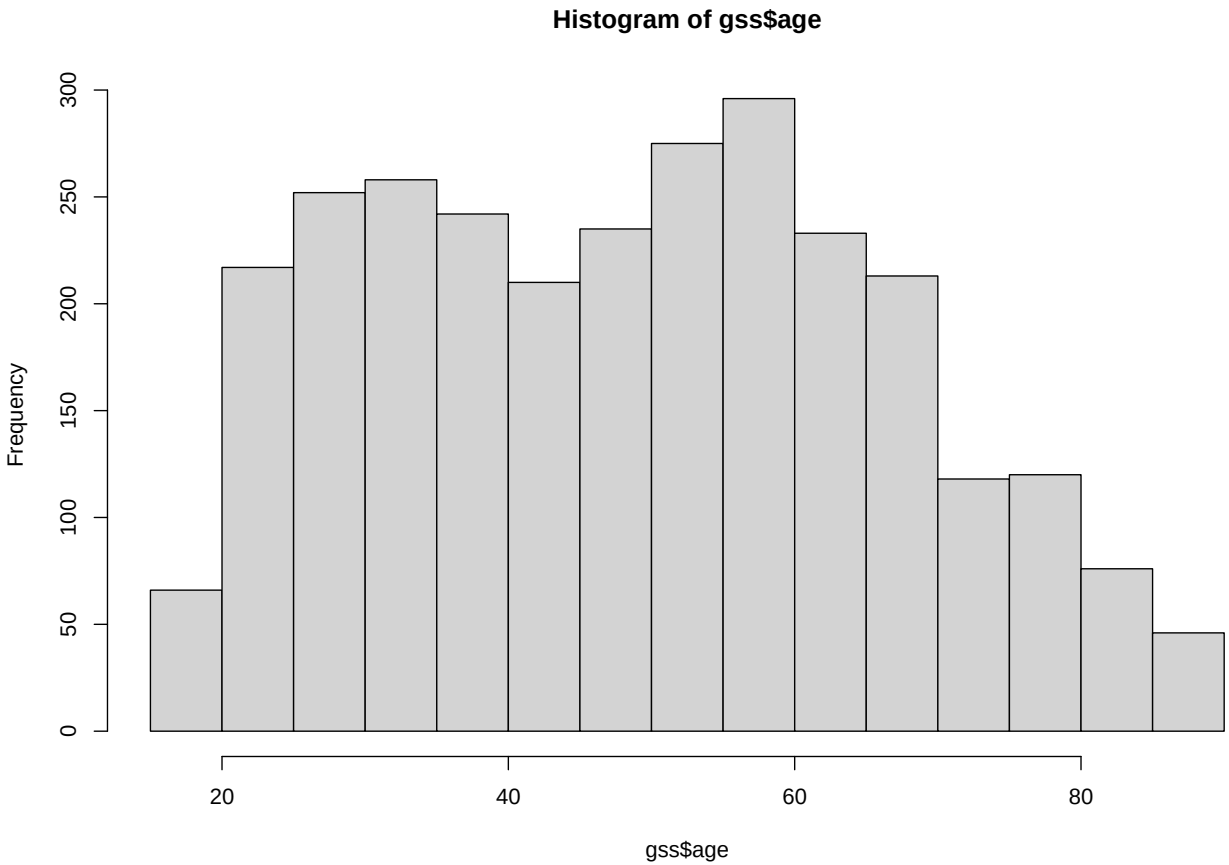
```
rename(gss, sexual_info = sex)
```

Creating new variables and renaming with `mutate()` and %>%

```
m <- gss %>% select(age, race, sex) %>% mutate(age_2 = age^2) %>%
  rename(ethnicity = race)
head(m, 3)
#       age ethnicity sex age_2
# 59600  47         1   1  2209
# 59601  61         1   1  3721
# 59602  72         1   1  5184
```

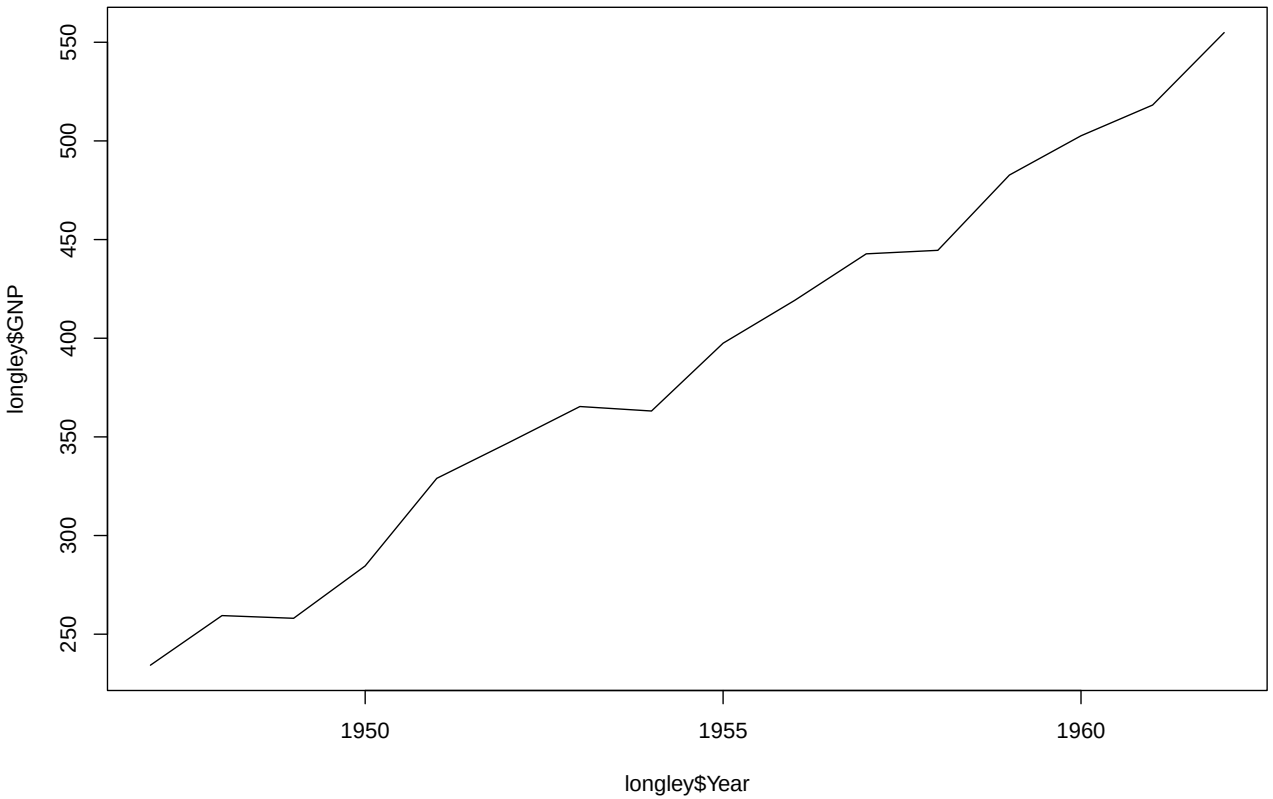
Simple plots

```
hist(gss$age)
```



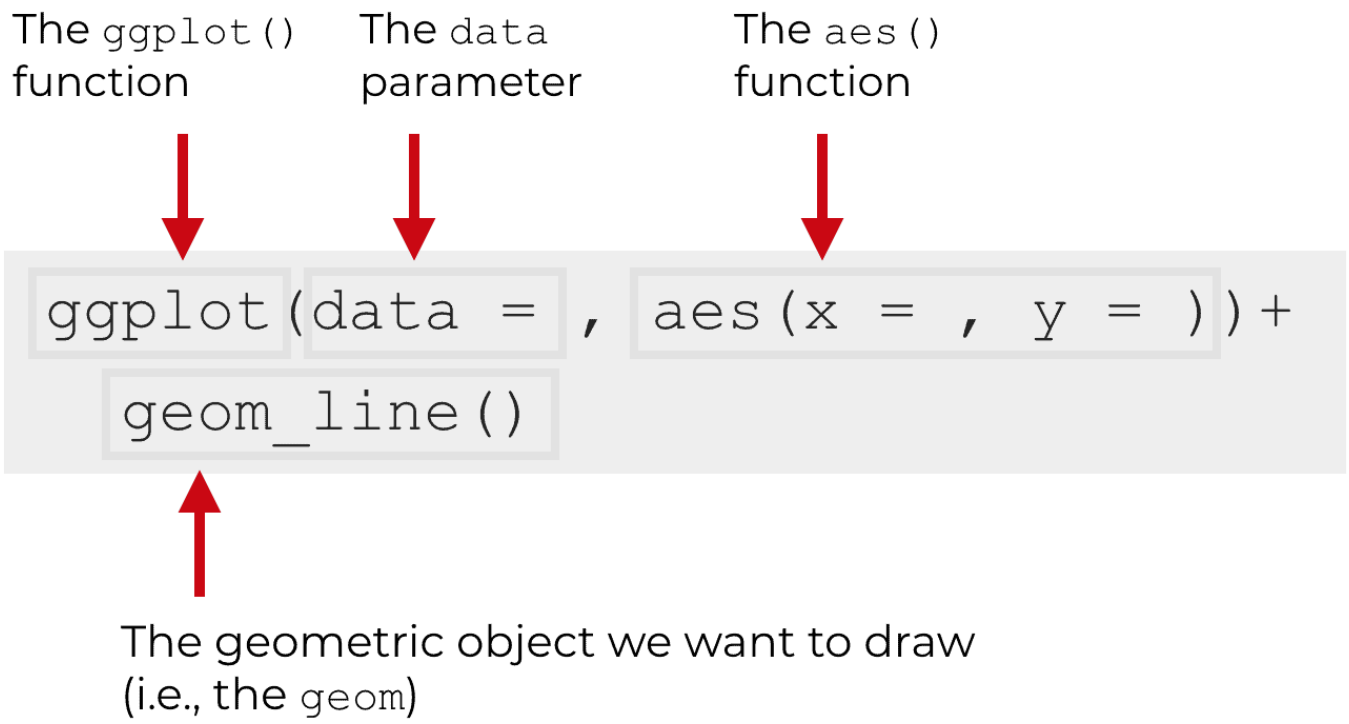
Simple plots

```
plot(longley$Year, longley$GNP, type="l")
```



Introducing ggplot2

- ggplot2 is to focus on data visualisation as part of the tidyverse
- ggplot2 visualises the data in a tidy dataframe. Thus, ggplot expects the input data to be in a dataframe
- There are four main parts of a basic ggplot2 visualisation: the `ggplot()` function, the data parameter, the `aes()` function, and the geom



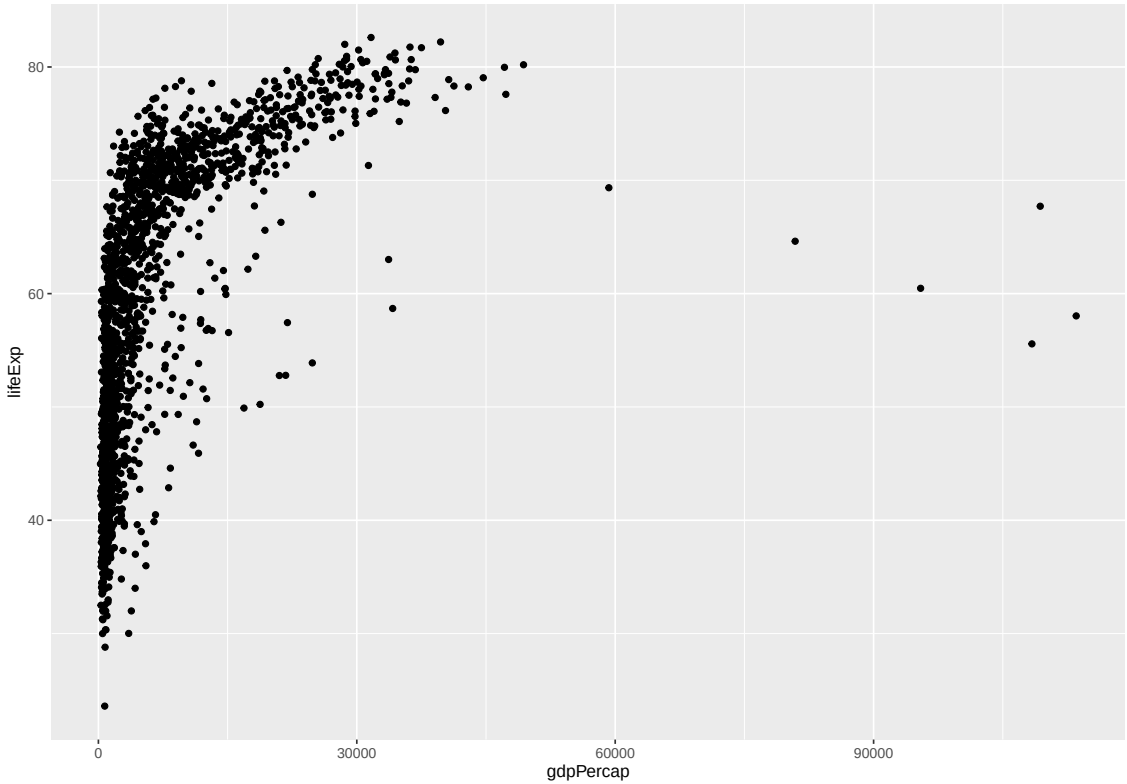
Introducing ggplot2

- You begin every plot by telling the `ggplot()` function what your data is
- When you provide an argument to the data parameter, it will always be a data.frame object of some type
- You will define how the variables in this data logically map onto the plot's aesthetics. Mappings are specified using the aes() function
- You can combine the argument that define the type of plot you want, which is called a geom. Each geom has a function that creates it
- For example, `geom_point()` makes scatterplots, `geom_bar()` makes barplots, `geom_boxplot()` makes boxplots

Plotting with ggplot2

```
library(gapminder)
```

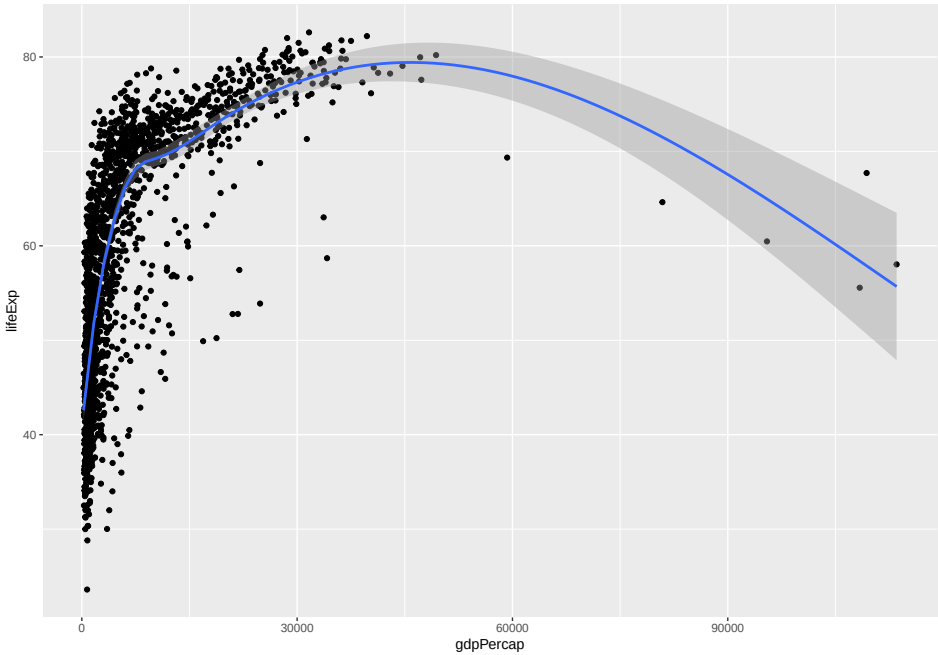
```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap, y = lifeExp))
p + geom_point()
```



Plotting with ggplot2

- You can build your plots layer by layer

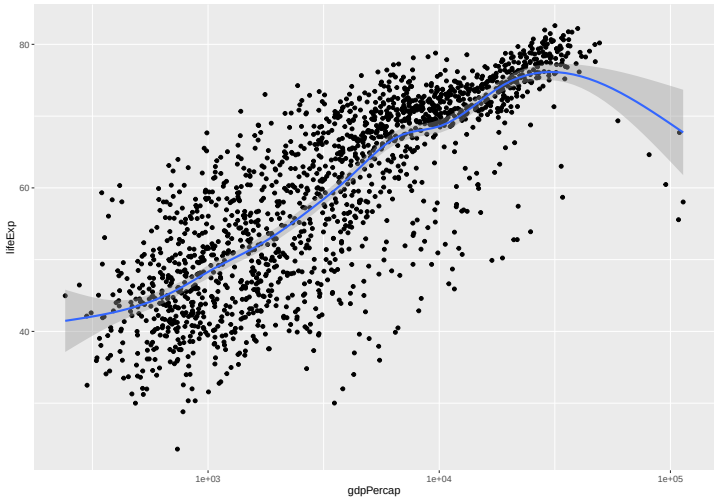
```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y=lifeExp))
p + geom_point() + geom_smooth()
# `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs =
```



Plotting with ggplot2

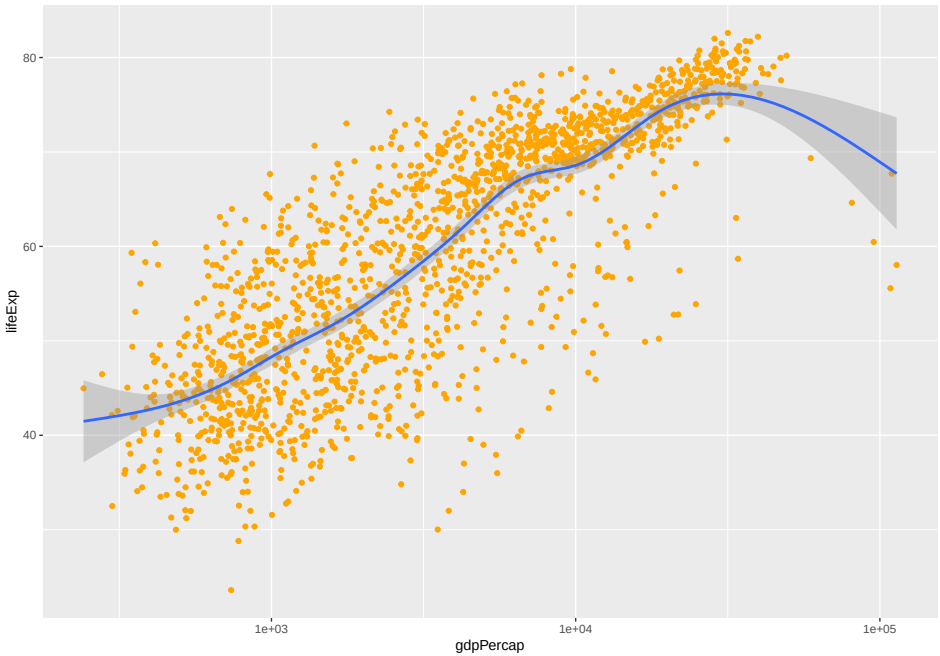
- Gross Domestic Product per capita is not normally distributed across the country years. The x-axis scale would probably look better if it were transformed from a linear scale to a log scale

```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y=lifeExp))
p + geom_point() + geom_smooth() +
    scale_x_log10()
# `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs =
```

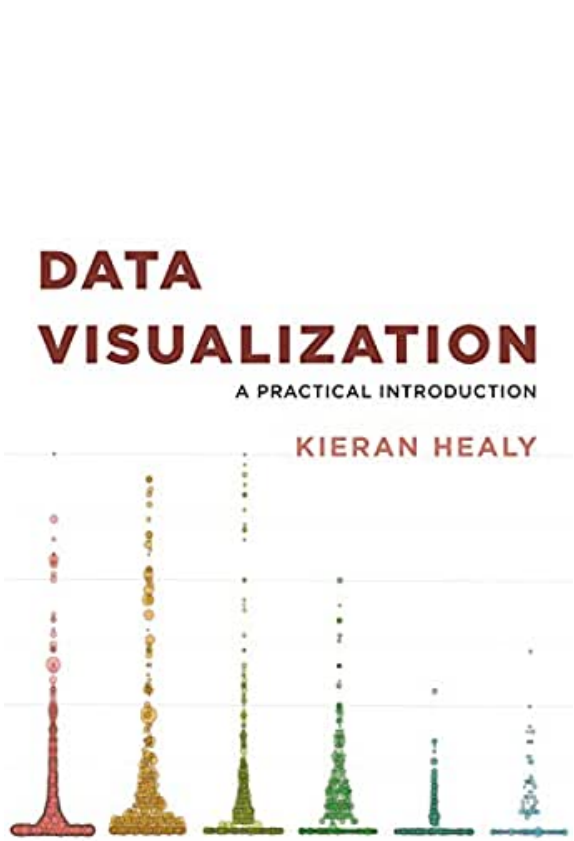


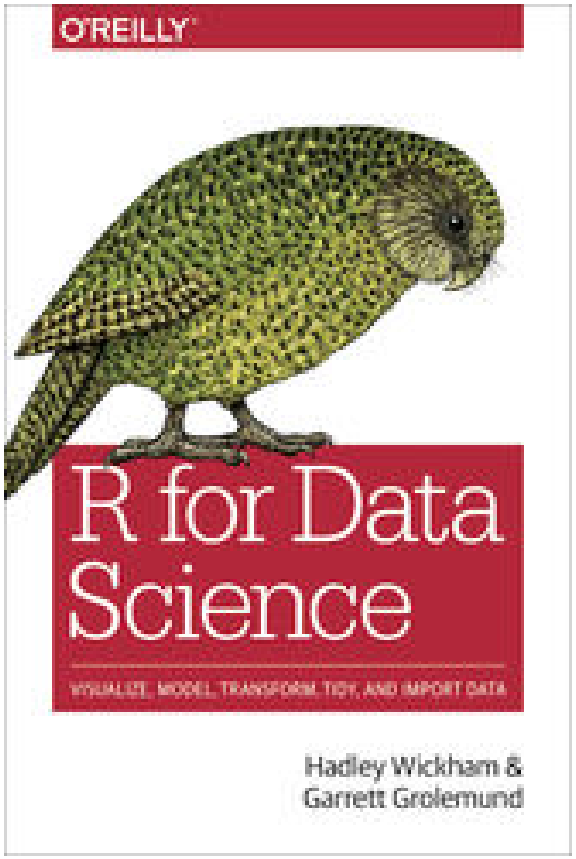
Plotting with ggplot2

```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y = lifeExp))
p + geom_point(color = "orange") +
    geom_smooth() +
    scale_x_log10()
# `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs =
```



Open sources for R





- https://community.rstudio.com/
- https://socviz.co/ (DataViz)
- https://r4ds.had.co.nz/ (R for Data Science)