DSA2101

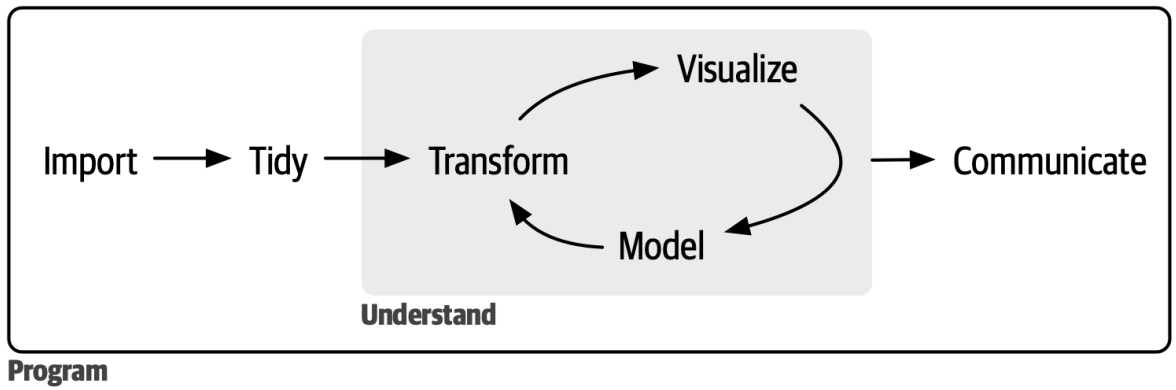
Essential Data Analytics Tools: Data Visualization

Yuting Huang

Week 1: Basics in R Programming I

Introduction

The goal for our module is to help you learn the most important tools in `R` that allow you to do data science.

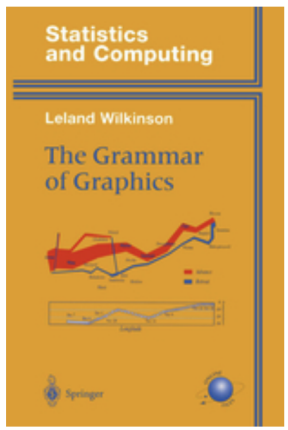


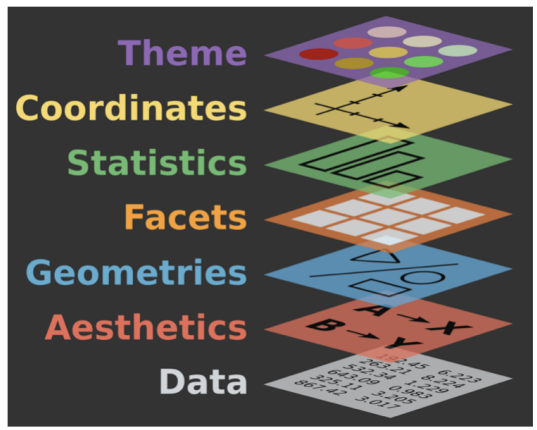
- The workflow is roughly organized according to the order in which you use them in a data analytics project.
- Although, of course, you will iterate through them multiple times.

Data visualization

There are many approaches in `R` for making visualizations: base `R`, `ggplot2`, ...

- `ggplot2` is `tidyverse`'s data visualization package.
- `gg` stands for *Grammar of Graphics*, a book by Leland Wilkinson.





- We will learn about it in the second half of the semester, after we lay out all the foundations.

Why do we visualize?

1. Discover patterns that may not be obvious from numerical summaries.

- The **Anscombe's Quartet**.

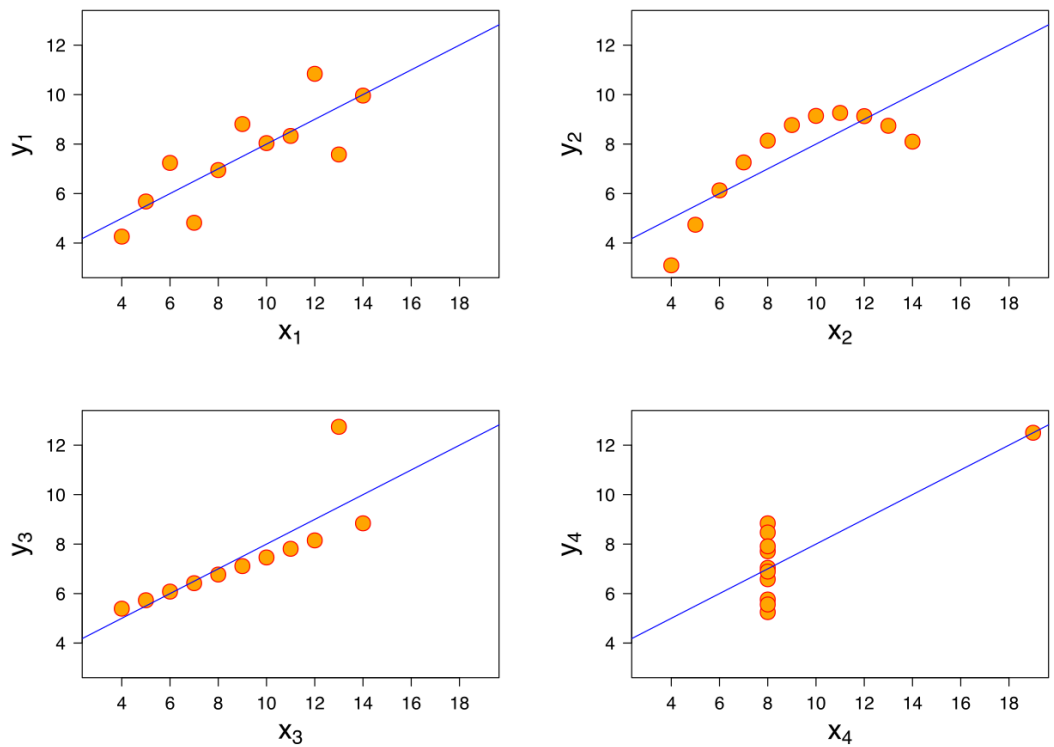
Anscombe's Data										
Observation	x1	y1		x2	y2		x3	y3		x4	y4
1	10	8.04		10	9.14		10	7.46		8	6.58
2	8	6.95		8	8.14		8	6.77		8	5.76
3	13	7.58		13	8.74		13	12.74		8	7.71
4	9	8.81		9	8.77		9	7.11		8	8.84
5	11	8.33		11	9.26		11	7.81		8	8.47
6	14	9.96		14	8.1		14	8.84		8	7.04
7	6	7.24		6	6.13		6	6.08		8	5.25
8	4	4.26		4	3.1		4	5.39		19	12.5
9	12	10.84		12	9.13		12	8.15		8	5.56
10	7	4.82		7	7.26		7	6.42		8	7.91
11	5	5.68		5	4.74		5	5.73		8	6.89
				Summary Statistics							
N	11	11		11	11		11	11		11	11
mean	9.00	7.50		9.00	7.500909		9.00	7.50		9.00	7.50
SD	3.16	1.94		3.16	1.94		3.16	1.94		3.16	1.94
r	0.82			0.82			0.82			0.82	

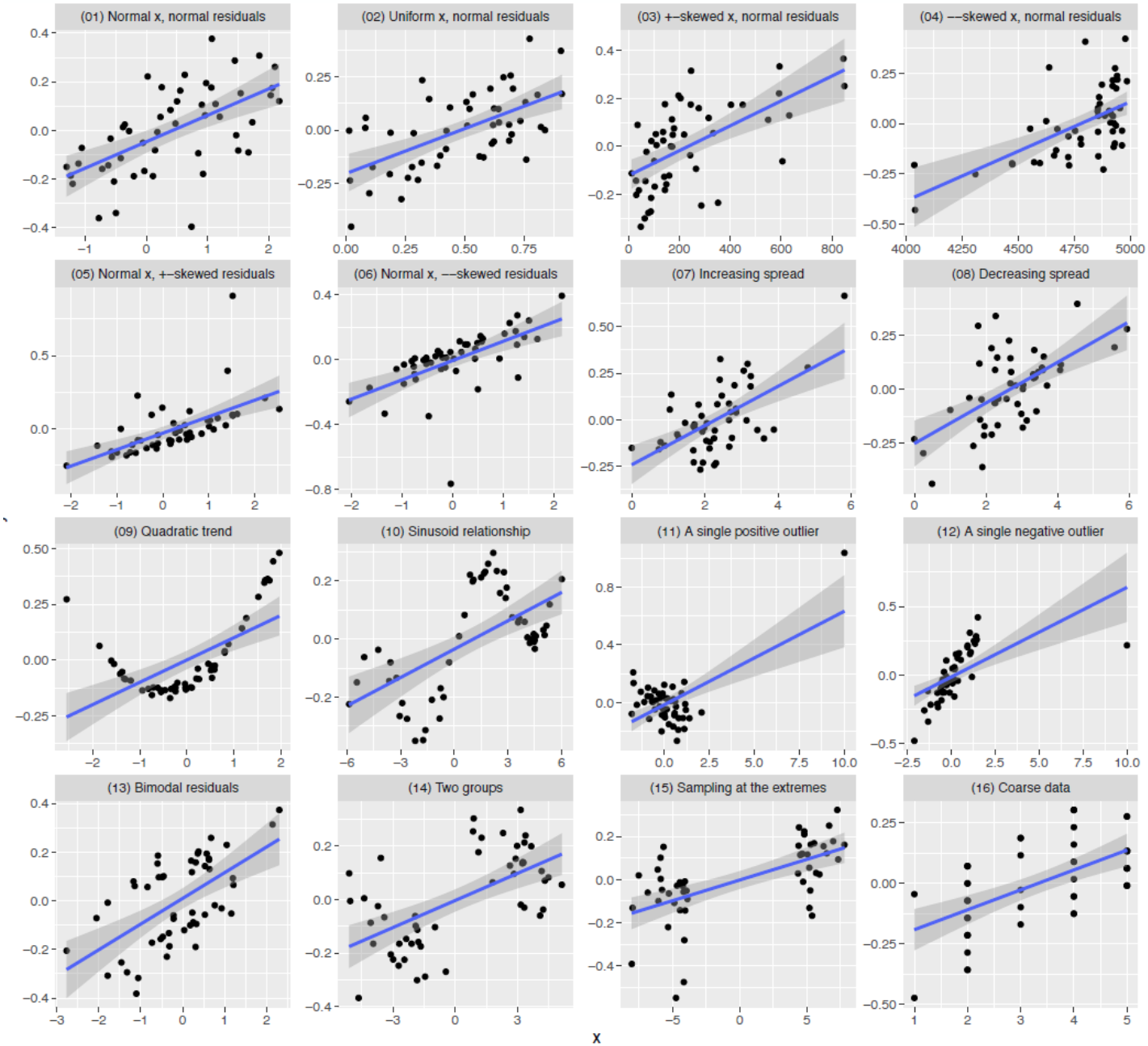
- The four data sets have similar descriptive statistics.

Why do we visualize?

The four data sets can be fitted to the same line: $y = 3 + 0.5x$

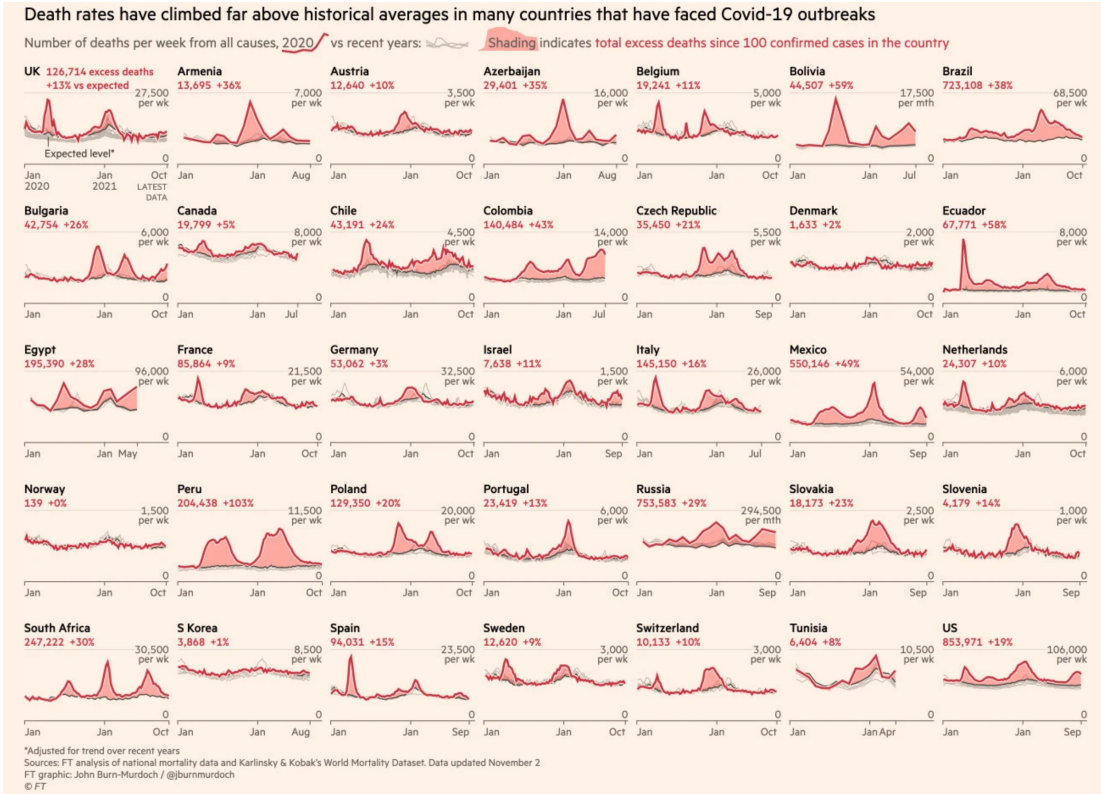
But each generates a different kind of plot:




(01) Normal x, normal residuals
(02) Uniform x, normal residuals
(03) +–skewed x, normal residuals
(04) ––skewed x, normal residuals
(05) Normal x, +–skewed residuals
(06) Normal x, ––skewed residuals
(07) Increasing spread
(08) Decreasing spread
(09) Quadratic trend
(10) Sinusoid relationship
(11) A single positive outlier
(12) A single negative outlier
(13) Bimodal residuals
(14) Two groups
(15) Sampling at the extremes
(16) Coarse data
x

Why do we visualize?

2. Convey information in a way that is otherwise difficult or impossible to convey.



Source: Financial Times, Dec 20, 2021.

Getting started

1. Prerequisites: Installing `R` and Rstudio
2. Basics in `R` programming
   - `R` objects
   - `R` syntax
   - `R` functions
   - Plotting in base `R`
   - Generate reports with `R` Markdown

What is R / RStudio?

- R is a statistical programming language.
- RStudio is an integrated development environment (IDE) for R.
  - A convenient interface that makes working with R much easier.

R: Engine

RStudio: Dashboard

Download and install R and RStudio

To download `R`, go to CRAN, the Comprehensive `R` Archive Network.

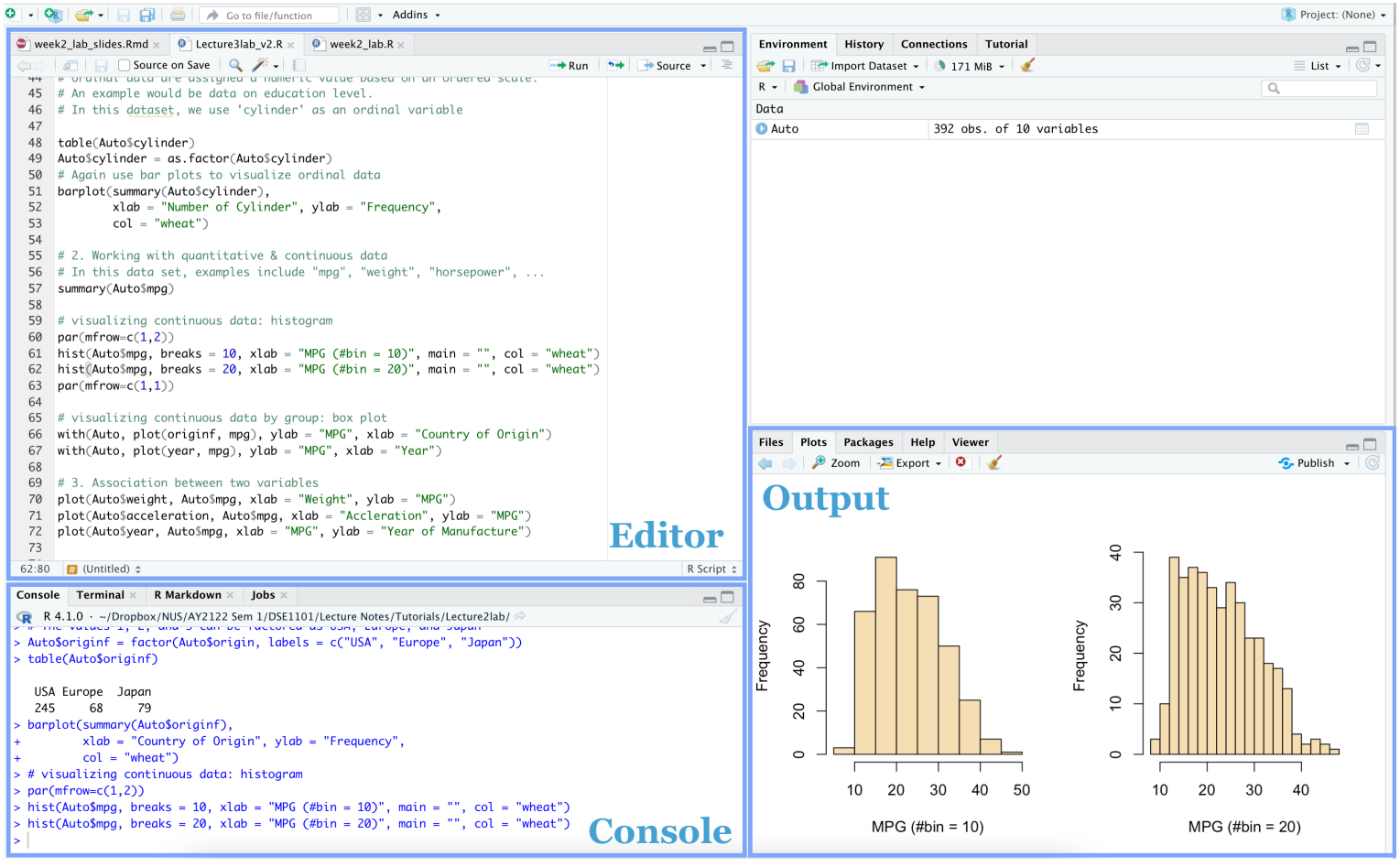
- http://cran.r-project.org/
- Download the version for your operation system and install it.

The installation file for **RStudio** can be obtained from this URL:

- http://www.rstudio.com/products/rstuidio/download

Key regions in the RStudio interface

Four panels: Editor, Console, Output, and Environment.



RStudio interface

1. **Editor**: is where you type `R` commands.
2. **Console**: shows the outputs from your `R` commands
3. **Environment**: lists all items that have been created in the current session.
   - History: displays all commands that have been previously entered in the current session
4. **Output**:
   - File: shows the structure of your working directory
   - Plots: includes all plots that have been created in the current session.
   - Help: displays documentation and help files.

Working directory

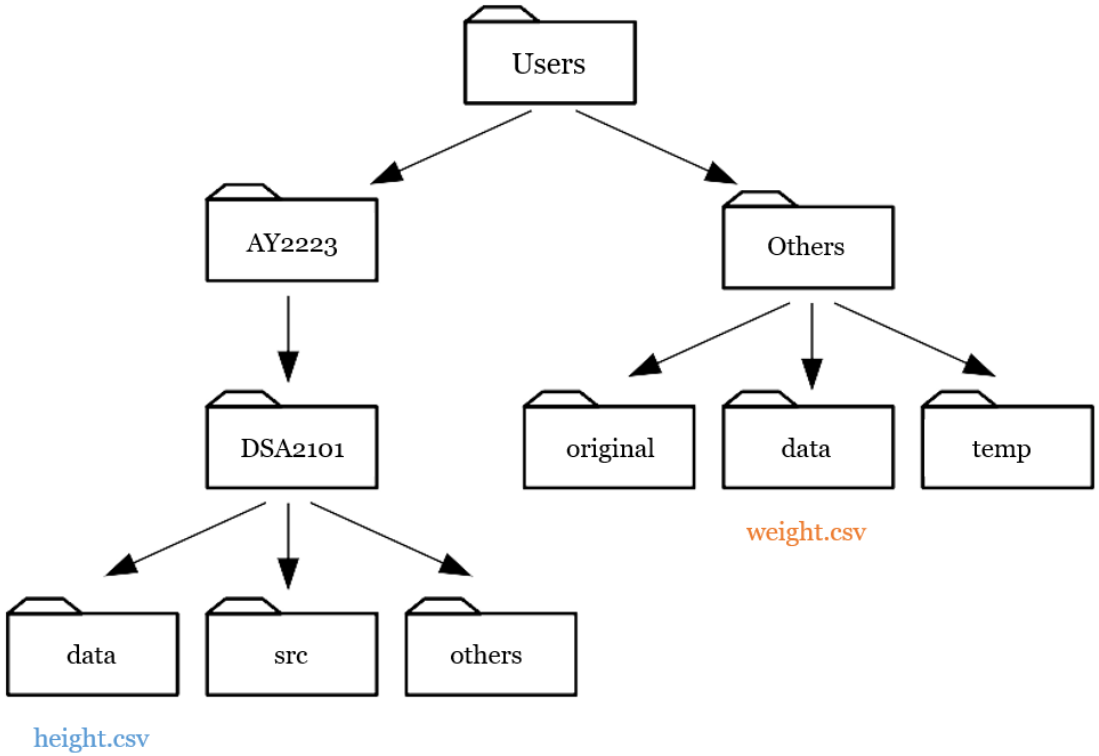
The working directory is where `R` looks when you ask it to read from a file or to write to a file.

- You can get your current directory with

```
getwd()
```

The function returns the **absolute path** of the current working directory.

- Absolute path: the exact address of a file on our computer.
- **Relative path**: the address of a file relative to our current working directory.



Let's say `getwd()` gives us `C:/Users/AY2223/DSA2101/src`

- To access **height** data: `../data/height.csv`
- To access **weight** data: `../../Others/data/weight.csv`

File path

We strongly recommend the following practice:

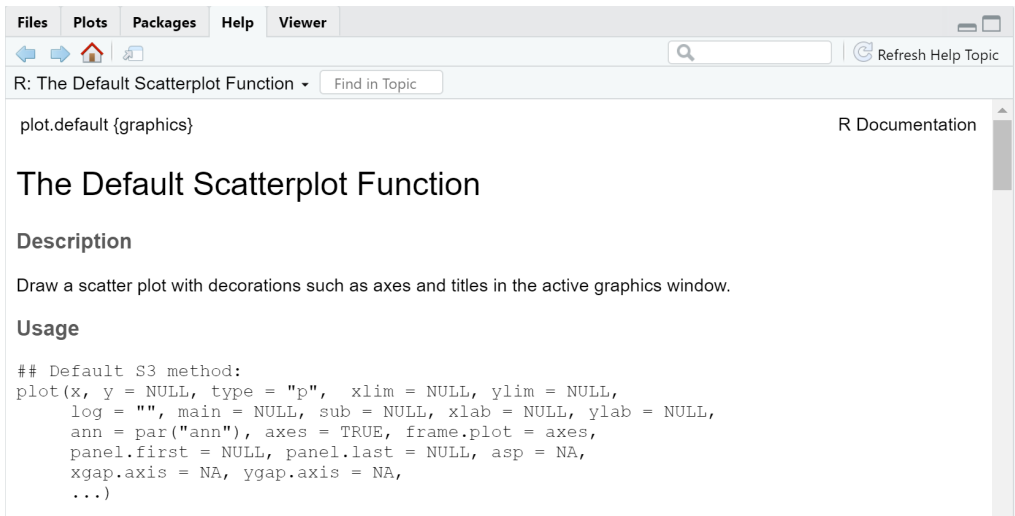
- create a folder **DSA2101** for our class and store all the code, data, and markdown files inside.
- Within DSA2101, create a folder called **src** to store all your `R` scripts and `Rmd` files.
- Within DSA2101, create another folder called **data** to store all your data files.
- The **src** and **data** folders should be at the same level.

Use relative path in all code you write.

To get help

If you need to find out more information about a particular function, e.g., what arguments it expects, what it does, etc, use the `?` operator.

```
?plot
```



It is **crucial** to be comfortable reading `R` help documentations.

Errors, warnings, and messages

One thing that intimidates new `R` and RStudio users is how it reports errors, warnings and messages.

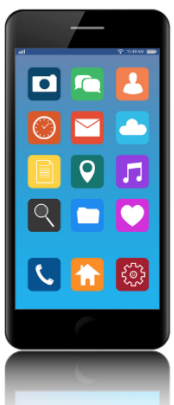
- "Error in. . . ": The code will not run.
- "Warning messages": Generally your code will still work, but with some caveats. R still produces results.
- When the red text doesn't start with either "Error" or "Warning", it is just a friendly message.

R packages

Another point of confusion with many new `R` users is the idea of `R` packages.

- `R` packages extend the functionality of `R` by providing additional functions and data.
  - `tidyverse`, `ggplot2`...

R: A new phone



R Packages: Apps you can download

General guidelines

Lectures and tutorials for this module will be highly **interactive** and require **lots of learning-by-doing**. + We highly encourage you to create an R script to follow along on your own laptop.

- To do so, navigate to `File -> New File -> R Script` to create a new `R` script for each lecture.
- Type the commands from the lecture notes in your script editor.
- Then just hit `Run` to see the outputs.

Basics in R programming

Week 1:

1. R objects
2. R syntax

Week 2:

3. R functions
4. Plotting in base R
5. Generate reports with R Markdown

R objects

When working in R, it is useful to remember two principles:

- Everything in R is an object.
- Every object has a class.

Basically, R uses an object-oriented style of programming.

If we know the class of an object, we can begin to understand what functions can be applied to it, and in general how it will behave under different conditions.

Naming objects in R

In the course of working in R, it is necessary to create new R objects, either to store data, or to store the output of a function.

- When doing so, remember to avoid the following words/letters as they are reserved by R:

```
# Reserved words in R
FALSE  Inf  NA  NaN  NULL  TRUE
break  else  for  function  if  in  next  repeat  while
c  q  s  t  C  D  F  I  T
```

- Also, R is case sensitive – `mydata` and `myData` refer to different objects.

Data types

Throughout the semester, we will cover four types of data:

- **Numeric**: numbers that contain a decimal.
- **Integers**: whole numbers.
- **Logical**: data that take on the value of either `TRUE` or `FALSE`.
- **Character**: data that are used to represent string values.
  - A special type of character string is called a **factor**, which is a string, but with additional attributes (e.g., levels or orders).

We can find out the type of any object using the `class()` function.

- The = is the assignment operator. Equivalently, we can use a left arrow (<-).

```
num = 2.7
class(num)
```

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

```
char = "hello"
class(char)
```

```
## [1] "character"
```

```
logi = TRUE
is.logical(logi)
```

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

Coercion functions

- Use the `class()` function to find out the type of an object.
- Use the following functions to conduct logical test and coerce objects.

Type	Testing	Coercing
numeric	`is.numeric()`	`as.numeric()`
logical	`is.logical()`	`as.logical()`
character	`is.character()`	`as.character()`
factor	`is.factor()`	`as.factor()`

Data structures

`R` has many data structures, including

- **Vector**: the basic building block for storing data.
- **Matrix**: two-dimensional object of the same type and length.
- **Data frame**: the most common structure to represent tabular (rectangular) data. It has a matrix-like structure, but its columns can be of differing types (e.g., numeric, logical, character, factor...).
- **List**: a collection of data objects. Elements can be of different types and/or length.

Vector

R has no scalars. The basic building block for storing data is a **vector**.

```
z = c(1, 2, 3)     # Create a vector of length 3, containing values 1, 2, 3
z                  # Print z
```

```
## [1] 1 2 3
```

- `c()` is a combine function that takes comma-separated numbers or strings and joins them into a vector.
- Now your environment contains the object `z`. Go and click on the **Environment** tab on RStudio. You should see it there.

Vector

Vectors can be created using different commands:

- The `:` operator tells `R` to create a sequence of integers.

```
-2:2
```

```
## [1] -2 -1  0  1  2
```

```
x = -2:2          # Assign the sequence to a vector named x
x                 # Print x
```

```
## [1] -2 -1  0  1  2
```

```
y = 2^x           # Raise 2 to powers given by x
y                 # Print y
```

```
## [1] 0.25 0.50 1.00 2.00 4.00
```

Select elements within a vector

Elements in a vector can be selected using a sequence of integers and the [ ] brackets.

```
y[2]              # Select the second element
```

```
## [1] 0.5
```

```
y[2:4]            # Select elements at indexes 2-4
```

```
## [1] 0.5 1.0 2.0
```

```
y[length(y)]      # Select the last element
```

```
## [1] 4
```

Class of a vector

The class of a vector is a property of a vector – dependent on the nature of its elements – we do not assign it.

- All elements of a vector will be of the same class.

```
firstname = c("adam", "brian", "cathy")      # character
avg = c(1.2)                                 # numeric, length = 1
pass = c(TRUE, FALSE, TRUE)                  # logical, length = 3
```

```
class(pass)                                  # Check the class type of a vector
```

```
## [1] "logical"
```

Matrix

A **matrix** is a two-dimensional collection of elements of the same data type.

```
mymat = matrix(1:9, nrow = 3, byrow = TRUE)
mymat
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
dim(mymat)            # Dimension of matrix
```

```
## [1] 3 3
```

Select elements within a matrix

We can select elements of a matrix using the bracket method:

```
mymat[2, 3]           # Select the element at the 2nd row and 3rd column
```

```
## [1] 6
```

```
mymat[2, ]           # Select elements in the 2nd row
```

```
## [1] 4 5 6
```

```
mymat[c(1, 2), ]     # Select elements in the 1st and 2nd rows
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Data frame

A **data frame** is the type of object normally used to store data in `R`.

- It is aligned with the concept of observation as rows, and variables as columns.
- The difference between a data frame and a matrix is that in a data frame, columns can be of different class types.

Here is an example of the operating budget for a company:

Budget category	**Amount**
Manpower	\$519.4m
Asset	\$38.0m
Other	\$141.4m

Let us manually create a data frame in `R`, containing this data.

Create data frames

The function `data.frame()` creates data frames. Objects passed to the function should have the same length.

```
budget_cat = c("Manpower", "Asset", "Other")
amount = c(519.4, 38.0, 141.4)
op_budget = data.frame(budget_cat, amount)
op_budget
```

```
##   budget_cat amount
## 1   Manpower  519.4
## 2      Asset   38.0
## 3      Other  141.4
```

Select elements of a data frame

Data frames and matrices are different types of objects, but their elements can be selected using a similar syntax.

```
op_budget[, "budget_cat"]   # Select the budget category
```

```
## [1] "Manpower" "Asset"    "Other"
```

- More often, we use the $ symbol to select individual column(s) in a data frame:

```
op_budget$budget_cat
```

```
## [1] "Manpower" "Asset"    "Other"
```

Example: `cars` data

There are several example data frames stored in base `R`. Let us inspect the `cars` data set.

```
data(cars)              # load data
class(cars)             # check types of the object
```

```
## [1] "data.frame"
```

```
names(cars)             # see the column names
```

```
## [1] "speed" "dist"
```

```
head(cars, n = 3)       # see the first 3 rows of the data frame
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
```

List

A data frame is a special type of structure in `R`, known as a **list**.

- A list is a **collection of objects**.
- These objects can be different from one another.
- ... can also be of different lengths, or even lists of their own.
- A very flexible data structure, ideal for storing irregular or non-rectangular data.

```r
mylist = list(A = seq(1, 5, by = 2),
              B = c(TRUE, FALSE, TRUE),
              C = matrix(1:6, nrow = 3)) # a list of 3 objects
mylist
```

```
## $A
## [1] 1 3 5
##
## $B
## [1]  TRUE FALSE  TRUE
##
## $C
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```r
mylist[[2]]     # Use double square brackets to access a object
```

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

```r
mylist$B        # Alternatively, use $ and access a object by its name
```

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

Save and read R objects

The `saveRDS()` and `readRDS()` functions allow for an object to be saved and retrieved for use.

```
saveRDS(mylist, file = "../data/mylist.rds")
mylist2 = readRDS("../data/mylist.rds") # Read in the object as mylist2
mylist2
```

```
## $A
## [1] 1 3 5
##
## $B
## [1]  TRUE FALSE  TRUE
##
## $C
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Example: Hawker center data

Sometimes, we may be given an object without much information about it.

- The `str()` function can be very useful to learn about the **structure** of the object.
- The RDS file below contains a list of hawker centers, retrieved from onemap.sg.

```
hawkers = readRDS("../data/hawker_ctr_raw.rds")
class(hawkers)
```

```
## [1] "list"
```

```
str(hawkers, max.level = 1)
```

```
## List of 1
##  $ SrchResults:List of 117
```

Structure of a list

Thus, the `hawkers` data set is a list of length 1. What is in that list?

```
# Try this
str(hawkers[[1]], max.level = 1)
```

- It is a list of 117 sublists.
- The first sublist is of length 1.
- The remaining 116 are of length 12. What do they contain?

Structure of a list

Each of those 116 sublists contains information on one hawker center.

```
str(hawkers[[1]][[7]])   # Inspect items in the 7th sublist
```

```
## List of 12
##  $ ADDRESSBUILDINGNAME    : chr ""
##  $ ADDRESSFLOORNUMBER     : chr ""
##  $ ADDRESSPOSTALCODE      : chr "320091"
##  $ ADDRESSSTREETNAME      : chr "Whampoa Drive"
##  $ ADDRESSUNITNUMBER      : chr ""
##  $ DESCRIPTION            : chr "HUP Standard Upgrading"
##  $ HYPERLINK              : chr ""
##  $ NAME                   : chr "Blks 91/92 Whampoa Drive"
##  $ PHOTOURL               : chr ""
##  $ ADDRESSBLOCKHOUSENUMBER: chr "91/92"
##  $ XY                     : chr "30309.21,33962.7799"
##  $ ICON_NAME              : chr "HC icons_Opt 8.jpg"
```

In Week 3, we will learn how to deal with this data object.

Remove objects from the workspace

As you work, you may find that you accumulate a number of objects within your workspace.

- You can inspect them in the **Environment** tab, and remove them there.
- You can also remove them with the `rm()` function.

```
x = 1:5
rm(list = c("x"))     # Remove object named x
rm(list = ls())       # Remove ALL objects from the workspace
```

R Syntax

In R, commands that we enter into the console are either **expressions** or **assignments**.

- An expression is evaluated and printed:

```
pi + 1
```

```
## [1] 4.141593
```

- In an assignment, the expression portion of it is first evaluated. After that, the output is passed to a variable. The result is not printed:

```
a = pi + 1
```

The `=` is the **assignment operator**. Equivalently, you can use `<-`.

Arithmetic expressions

The basic unit in R is a vector. Arithmetic operations are performed element by element.

```
x = 3
y = 4
x + y    # addition
x - y    # subtraction
x * y    # multiplication
x / y    # division
x ^ y    # exponentiation
```

For more arithmetic operations, type the following into your console:

```
?Arithmetic
```

Vectorized operations

- Most of R's functions are vectorized.

```
x = 5:10
y = 10:15
x + y
```

```
## [1] 15 17 19 21 23 25
```

- What happens when we do this?

```
x + 2
```

```
## [1]  7  8  9 10 11 12
```

- R uses a **recycling rule** whenever it is presented with vectors of varying lengths in an expression.

Recycling rule

When presented with vector of varying lengths, the output is a vector with the same length as the longest vector in the expression.

- Shorter vectors are recycled as often as needed, until they match the length of the longest vector.
- In particular, a single number is repeated an appropriate number of times.

```
x = 5:10          # A vector of length 6
y = c(x, x)       # A vector of length 12
2*x + y + 1       # A vector of length 12
```

```
##  [1] 16 19 22 25 28 31 16 19 22 25 28 31
```

Logical expressions

Logical vectors are vectors of `TRUE` and/or `FALSE` values.

- These are often generated by conditions.
- When the following binary operators are applied to numeric vectors, the output will be a logical vector:

```
x < y
x <= y
x > y
x >= y
x == y
x != y
```

Examples of logical expressions

```
x = 1:5
x < 3
```

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

```
x == 1
```

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

```
x != 17
```

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

The recycling rule is in play for all of the above expressions.

Logical expressions

It is also possible to find the element-wise intersection, union, and negation of vector-valued logical expressions

```
y = x <= 3; z = x >= 3
y & z          # Intersection
```

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

```
y | z          # Union
```

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

```
!y             # Negation
```

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

Selection of logical expressions

Logical vectors can be used to select a subset of elements from a vector:

```
x
```

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

```
y
```

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

```
x[y]
```

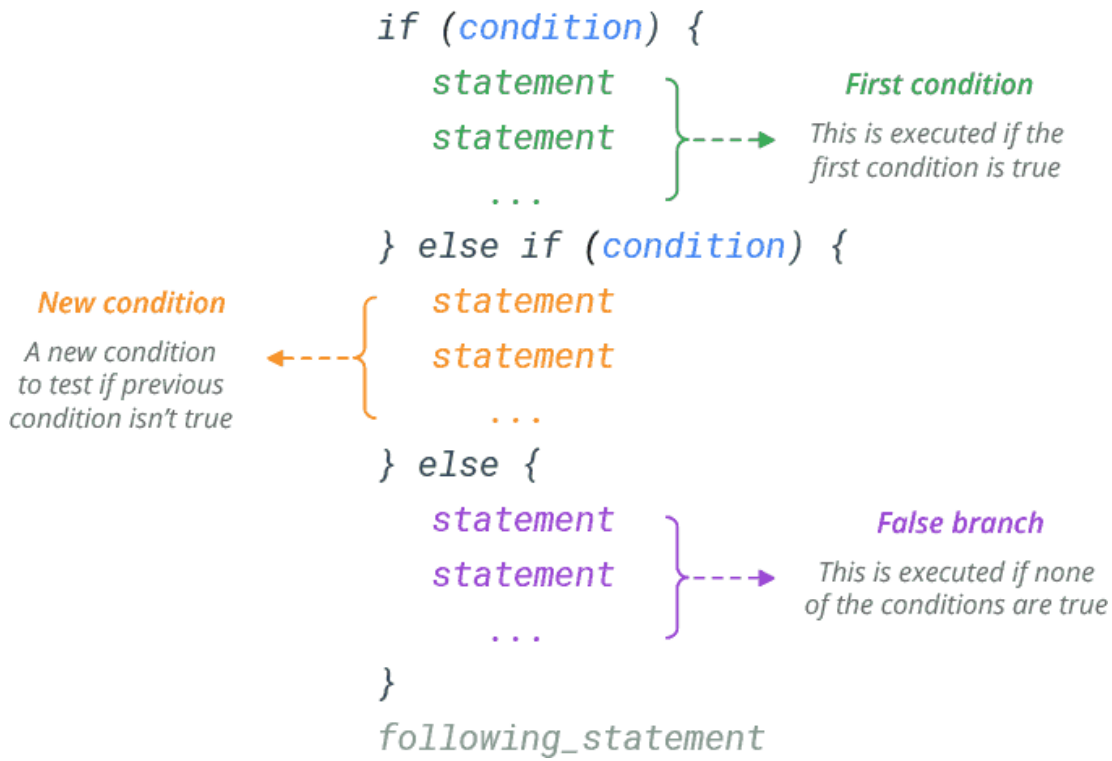
```
## [1] 1 2 3
```

We will return to this particular use of logical vectors when learning about subsetting data.

Conditional executions and loops

Many times, we want to execute a certain set of instructions if a condition is `TRUE`, and another set of instructions otherwise.

- In this case, we use `if-else` statements.



Rolling a die

Suppose that we wish to simulate a game of dice between two players, A and B. They each have a fair six-sided die. The player with the higher roll is the winner. If they both get the same number, the result is a draw.

- Simulate a roll of a six-sided die:

```
sample(1:6, size = 1)  # Randomly select one number from vector 1:6
```

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

- If we wish to roll two dice, then

```
sample(1:6, size = 2, replace = TRUE)      # Sample with replacement
```

```
## [1] 6 6
```

Conditional executions of the dice game

Players A and B roll the dice

```
set.seed(2101)                  # Set seed for reproducibility
A = sample(1:6, size = 1)
B = sample(1:6, size = 1)
```

Compare the results and determine the winner

```
if (A > B) {
  print("A is the winner.")
} else if (A == B) {
  print("It is a draw.")
} else {
  print("B is the winner.")
}
```

```
## [1] "B is the winner."
```

Repeated executions

Now suppose we want to simulate the game for 1000 times.

We can use a `for loop`. For each iteration, we store the result in a character vector named `results`

```
set.seed(2101)                  # For reproducibility
results = rep(0, 1000)          # results vector, length = 1000

for (i in 1:1000) {             # The for loop begins here
  A = sample(1:6, size = 1)
  B = sample(1:6, size = 1)

  if (A > B) {
    results[i] = "A"
  } else if (A == B) {
    results[i] = "Draw"
  } else {
    results[i] = "B"
  }
}
```

Repeated executions

Summary table for the 1000 results:

```
table(results)
```

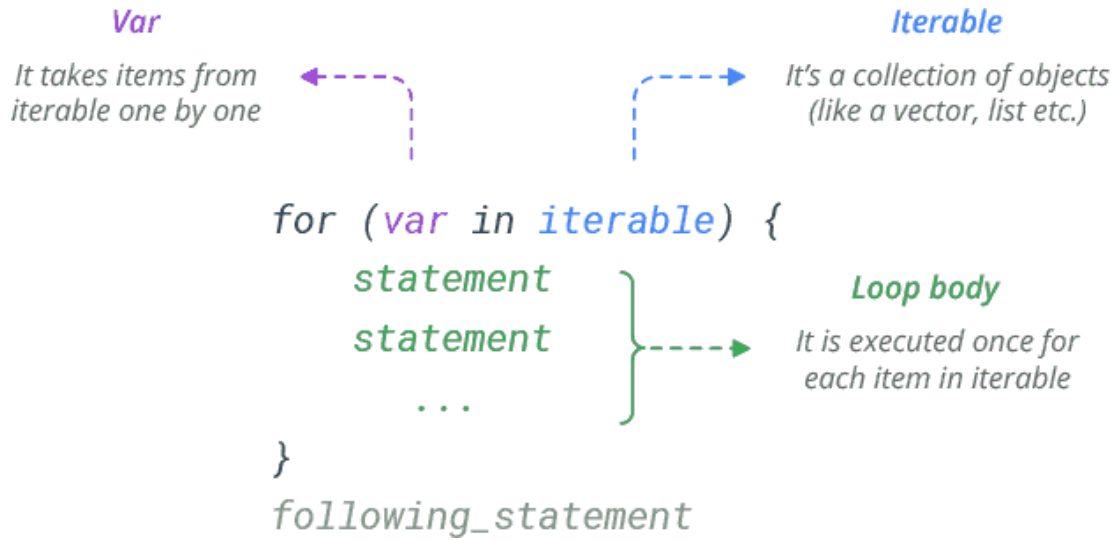
```
## results
##    A    B Draw
##  419  431  150
```

The proportion of times that a draw results is 0.150.

Does this agree with theory?

For loop

What we just saw is a `for loop`. It iterates over items of a vector or a list and execute the statements in each iteration.



Combine conditional and repeated execution

What if we want to stop the game as soon as A wins?

- In this case, we need to use a `break` statement, to break out of the loop.
- Although it is possible to write a `for` loop to do this, we do not need to. We only need a loop that will continue until a particular condition is reached.

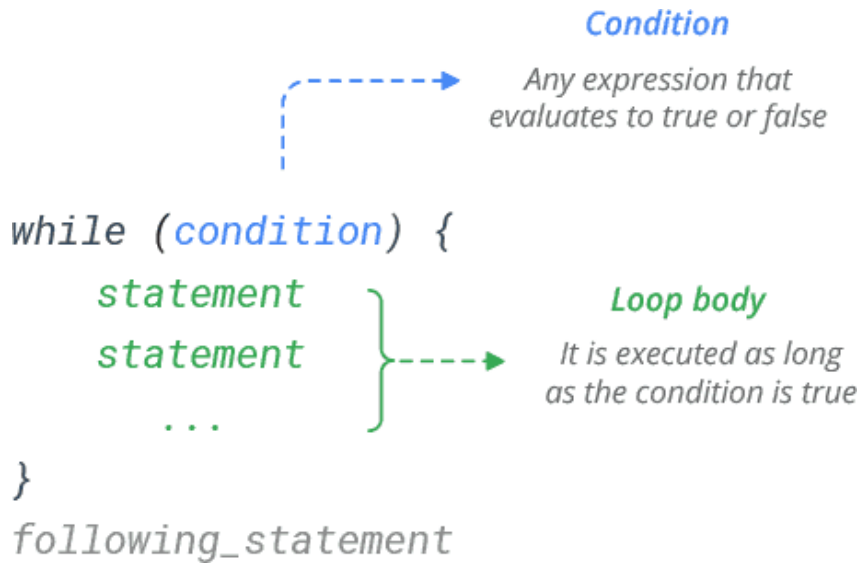
Example: First win for player A

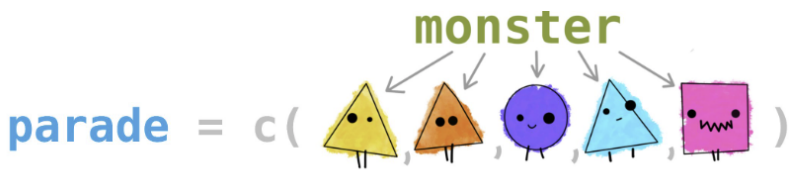
```
set.seed(2101)
game_counter = 0

while(TRUE) {                    # A while loop starts here
  A = sample(1:6, size = 1)
  B = sample(1:6, size = 1)

  game_counter = game_counter + 1
  if (A > B)
    break
}

cat(paste("First win for A occurs at game", game_counter))
```

```
## First win for A occurs at game 2
```

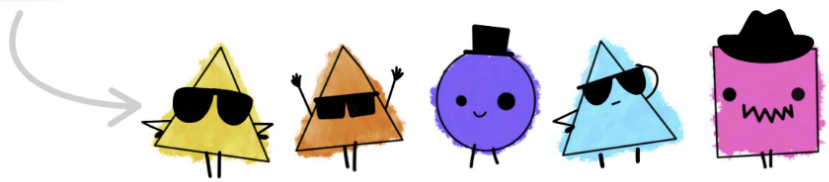
While loop

- What we just saw in the outer loop is a `while loop`.
- It is used when we want to perform a task indefinitely, **until** a certain condition is met.






```
for (monster in parade) {

        if (shape(monster) == triangle) {
            monster_style = monster + sunglasses
        }

        else {
            monster_style = monster + hat
        }

        print(monster_style)

}
```



Artwork by Allison Horst

Summary

- **Getting ready**: Download and install the latest versions of `R` and `RStudio`.
- **Basics in `R` programming**
  - **`R` objects**
    - Data types
    - Data structures
    - Accessing element(s) in data objects
  - **`R` syntax**
    - Expressions and assignments
    - Vectorized operations
    - Conditional executions: `if-else` statements
    - Repeated executions: `for` and `while` loops