R for Research in Action

Francisco J. Navarro

2024-09-19

Table of contents

Preface

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris sed nisi porta, lobortis mi eget, laoreet sapien. Etiam sit amet tellus luctus, rhoncus dolor pharetra, faucibus sem. Nulla viverra risus et urna porttitor mattis. In et tortor a justo convallis ultricies. Praesent consectetur bibendum tellus eget malesuada. In fermentum sodales luctus. Curabitur in ornare eros. In pulvinar tempor odio et cursus. Aenean tristique vel leo vel congue. Pellentesque id metus mollis, tincidunt nibh a, dapibus elit. Etiam at augue et nisl volutpat ornare pulvinar vitae urna. Duis auctor urna quam, eget rhoncus diam fermentum at.

Praesent scelerisque efficitur euismod. Aliquam neque augue, fermentum a magna vitae, interdum varius turpis. Fusce commodo maximus magna sit amet rutrum. Nulla sit amet sem a eros tempus finibus non ac leo. Donec sit amet gravida eros. Donec ut lacus eget velit aliquet luctus. Ut pellentesque tellus varius, gravida velit eu, finibus ante. Maecenas vehicula, elit non volutpat mollis, est augue placerat orci, nec condimentum neque nulla vulputate elit. Nunc hendrerit hendrerit cursus. Aenean elementum mi dolor, sed maximus ex rutrum convallis. Nulla lobortis faucibus justo, in elementum ligula semper non. Quisque sollicitudin pretium tortor vel sollicitudin. Curabitur quis luctus ante. Ut sit amet dui consequat, tristique diam a, suscipit lacus.

Part I

Introduction

1 R Objects

1.1 Classes

R has five basic or “atomic” **classes of objects**:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

Entering `1` in R gives you a numeric object; entering `1L` explicitly gives you an integer object.

1.2 Attributes

Attributes of an object (if any) can be accessed using the `attributes()` function:

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class (e.g. integer, numeric)
- length

The `mode()` of an object tells us how it’s stored. It could happen that two different objects are stored in the same mode with different classes.

- For **vectors** the class and mode will always be numeric, logical, or character.
- For **matrices and arrays** a class is always a matrix or array, but its mode can be numeric, character, or logical.

The primary purpose of the `class()` function is to know how different functions, including generic functions, work (e.g. print, or plot). There is a collection of R commands used to assess whether a particular object belongs to a certain class, these start with **is.**; for example, `is.numeric()`, `is.logical()`, `is.character()`, `is.list()`, `is.factor()`, and `is.data.frame()`

1.3 Mixing Objects

This is not allowed! When different objects are mixed in a vector, coercion occurs so that every element in the vector is of the same class.

```r
y <- c(1.7, "a")
class(y)
```

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

```r
y <- c(TRUE, 2)
class(y)
```

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

```r
y <- c("a", TRUE)
class(y)
```

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

1.4 Explicit Coercion

Objects can be explicitly coerced from one class to another using the `as.` functions, if available.

```r
x <- 0:6
class(x)
```

```
[1] "integer"
```

```r
as.numeric(x)
```

```
[1] 0 1 2 3 4 5 6
```

1.5 Names

Object names must start with a letter and can only contain letters, numbers, `_`, and `.`.

You want your object names to be descriptive, so you'll need to adopt a convention for multiple words.

Example:

```
i_use_snake_case
otherPeopleUseCamelCase
some.people.use.periods
```

1.6 R Workspace

The **objects** that you create using R remain in existence until you explicitly delete them or you conclude the session.

- To **list** all currently defined objects, use `ls()` or `objects()`.
- To **remove object x**, use `rm(x)`. To **remove all** currently defined objects, use `rm(list = ls())`.
- To **save all** of your existing objects to a file called `fname` in the current working directory, use `save.image(file = "fname")`.
- To **save specific** objects (say x and y) use `save(x, y, file = "fname")`.
- To **load a set** of saved objects use `load(file = "fname")`.
- To save this history to the file `fname` use `savehistory(file = "fname")` and to load the history file fname use `loadhistory(file = "fname")`.

1.7 Expressions and Assignments

In R an **expression** is used to denote a phrase of code that can be executed.

Example:

```
## An expression
seq(10, 20, 3)
```

The combination of expressions that are saved for evaluation is called an **assignment**:

```r
## An assignment
object_name <- value
```

When reading that code, say "object name gets value" in your head. You will make lots of assignments, and `<-` is a pain to type.

You can save time with RStudio's keyboard shortcut: **Alt + -** (the minus sign).

2 Data Types

2.1 Vectors

A **vector** is the most convenient way to store more than one data value.

A **vector** is a contiguous cell that contains data, where each cell can be accessed by an index. In other words, a vector is an indexed set of objects.

All the elements of an atomic vector have to be of the **same type** —numeric, character, or logical— which is called the mode of the vector.

2.1.1 How to create a vector?

There are many ways to create a vector, but these are four basic functions for constructing vectors:

1. The `c()` (combine) function can be used to create vectors of objects by concatenating things together.

   Examples:

   ```
   x <- c(0.5, 0.6)
   class(x)
   ```

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

   ```
   x <- c(TRUE, FALSE)
   class(x)
   ```

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

   ```
   x <- c(T, F)
   class(x)
   ```

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

```r
x <- c("a", "b", "c")
class(x)
```

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

```r
x <- 9:29
class(x)
```

```
[1] "integer"
```

```r
x <- c(1+0i, 2+4i)
class(x)
```

```
[1] "complex"
```

2. `seq(from, to, by)`:

   ```r
   (x <- seq(1, 20, 2))
   ```

   ```
    [1]  1  3  5  7  9 11 13 15 17 19
   ```

3. `rep(x, times)`:

   ```r
   (y <- rep(3, 4))
   ```

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

4. You can also use the `vector()` function to initialize vectors:

   Example:

   ```r
   x <- vector("numeric", length = 10)
   class(x)
   ```

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

2.1.2 Add a value to a variable

Example:

```r
x[4] <- 9
```

2.1.3 Add names to a vector

```
# Create a vector with the name of each element
named.num.vec <- c(x1=1, x2=3, x3=5)
named.num.vec
```

```
x1 x2 x3
 1  3  5
```

This is another option to add names:

```
names(x) <- c("a", "b", "c")
```

2.1.4 length()

The function `length(x)` gives the number of elements of 'x'.

```
x <- 100:100
length(x)
```

```
[1] 1
```

It is possible to have a vector with no elements

```
x <- c()
length(x)
```

```
[1] 0
```

2.2 Factors

Statisticians typically recognise **three basic types of variable**: numeric, ordinal, and categorical. In R the data type for ordinal and categorical vectors is **factor**. The possible values of a factor are referred to as its **levels**.

In practice, a **factor** is not much different from a character vector, except that the elements of a factor can take only a limited number of values (of which R keeps a record), and in statistical routines R is able to treat a factor differently than a character vector.

To create a factor we apply the function `factor()` to some vector `x`. By default the distinct values of `x` become the **levels**, or we can specify them using the optional `levels` argument.

Example:

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
table(x)
```

```
x
 no yes
  2   3
```

```
levels(x)
```

```
[1] "no"  "yes"
```

Note the use of the function `table()` to calculate the number of times each level of the factor appears. `table()` can be applied to other modes of vectors as well as factors. The output of the `table()` function is a one-dimensional array (as opposed to a vector). If more than one vector is passed to `table()`, then it produces a multidimensional array.

The order of the **levels** of a factor can be set using the levels argument to `factor()`. By default R arranges the levels of a factor alphabetically. If you specify the levels yourself, then R uses the ordering that you provide.

Example:

```
x <- factor(c("yes", "yes", "no", "yes", "no"), levels = c("yes", "no"), ordered=TRUE)
x
```

```
[1] yes yes no  yes no
Levels: yes < no
```

Using factors with **labels** is better than using integers because factors are self-describing.

Example:

```
phys.act <- factor(phys.act, levels = c("L", "M", "H"),
  labels = c("Low", "Medium", "High"),
  ordered = TRUE)
```

We check whether or not an object `x` is a factor using `is.factor(x)`.

Example:

```
is.factor(x)
```

```
[1] TRUE
```

Usually it is convenient to transform a numeric variable into a data.frame:

```
airquality <- transform(airquality, Month = factor(Month))
```

`cut()` is a generic command to create factor variables from numeric variables:

Example:

```
numvar <- rnorm(100)
num2factor <- cut(numvar, breaks=5) ## the levels are produced using the actual range of v
num2factor
```

```
  [1] (0.567,1.62]   (-0.489,0.567] (-1.55,-0.489] (-0.489,0.567] (-2.61,-1.55]
  [6] (-0.489,0.567] (-1.55,-0.489] (-1.55,-0.489] (-0.489,0.567] (0.567,1.62]
 [11] (0.567,1.62]   (-0.489,0.567] (0.567,1.62]   (-2.61,-1.55]  (0.567,1.62]
 [16] (-0.489,0.567] (-0.489,0.567] (-1.55,-0.489] (-0.489,0.567] (-0.489,0.567]
 [21] (-0.489,0.567] (0.567,1.62]   (-1.55,-0.489] (0.567,1.62]   (-0.489,0.567]
 [26] (-2.61,-1.55]  (0.567,1.62]   (1.62,2.69]    (-0.489,0.567] (-0.489,0.567]
 [31] (0.567,1.62]   (-1.55,-0.489] (-0.489,0.567] (-0.489,0.567] (0.567,1.62]
 [36] (-0.489,0.567] (-1.55,-0.489] (-0.489,0.567] (-0.489,0.567] (-0.489,0.567]
 [41] (0.567,1.62]   (-0.489,0.567] (-1.55,-0.489] (-0.489,0.567] (-1.55,-0.489]
 [46] (-0.489,0.567] (-1.55,-0.489] (-0.489,0.567] (-1.55,-0.489] (-1.55,-0.489]
 [51] (-0.489,0.567] (-0.489,0.567] (-1.55,-0.489] (-0.489,0.567] (-0.489,0.567]
 [56] (-0.489,0.567] (0.567,1.62]   (0.567,1.62]   (1.62,2.69]    (-1.55,-0.489]
 [61] (-0.489,0.567] (-0.489,0.567] (-0.489,0.567] (-1.55,-0.489] (-0.489,0.567]
 [66] (-1.55,-0.489] (-1.55,-0.489] (-0.489,0.567] (-0.489,0.567] (-0.489,0.567]
 [71] (-0.489,0.567] (-1.55,-0.489] (-1.55,-0.489] (-1.55,-0.489] (-0.489,0.567]
 [76] (0.567,1.62]   (0.567,1.62]   (-1.55,-0.489] (-0.489,0.567] (-0.489,0.567]
 [81] (-0.489,0.567] (0.567,1.62]   (-2.61,-1.55]  (-1.55,-0.489] (-1.55,-0.489]
 [86] (-1.55,-0.489] (0.567,1.62]   (-1.55,-0.489] (-1.55,-0.489] (-0.489,0.567]
 [91] (-0.489,0.567] (-1.55,-0.489] (-0.489,0.567] (1.62,2.69]    (-0.489,0.567]
 [96] (-0.489,0.567] (-0.489,0.567] (0.567,1.62]   (-0.489,0.567] (-1.55,-0.489]
5 Levels: (-2.61,-1.55] (-1.55,-0.489] (-0.489,0.567] ... (1.62,2.69]
```

```r
num2factor <- cut(numvar, breaks=5, labels= c("lowest group", "lower middle group", "middl
data.frame(table(num2factor)) ## displaying the data in tabular form
```

```
          num2factor Freq
1       lowest group    4
2 lower middle group   28
3       middle group   47
4       upper middle   18
5      highest group    3
```

2.3 Matrices

Matrices are stored as vectors with an added dimension attribute. The dimension attribute is itself an integer vector of length 2, which gives the number of rows and columns.

The matrix elements are stored **column-wise** in the vector. This means that it is possible to access the matrix elements using a single index:

```r
(A <- matrix(c(3,5,2,3), 2, 2))
```

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

```r
A[2]
```

```
[1] 5
```

```r
A[,2]
```

```
[1] 2 3
```

2.3.1 How to create a matrix?

Matrices are constructed column-wise, so entries can be thought of starting in the "upper left" corner and running down the columns.

Example:

```r
(m <- matrix(1:6, nrow = 2, ncol = 3, byrow = FALSE))
```

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

```r
dim(m)
```

```
[1] 2 3
```

Matrices can be created by column-binding or row-binding with the `cbind()` and `rbind()` functions:

Examples:

```r
x <- 1:3
y <- 10:12
cbind(x, y)
```

```
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
```

```r
rbind(x, y)
```

```
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

Matrices can also be created directly from **vectors** by adding a **dimension attribute**:

Example:

```r
m <- 1:10
dim(m) <- c(2, 5)
m
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

To **create a matrix**`A` with **one column** from a vector `x`, we use:

```
A <- as.matrix(x)
```

To **create a vector from a matrix A**, we use:

```
x <- as.vector(A)
```

2.3.2 Create a diagonal matrix

To create a diagonal matrix use `diag(x)`:

```
B <- diag(c(1,2,3))
```

2.3.3 Add names to matrices

Matrices can have **names**:

Example:

```
dimnames(m) <- list(c("a", "b"), c("c", "d", "e"))
m
```

These are other options for **column and row names**:

```
colnames(m) <- c("c", "d", "e")
```

```
rownames(m) <- c("a", "b")
```

2.3.4 Operations with matrices

To perform **matrix multiplication** we use the operator `%*%`. Remember that `*` acts element wise on matrices.

Other functions for using with matrices are:

- `nrow(x)`
- `ncol(x)`
- `det(x)` (the determinant)
- `t(x)` (the transpose)
- `solve(A, B)` (returns x such that A %*% x == B).
- If A is invertible then solve(A) returns the matrix inverse of A.

2.4 Data Frames

It is a list of vectors restricted to be of **equal length**. Each vector —or **column**— corresponds to a variable in an experiment, and each **row** corresponds to a single observation or experimental unit. Each vector can be of any of the basic modes of object.

The **dataframe** is like a **matrix** but extended to allow for different object modes in different columns. Unlike matrices, data frames can store **different classes of objects** in each column (matrices must have every element be the same class, e.g. all integers or all numeric). Obviously to work with datasets from **real experiments** we need a way to group data of differing modes.

Data frames are used to store tabular data in R. **Data frames** are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.

Data frames are usually created by:

1. reading in a dataset using the `read.table()` or `read.csv()`
2. creating a dataframe with `data.frame()`:

   Example:

   ```
   (x <- data.frame(foo = 1:4, bar = c(T, T, F, F)))
   ```

   ```
     foo   bar
   1   1  TRUE
   2   2  TRUE
   3   3 FALSE
   4   4 FALSE
   ```

   ```
   # To summarise the structure of a list (or dataframe), use str()
   str(x)
   ```

```
'data.frame':	4 obs. of  2 variables:
 $ foo: int  1 2 3 4
 $ bar: logi  TRUE TRUE FALSE FALSE
```

3. coerced from other types of objects like lists:

```
x <- as.data.frame(x)
```

Dataframes can be converted to a **matrix** by calling `data.matrix()`.

The **dplyr package** has an optimized set of functions designed to work efficiently with dataframes.

2.4.1 Columns and Rows

You can construct a dataframe from a **collection of vectors and/or existing dataframes** using the function `data.frame`, which has the form: `data.frame(col1 = x1, col2 = x2, ..., df1, df2, ...)`. Here `col1`, `col2`, etc., are the column names (given as character strings without quotes) and `x1`, `x2`, etc., are vectors of equal length. `df1`, `df2`, etc., are dataframes, whose columns must be the same length as the vectors `x1`, `x2`, etc. Column names may be omitted, in which case R will choose a name for you.

Column names indicate the names of the variables or predictors `names()`. We can also create a new variable within a dataframe, by naming it and assigning it a value:

Example:

```
ufc$volume.m3 <- pi * (ufc$dbh.cm / 200)^2 * ufc$height.m / 2
```

Equivalently one could assign to `ufc[6]` or `ufc["volume.m3"]` or `ufc[[6]]` or `ufc[["volume.m3"]]`.

The command `names(df)` will return the names of the dataframe df as a vector of character strings.

Example:

```
ufc.names <- names(ufc)
# To change the names of df you pass a vector of character strings to `names(df)`
names(ufc) <- c("P", "T", "S", "D", "H", "V")
```

When you create dataframes and any one of the column's classes is a **character**, it automatically gets converted to factor, which is a default R operation. However, there is one argument, `stringsAsFactors=FALSE`, that allows us to prevent the automatic conversion of character to factor during data frame creation.

Dataframes have a special attribute called `row.names()` which indicate information about each **row** of the data frame. You can change the **row names** of df by making an assignment to `row.names(df)`.

2.4.2 `subset()`

The function `subset()` is a convenient tool for selecting the **rows of a dataframe**, especially when combined with the operator **%in%**.

Example:

```
# Suppose you are only interested in the height of trees of species DF (Douglas Fir) or GF
fir.height <- subset(ufc, subset = species %in% c("DF", "GF"),
                     select = c(plot, tree, height.m))
head(fir.height)
```

2.4.3 `attach()`

R allows you to **attach a dataframe** to the workspace. When attached, the variables in the dataframe can be referred to without being prefixed by the name of the dataframe.

Example:

```
attach(ufc)
max(height.m[species == "GF"])
```

When you attach a dataframe R actually makes a **copy of each variable**, which is deleted when the dataframe is detached. Thus, if you change an attached variable you do not change the dataframe. After we use the `attach()` command, we need to use `detach()` to remove individual variables from the working environment.

Nonetheless, note that the `with()` and `transform()`functions provide a safer alternative.

2.5 Lists

Lists are a special type of vector that can contain elements of **different type** (we can store single constants, vectors of numeric values, factors, data frames, matrices, and even arrays), namely, a list is a general data storage object that can house pretty much any other kind of R object.

Like a vector, a list is an **indexed set of objects** (and so has a length), but unlike a vector the elements of a list can be of different types, including other lists! The **mode** of a list is *list*.

The **power and utility** of lists comes from this generality. A list might contain an individual measurement, a vector of observations on a single response variable, a dataframe, or even a list of dataframes containing the results of several experiments.

In R lists are often used for collecting and storing **complicated function output**. **Dataframes** are special kinds of lists.

2.5.1 How to Create a List?

Lists can be explicitly created using the `list()` function, which takes an arbitrary number of arguments:

Example 1:

```
(x <- list(1, "a", TRUE, 1 + 4i))
```

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

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

Example 2:

```
(my.list <- list("one", TRUE, 3, c("f","o","u","r")))
```

```
[[1]]
[1] "one"

[[2]]
[1] TRUE
```

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

[[4]]
[1] "f" "o" "u" "r"
```

```r
my.list[[2]]
```

```
[1] TRUE
```

```r
mode(my.list[[2]])
```

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

```r
my.list[[4]][1]
```

```
[1] "f"
```

```r
my.list[4][1]
```

```
[[1]]
[1] "f" "o" "u" "r"
```

We can also create an **empty list** of a pre-specified length with the `vector()` function:

Example:

```r
(x <- vector(mode = "list", length = 5)) # the elements are NULL
```

```
[[1]]
NULL

[[2]]
NULL
```

```
[[3]]
NULL

[[4]]
NULL

[[5]]
NULL
```

To flatten a list x, that is **convert it to a vector**, we use `unlist(x)`.

Example:

```r
x <- list(1, c(2, 3), c(4, 5, 6))
unlist(x)
```

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

Many **functions** produce list objects as their output. For example, when we fit a least squares regression, the **regression object** itself is a list, and can be manipulated using list operations.

Example:

```r
lm.xy <- lm(y ~ x, data = data.frame(x = 1:5, y = 1:5))
mode(lm.xy)
```

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

```r
names(lm.xy)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"         
 [5] "fitted.values" "assign"        "qr"            "df.residual"  
 [9] "xlevels"       "call"          "terms"         "model"        
```

2.5.2 Names

The elements of a list can be named when the list is created, using arguments of the form `name1 = x1`, `name2 = x2`, etc., or they can be named later by assigning a value to the `names` attribute.

Example:

```
my.list <- list(first = "one", second = TRUE, third = 3,
                fourth = c("f","o","u","r"))
names(my.list)
```

```
[1] "first"  "second" "third"  "fourth"
```

Unlike a dataframe, the elements of a list do not have to be named. Names can be used (within quotes) when indexing with single or double square brackets, or they can be used (with or without quotes) after a dollar sign to extract a list element.

2.6 Arrays

Sometimes, you need to store multiple matrices or data frames into a single object; in this case, we can use **arrays** to store this data.

Data frames and matrices are of two dimensions only, but an **array** can be of any number of dimensions.

Here is a simple example to store three matrices of order 2 x 2 in a single array object:

Examples:

```
(mat.array <- array(dim=c(2,2,3)))
```

```
, , 1

     [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA

, , 2

     [,1] [,2]
```

```
[1,]   NA   NA
[2,]   NA   NA

, , 3

     [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA
```

```
(mat.array[,,1] <- rnorm(4))
```

```
[1] -0.11596596 -0.62803603  0.07458085 -0.21493634
```

```
(mat.array[,,1] <- rnorm(4))
```

```
[1] -2.6408501 -0.9905649  1.5851652 -0.4581713
```

```
(mat.array[,,2] <- rnorm(4))
```

```
[1]  0.3578089 -0.8048089  0.4317092  0.7714949
```

```
(mat.array[,,3] <- rnorm(4))
```

```
[1] -1.2335690 -1.1881291 -1.2988609  0.6183872
```

3 Packages

A **package** is an archive of files that conforms to a certain format and structure and that provides extra functionality, usually extending R in a particular direction. The R community has produced many high-quality R packages for performing specific tasks,

Any package is in one of **three states**:

- **Installed and loaded**. A package that is loaded is directly available to your R session. Find out which packages are loaded using `sessionInfo()`.
- **Installed but not loaded**. A package that is installed is available for loading but its contents are not available until it is loaded. The function `help.start()` gives details of the packages that are installed on your computer.
- **Not installed**. These packages cannot be loaded. If a package is **not installed** then the `library()` function produces an error. If the install status is uncertain at the time of calling library (for example if you are writing a function that requires the package), then use the `require()` function, which returns FALSE if the package is not installed, rather than an error.

Installing all available packages would be a waste of space and time, as you would never use most of them. Similarly, loading all installed packages every time you start R would take some time, so by default R only loads the **base packages** when it starts and requires the user to load any others as and when they are needed.

Packages are divided into **three groups**:

- **Base**. Base packages are installed along with R, and their objects are always available.
- **Recommended**. Recommended packages are installed along with R but must be loaded before they can be used.
- **Other**. Other packages are not installed by default, and must be installed separately.

The command to find out what packages are available for loading is `installed.packages`. The output of the function is quite verbose, but we only need the first column:

Example:

```r
installed.packages()[1:5, 1] # Returns only the first five packages
```

```
    askpass    backports    base64enc          bit        bit64
  "askpass" "backports" "base64enc"        "bit"      "bit64"
```

All the packages that are available at a repository, and whose requirements are matched by the currently running version of R, can be listed using the command `available.packages()`. A package that is available in the repository but has not yet been installed may be installed using the `install.packages()` function. If we include the argument `dependencies = TRUE`, then the function will also install packages that are necessary to run the package or packages of interest; such packages are called dependencies.

The status of the packages that are installed can be compared with the repository using the `old.packages()` function, and easily updated using the `update.packages()` function.

R can be easily updated with **installr package**. To update R on MacOS, you need to use **updateR" package** instead.

Example:

```r
install.packages("installr")
library(installr)
updateR()
```

4 Code Style

Even as a very new programmer, it's a good idea to work on your code style. Using a consistent style makes it easier for others (including future-you!) to read your work and is particularly important if you need to get help from someone else.

A summary set of recommendations:

- Always use **text files/text editor**.
- **Indent** your code. Each indent should be a minimum of 4 spaces, and ideally 8 spaces.
- **Limit the width** of your code. You can limit the width of your text editor so that the code you write doesn't fly off into the wilderness on the right hand side.
- **Limit the length** of individual functions. A function should not take up more than one page of your editor (of course, this depends on the size of your monitor).## Names

4.1 Names

Remember that variable names should use only lowercase letters, numbers, and `_`. Use `_`to separate words within a name.

As a general rule of thumb, it's better to prefer **long**, descriptive names that are easy to understand rather than concise names that are fast to type.

In general, if you have a bunch of variables that are a variation on a theme, you're better off giving them a common **prefix** rather than a common suffix because autocomplete works best on the start of a variable.

4.2 Spaces

Put **spaces** on either side of mathematical operators apart from ^ (i.e. `+`, `-`, `==`, `<`, …), and around the assignment operator (`<-`).

Example:

```
# Strive for
z <- (a + b)^2 / d

# Avoid
z<-( a + b ) ^ 2/d
```

Don't put spaces inside or outside parentheses for regular function calls. Always put a space after a comma, just like in standard English.

```
# Strive for
mean(x, na.rm = TRUE)

# Avoid
mean (x ,na.rm=TRUE)
```

4.3 Pipes

`|>` should always have a space before it and should typically be the last thing on a line. This makes it easier to add new steps, rearrange existing steps, modify elements within a step, and get a sky view by skimming the verbs on the left-hand side.

```
# Strive for
flights |>
  filter(!is.na(arr_delay), !is.na(tailnum)) |>
  count(dest)

# Avoid
flights|>filter(!is.na(arr_delay), !is.na(tailnum))|>count(dest)
```

If the function you're piping into has **named arguments** (like `mutate()` or `summarize()`), put each argument on a new line. If the function doesn't have named arguments (like `select()` or `filter()`), keep everything on one line unless it doesn't fit, in which case you should put each argument on its own line.

```
# Strive for
flights |>
  group_by(tailnum) |>
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

# Avoid
flights |>
  group_by(
    tailnum
  ) |>
  summarize(delay = mean(arr_delay, na.rm = TRUE), n = n())
```

After the first step of the pipeline, **indent** each line by **two spaces**. RStudio will automatically put the spaces in for you after a line break following a `|>` . If you're putting each argument on its own line, indent by an extra two spaces. Make sure `)`is on its own line, and un-indented to match the horizontal position of the function name.

```
# Strive for
flights |>
  group_by(tailnum) |>
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
```

Be wary of writing very **long pipes**, say longer than **10-15 lines**. Try to break them up into smaller sub-tasks, giving each task an informative name.

The same basic rules that apply to the pipe also apply to **ggplot2**; just treat `+` the same way as `|>`.

Part II

Part 1: Manipulating Data

5 Reading Data

R provides a number of ways to read data from a file, the **most flexible** of which is the `scan()` function. We use scan to read a vector of values from a file. It has the form:

Example:

```
scan(file = "", what = 0, n = -1, sep = "", skip = 0, quiet = FALSE)
```

Remember that `scan()` returns a **vector**.

5.1 Tabular Data

It is common for data to be arranged in **tables**, with columns corresponding to variables and rows corresponding to separate observations. These **dataframes** are usually read into R using the function `read.table()`, which has the form:

```
read.table(file, header = FALSE, sep = "")
# read.table() returns a dataframe
```

`read_table()` reads a common variation of fixed-width files where columns are separated by white space.

There are two commonly used **variants** of `read.table()`:

5.1.1 `read.csv()`

`read.csv()` is for comma-separated data and is equivalent to `read.table(file, header = TRUE, sep = ",")`.

Sometimes, it could happen that the file extension is **.csv**, but the data is **not comma separated**. In that case, we can still use the `read.csv()` function, but in this case we have to specify the separator:

Example:

```
read.csv("iris_semicolon.csv", stringsAsFactors = FALSE, sep=";")
anscombe_tab_2 <- read.table("anscombe.txt", header=TRUE)
```

Usually, `read_csv()` uses the **first line** of the data for the column names, which is a very common convention. But it's not uncommon for a few lines of metadata to be included at the top of the file. You can use `skip = n` to skip the first n lines or use `comment = "#"` to drop all lines that start with (e.g.) `#`.

Example:

```
read_csv("data/students.csv,
  skip = 2,
  comment = "#"
)
```

In other cases, the data might **not have column names**. You can use `col_names = FALSE` to tell `read_csv()` not to treat the first row as headings and instead label them sequentially from X_1 to X_n. Alternatively, you can pass `col_names` a character vector which will be used as the column names (e.g. `col_names = c("x", "y", "z")`):

Note that by default `read_csv()` only recognizes empty strings (" ") in as NAs, however you surely want it to also recognize the **character string "N/A"**.

Example:

```
students <- read_csv("data/students.csv", na = c("N/A", ""))
```

If a **.csv file** contains both numeric and character variables, and we use `read.csv()`, the character variables get automatically converted to the **factor** type. We can prevent character variables from this automatic conversion to factor, by specifying `stringsAsFactors=FALSE` within the `read.csv()` function.

Example:

```
read.csv("iris.csv", stringsAsFactors=F)
```

readr provides a total of nine column types for you to use: `col_logical()`, `col_logical()`, `col_factor()`... Find more information here.

5.1.2 `read.delim()`

`read_delim()` reads in files with any delimiter, attempting to automatically guess the delimiter if you don't specify it. For example, `read.delim()`is for tab-delimited data and is equivalent to `read.table(file, header = TRUE, sep = "\t")`.

Example:

```
## skips the first 2 lines
anscombe <- read.csv("CSVanscombe.csv", skip=2)
```

5.1.3 Other file types

Once you've mastered `read_csv()`, using **readr's** other functions is straightforward; it's just a matter of knowing which function to reach for:

- `read_csv2()`: reads semicolon-separated files. These use `;` instead of `,` to separate fields and are common in countries that use `,` as the decimal marker.
- `read_tsv()`: reads tab-delimited files.
- `read_fwf()`: reads fixed-width files. You can specify fields by their widths with `fwf_widths()` or by their positions with `fwf_positions()`.
- `read_log()`: reads Apache-style log files.

5.2 Reading Multiple Files

Sometimes your data is split across **multiple files** instead of being contained in a single file. For example, you might have sales data for multiple months, with each month's data in a separate file: `01-sales.csv` for January, `02-sales.csv` for February, and `03-sales.csv` for March.

With `read_csv()`you can read these data in at once and stack them on top of each other in a single data frame.

Example:

```
sales_files <- c("data/01-sales.csv", "data/02-sales.csv", "data/03-sales.csv")
read_csv(sales_files, id = "file")
```

The `id` argument adds a new column called file to the resulting data frame that identifies the file the data come from. This is especially helpful in circumstances where the files you're reading in do not have an identifying column that can help you trace the observations back to their original sources.

An alternative to read multiple data made **available online** is this:

Example:

```r
sales_files <- c(
  "https://pos.it/r4ds-01-sales",
  "https://pos.it/r4ds-02-sales",
  "https://pos.it/r4ds-03-sales"
)
read_csv(sales_files, id = "file")
```

5.3 Reading Line by Line

Text files can be **read line by line** using the `readLines()` function:

Example:

```r
con <- gzfile("words.gz")
x <- readLines(con, 10)
```

This approach is useful because it allows you to read from a file **without having to uncompress the file first**, which would be a waste of space and time.

To read in lines of **webpages** it can be useful the `readLines()` function:

Example:

```r
con <- url("http://www.jhsph.edu", "r") ## Read the web page
x <- readLines(con) ## Print out the first few lines head(x)
```

5.4 xls Format

If the dataset is stored in the ***.xls or* .xlsx format** you can use `read.xlsx2`:

Example:

```r
library(xlsx)
anscombe_xlsx <- read.xlsx2("xlsxanscombe.xlsx", sheetIndex=1)
```

5.5 RData Format

If you need to store more than one dataset in a single file we can use the ***.RData format**:

Example:

```
## example to load multiple datasets, and a vector of R objects from a single *.RData
load("robjects.RData")
## to check if objects have been loaded correctly
objects()
```

5.6 Import Stata/SPSS File

To import a **Stata/SPSS file** into R:

- First you need to call the foreign library:

  ```
  install.packages("foreign")
  ```

- Then use `read.data()`/`read.spss()`:

  ```
  data <- read.spss(file="data.spss", to.data.frame=TRUE)
  ```

- The output will always be a data frame:

  ```
  write.foreign(data, "mydata.txt", "mydata.sps", package="SPSS")
  ```

5.7 JSON File

To read a **JSON file**:

```
install.packages("rjson")
data <- fromJSON(file="data.json")
data2 <- as.data.frame(data)
```

5.8 `view()`

To view the data variable, you can use the `view()`.

5.9 Manual Data Entry

Sometimes you'll need to assemble a **tibble "by hand"** doing a little data entry in your R script.

There are **two useful functions** to help you do this which differ in whether you layout the tibble by columns or by rows.

- `tibble()` works by column:

  Example:

  ```
  tidyr::tibble(
    x = c(1, 2, 5),
    y = c("h", "m", "g"),
    z = c(0.08, 0.83, 0.60)
  )
  ```

  ```
  # A tibble: 3 x 3
        x y         z
    <dbl> <chr> <dbl>
  1     1 h      0.08
  2     2 m      0.83
  3     5 g      0.6
  ```

- `tribble()` (transposed tibble) lets you lay out your data row by row: column headings start with `~` and entries are separated by commas. This makes it possible to lay out small amounts of data in an easy to read form:

  Example:

  ```
  tidyr::tribble(
    ~x, ~y, ~z,
    1, "h", 0.08,
    2, "m", 0.83,
    5, "g", 0.60
  )
  ```

  ```
  # A tibble: 3 x 3
        x y         z
    <dbl> <chr> <dbl>
  1     1 h      0.08
  2     2 m      0.83
  3     5 g      0.6
  ```

6 Writing Data

R provides a number of commands for **writing** output to a file. You will generally use `write()` or `write.table()` for writing numeric values and `cat()` for writing text, or a combination of numeric and character values.

The command `write()` has the form:

```r
# `x` is the dataframe to save / `file` is the location to save it
write(x, file = "data", ncolumns = if(is.character(x)) 1 else 5, append = FALSE)
```

Example:

```r
write_csv(students, "students.csv")
```

Note that the **variable type** information is lost when you save to CSV. This makes CSVs a little **unreliable** for caching interim results, so you will need to recreate the column specification every time you load in.

Example:

```r
write_rds(students, "students.rds")
read_rds("students.rds")

# An alternative method
library(arrow)
write_parquet(students, "students.parquet")
read_parquet("students.parquet")
```

These store data in R's custom binary format called **RDS**. This means that when you reload the object, you are loading the exact same R object that you stored. **Parquet** tends to be much faster than **RDS** and is usable outside of R, but does require the **arrow package**.

6.1 Writing a Dataframe

To write a **dataframe** to a file we use this:

```
write.table(x, file = "", append = FALSE, sep = " ", row.names = TRUE, col.names = TRUE)
```

Here x is the vector to be written. If x is a **matrix or array** then it is converted to a vector (column by column) before being written. The other parameters are optional.

We can identify the **complete rows** from a two-dimensional object such as a dataframe (that is, rows that have no missing values) via the `complete.cases` command. We can easily remove rows with missing values using the `na.omit` function.

6.2 Writing Matrices

Because `write()` converts matrices to vectors before writing them, using it to write a matrix to a file can cause unexpected results. Since R stores its matrices by column, you should pass the **transpose of the matrix to write** if you want the output to reflect the matrix structure.

```
x <- matrix(1:24, nrow = 4, ncol = 6)
write(t(x), file = "../results/out.txt", ncolumns = 6)
```

6.3 cat()

A more flexible command for writing to a file is `cat()`, which has the form:

```
cat(..., file = "", sep = " ", append = FALSE)
```

Note that cat does not automatically write a newline after the expressions …. If you want a newline you must explicitly include the string `\n`.

6.4 Tabular Data

For **writing tabular data** to text files (i.e. CSV) or connections you may use `write.table()`, which has this format:

Example:

```
write.table()
```

6.5 `dump()`

There is also the very useful `dump()` function, which creates a text representation of almost any R object that can subsequently be read by source.

Example:

```r
x <- matrix(rep(1:5, 1:5), nrow = 3, ncol = 5)
dump("x", file = "../results/x.txt")
rm(x)
source("../results/x.txt")
x
```

7 Transforming Data

7.1 'head()andtail()'

After reading the file, you can use the `head()` and `tail()` functions to examine the object:

Example:

```
head(dataset)
tail()
```

7.2 Changing Variable Types

A common task after reading in data is to consider **variable types**. For example, if a variable in a dataset is a categorical variable with a known set of possible values but you see it is represented as a character, then you will want to make it a factor. Or a numeric variable is a character variable because one of the observations is typed out as a character instead of a numeric (e.g. "five" instead of 5).

Example:

```
students |>
  mutate(meal_plan = factor(meal_plan),
  age = parse_number(if_else(age == "five", "5", age))
```

After this, you will that the type of variable denoted underneath the variable name has changed from character () to factor ().

7.3 Changing Columns Names

Note that you can assign your **own column names** after reading a dataframe using the `names()` function, or when you read it in, using the `col.names` argument, which should be assigned a character vector the same length as the number of columns.

If there is no header and no `col.names` argument, then R uses the names `V1`, `V2`, etc.

7.4 Columns No-Syntatic Names

Sometimes, you might also notice that the columns names are surrounded by **backticks**. That's because they contain spaces, breaking R's usual rules for variable names; they're non-syntactic names.

~~~
*Example:*

``` R
students |>
  rename(
    student_id = `Student ID`,
    full_name = `Full Name`
  )
```
~~~

An alternative approach is to use `janitor::clean_names()` to use some heuristics to turn them all into snake case at once1.

```r
students |> janitor::clean_names()
```

7.5 Removing Duplicates

You can **remove duplicates** based on the `x` variable using:

```r
x <- c(1, 2, NA, 4, NA, 5)
data[!duplicated(data$x), ]
```

7.6 Reshaping Datasets

Statistical analysis sometimes requires wide data and sometimes long data. In such cases, we need to be able to fluently and fluidly **reshape** the data to meet the requirements of statistical analysis. The function `reshape()` reshapes a dataframe between 'wide' format (with repeated measurements in separate columns of the same row) and 'long' format (with the repeated measurements in separate rows).

Data reshaping is just a rearrangement of the form of the data—it does not change the content of the dataset.

Example:

```
students <- data.frame(sid=c(1,1,2,2), exmterm=c(1,2,1,2), math=c(50,65,75,69),
literature=c(40,45,55,59), language=c(70,80,75,78))

# Reshaping dataset using reshape function to wide format
wide_students <- reshape(students, direction="wide", idvar="sid", timevar="exmterm")

# Now again reshape to long format
long_students <- reshape (wide_students, direction="long", idvar="id")
```

7.6.1 The reshape package

- **Melting data (molten data)**:
  - Though melting can be applied to different R objects, the most common use is to melt a data frame.
  - To perform melting operations using the melt function, we need to know what the identification variables and measured variables in the original input dataset are.
  - One important thing to note is that, whenever we use the melt function, all the measured variables should be of the same type, that is, the measured variables should be either numeric, factor, character, or date.
  - To deal with the implicit missing value, it is good to use `na.rm = TRUE` with the melt function to remove the structural missing value (i.e., it will fill empty cells in the data table with NA).

  Example:

  ```
  ## the format of the resulting table is id/variable/value
  melt(students, id=c("sid","exmterm"), measured=c("math", "literature", "language"))
  ```

- **Casting molten data**:
  - Once we have molten data, we can rearrange it in any layout using the cast function from the reshape package.
  - There are two main arguments required to cast molten data: data and formula.
  - The basic casting formula is col_var_1+col_var_2 ~ row_var_1+ row_var_2, which describes the variables to appear in columns and rows.

  Example:

```r
# to return to the original data structure
cast(molten_students, sid+exmterm ~ variable)
```

- For faster and large data rearrangement, use the **reshape2 package** and the functions `dcast` (data frames) and `acast` (arrays):

  Example:

  ```r
  acast(molten_students, sid+exmterm~variable)
  ```

7.7 Missing Values

R represents missing observations through the data value **NA**. It is easiest to think of **NA** values as place holders for data that should have been there, but, for some reason, are not.

7.7.1 Detect NA

We can **detect** whether variables are missing value using:

- `is.na()` is used to test objects if they are NA.
- `is.nan()` is used to test for NaN.

Example:

```r
x <- c(1, 2, NaN, NA, 4)
is.na(x)
```

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

```r
is.nan(x)
```

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

To check if there is any **NA** in a dataframe:

Example:

```r
any(is.na(x))
```

```
[1] TRUE
```

7.7.2 Remove NAs

Example:

```
a <- c(11, NA, 13)
mean(a, na.rm = TRUE)
```

```
[1] 12
```

7.7.3 Extract NA from a vector

Example:

```
x <- c(1, 2, NA, 3)
z <- is.na(x)
x[!z]
```

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

7.7.4 NA vs NULL

Note that **NA** and **NULL** are not equivalent:

- **NA** is a place holder for something that exists but is missing.
- **NULL** stands for something that never existed at all.

7.7.5 Removing NA values

You can remove rows with **NA** values in any variables:

```
na.omit(data)
```

```
## Another example
x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x)
x[!bad]
```

```
[1] 1 2 4 5
```

How can you take the **subset with no missing values** in any of those objects?:

Example:

```
good <- complete.cases(airquality)
head(airquality[good,])
```

8 Tidy Data

The **tidyverse** is a set of packages that work in harmony because they share common data representations and API design. The tidyverse package is designed to make it easy to install and load core packages from the tidyverse in a single command.

If you'd like to learn how to use the tidyverse effectively, the best place to start is R for Data Science (2e).

To check that you're using the **latest version** of each package the easiest way is to run `tidyverse_update()`.

8.1 Installation

- Install all the packages in the tidyverse by running:

  ```
  install.packages("tidyverse")
  ```

- Run `library(tidyverse)` to load the core tidyverse and make it available in your current R session.

  Note the conflicts message that's printed when you load the tidyverse. It tells you that dplyr overwrites some functions in base R. If you want to use the base version of these functions after loading dplyr, you'll need to use their full names: `stats::filter()` and `stats::lag()`.

- Learn more about the tidyverse package https://tidyverse.tidyverse.org

8.2 Core Packages

`library(tidyverse)` will load the **core tidyverse** packages:

- **ggplot2**, for data visualisation, more info.
- **dplyr**, for data manipulation, more info.
- **tidyr**, for data tidying, more info
- **readr**, for data import, more info

- **purrr**, for functional programming, more info.
- **tibble**, for tibbles, a modern re-imagining of data frames, more info.
- **stringr**, for strings, more info
- **forcats**, for factors, [more info] (https://forcats.tidyverse.org/)
- **lubridate**, for date/times.

8.3 Functionalities

8.3.1 Import

As well as **readr**, for reading flat files, the tidyverse package installs a number of other packages for reading data:

- **DBI** for relational databases. You'll need to pair DBI with a database specific backends like RSQLite, RMariaDB, RPostgres, or odbc. More info here.
- **haven** for SPSS, Stata, and SAS data.
- **httr** for web APIs.
- **readxl** for .xls and .xlsx sheets.
- **googlesheets4** for Google Sheets via the Sheets API v4.
- **googledrive** for Google Drive files.
- **rvest** for web scraping.
- **jsonlite** for JSON. (Maintained by Jeroen Ooms.)
- **xml2** for XML.

8.3.2 Wrangle

In addition to **tidyr**, and **dplyr**, there are five packages (including **stringr** and **forcats**) which are designed to work with specific types of data:

- **lubridate** for dates and date-times.
- **hms** for time-of-day values.
- **blob** for storing blob (binary) data.

There are also two packages that allow you to interface with different backends using the same dplyr syntax:

- **dbplyr** allows you to use remote database tables by converting dplyr code into SQL.
- **dtplyr** provides a data.table backend by automatically translating to the equivalent, but usually much faster, data.table code. Program

8.3.3 Programming

In addition to **purrr**, which provides very consistent and natural methods for iterating on R objects, there are two additional tidyverse packages that help with general programming challenges:

- **magrittr** provides the pipe, `%>%` used throughout the tidyverse. It also provide a number of more specialised piping operators (like `%$%` and `%<>%`) that can be useful in other places.
- **glue** provides an alternative to `paste()` that makes it easier to combine data and strings.

8.3.4 Modeling

Modeling with the tidyverse uses the collection of **tidymodels packages**, which largely replace the modelr package used in R4DS.

Visit the Getting Started Guide for more detailed examples, or go straight to the Learn page.

8.4 Tidying Data

We'll focus on **tidyr**, a package that provides a bunch of tools to help tidy up messy datasets. **tidyr** is a member of the core tidyverse.

```
library(tidyverse)
```

There are two main **advantages** of tidy data:

- There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.
- There's a specific advantage to placing variables in columns because it allows R's vectorized nature to shine.

8.4.1 Pivot Data

You'll begin by figuring out what the underlying **variables and observations** are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. Next, you'll **pivot** your data into a tidy form, with variables in the columns and observations in the rows.

tidyr provides two functions for pivoting data: `pivot_longer()` and `pivot_wider()`. We'll first start with pivot_longer() because it's the most common case. Let's dive into some examples.

8.4.2 Lengthening Data

Lengthening data means increasing the number of rows and decreasing the number of columns. The inverse transformation is `pivot_wider()`.

In this dataset, each observation is a song. The first three columns (`artist`, `track` and `date.entered`) are variables that describe the song. Then we have 76 columns (`wk1-wk76`) that describe the rank of the song in each week1. Here, the column names are one variable (the `week`) and the cell values are another (the `rank`).

```r
tidyr::billboard
```

```
# A tibble: 317 x 79
   artist     track date.entered   wk1   wk2   wk3   wk4   wk5   wk6   wk7   wk8
   <chr>      <chr> <date>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 2 Pac      Baby~ 2000-02-26      87    82    72    77    87    94    99    NA
 2 2Ge+her    The ~ 2000-09-02      91    87    92    NA    NA    NA    NA    NA
 3 3 Doors D~ Kryp~ 2000-04-08      81    70    68    67    66    57    54    53
 4 3 Doors D~ Loser 2000-10-21      76    76    72    69    67    65    55    59
 5 504 Boyz   Wobb~ 2000-04-15      57    34    25    17    17    31    36    49
 6 98^0       Give~ 2000-08-19      51    39    34    26    26    19     2     2
 7 A*Teens    Danc~ 2000-07-08      97    97    96    95   100    NA    NA    NA
 8 Aaliyah    I Do~ 2000-01-29      84    62    51    41    38    35    35    38
 9 Aaliyah    Try ~ 2000-03-18      59    53    38    28    21    18    16    14
10 Adams, Yo~ Open~ 2000-08-26      76    76    74    69    68    67    61    58
# i 307 more rows
# i 68 more variables: wk9 <dbl>, wk10 <dbl>, wk11 <dbl>, wk12 <dbl>,
#   wk13 <dbl>, wk14 <dbl>, wk15 <dbl>, wk16 <dbl>, wk17 <dbl>, wk18 <dbl>,
#   wk19 <dbl>, wk20 <dbl>, wk21 <dbl>, wk22 <dbl>, wk23 <dbl>, wk24 <dbl>,
#   wk25 <dbl>, wk26 <dbl>, wk27 <dbl>, wk28 <dbl>, wk29 <dbl>, wk30 <dbl>,
#   wk31 <dbl>, wk32 <dbl>, wk33 <dbl>, wk34 <dbl>, wk35 <dbl>, wk36 <dbl>,
#   wk37 <dbl>, wk38 <dbl>, wk39 <dbl>, wk40 <dbl>, wk41 <dbl>, wk42 <dbl>, ...
```

To tidy this data, we'll use `pivot_longer()`:

```r
tidyr::billboard |>
  tidyr::pivot_longer(
    cols = starts_with("wk"),
    names_to = "week",
    values_to = "rank",
    values_drop_na = TRUE # to get rid of NAs
  )
```

```
# A tibble: 5,307 x 5
   artist  track                   date.entered week   rank
   <chr>   <chr>                   <date>       <chr> <dbl>
 1 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk1      87
 2 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk2      82
 3 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk3      72
 4 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk4      77
 5 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk5      87
 6 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk6      94
 7 2 Pac   Baby Don't Cry (Keep... 2000-02-26   wk7      99
 8 2Ge+her The Hardest Part Of ... 2000-09-02   wk1      91
 9 2Ge+her The Hardest Part Of ... 2000-09-02   wk2      87
10 2Ge+her The Hardest Part Of ... 2000-09-02   wk3      92
# i 5,297 more rows
```

This data is now tidy, but we could make future computation a bit easier by converting values of week from character strings to numbers using `mutate()` and `readr::parse_number()`. `parse_number()` is a handy function that will extract the first number from a string, ignoring all other text.

```
billboard_longer <- tidyr::billboard |>
  tidyr::pivot_longer(
    cols = starts_with("wk"),
    names_to = "week",
    values_to = "rank",
    values_drop_na = TRUE
  ) |>
  dplyr::mutate(
    week = readr::parse_number(week)
  )
billboard_longer
```

```
# A tibble: 5,307 x 5
   artist  track                   date.entered  week  rank
   <chr>   <chr>                   <date>       <dbl> <dbl>
```

```
 1 2 Pac   Baby Don't Cry (Keep... 2000-02-26       1    87
 2 2 Pac   Baby Don't Cry (Keep... 2000-02-26       2    82
 3 2 Pac   Baby Don't Cry (Keep... 2000-02-26       3    72
 4 2 Pac   Baby Don't Cry (Keep... 2000-02-26       4    77
 5 2 Pac   Baby Don't Cry (Keep... 2000-02-26       5    87
 6 2 Pac   Baby Don't Cry (Keep... 2000-02-26       6    94
 7 2 Pac   Baby Don't Cry (Keep... 2000-02-26       7    99
 8 2Ge+her The Hardest Part Of ... 2000-09-02       1    91
 9 2Ge+her The Hardest Part Of ... 2000-09-02       2    87
10 2Ge+her The Hardest Part Of ... 2000-09-02       3    92
# i 5,297 more rows
```

Other cases on how to deal with lenghtening data can be found here.

8.4.3 Widening data

`pivot_wider()` makes datasets wider by increasing columns and reducing rows and helps when one observation is spread across multiple rows.

Example:

```
tidyr::cms_patient_experience
```

```
# A tibble: 500 x 5
   org_pac_id org_nm                             measure_cd measure_title prf_rate
   <chr>      <chr>                              <chr>      <chr>            <dbl>
 1 0446157747 USC CARE MEDICAL GROUP INC         CAHPS_GRP~ CAHPS for MI~       63
 2 0446157747 USC CARE MEDICAL GROUP INC         CAHPS_GRP~ CAHPS for MI~       87
 3 0446157747 USC CARE MEDICAL GROUP INC         CAHPS_GRP~ CAHPS for MI~       86
 4 0446157747 USC CARE MEDICAL GROUP INC         CAHPS_GRP~ CAHPS for MI~       57
 5 0446157747 USC CARE MEDICAL GROUP INC         CAHPS_GRP~ CAHPS for MI~       85
 6 0446157747 USC CARE MEDICAL GROUP INC         CAHPS_GRP~ CAHPS for MI~       24
 7 0446162697 ASSOCIATION OF UNIVERSITY PHYSI~ CAHPS_GRP~ CAHPS for MI~       59
 8 0446162697 ASSOCIATION OF UNIVERSITY PHYSI~ CAHPS_GRP~ CAHPS for MI~       85
 9 0446162697 ASSOCIATION OF UNIVERSITY PHYSI~ CAHPS_GRP~ CAHPS for MI~       83
10 0446162697 ASSOCIATION OF UNIVERSITY PHYSI~ CAHPS_GRP~ CAHPS for MI~       63
# i 490 more rows
```

The core unit being studied is an organization, but each organization is spread across six rows, with one row for each measurement taken in the survey organization. We can see the complete set of values for `measure_cd` and `measure_title` by using `distinct()`:

```
tidyr::cms_patient_experience |>
  dplyr::distinct(measure_cd, measure_title)
```

```
# A tibble: 6 x 2
  measure_cd   measure_title
  <chr>        <chr>
1 CAHPS_GRP_1  CAHPS for MIPS SSM: Getting Timely Care, Appointments, and Infor~
2 CAHPS_GRP_2  CAHPS for MIPS SSM: How Well Providers Communicate
3 CAHPS_GRP_3  CAHPS for MIPS SSM: Patient's Rating of Provider
4 CAHPS_GRP_5  CAHPS for MIPS SSM: Health Promotion and Education
5 CAHPS_GRP_8  CAHPS for MIPS SSM: Courteous and Helpful Office Staff
6 CAHPS_GRP_12 CAHPS for MIPS SSM: Stewardship of Patient Resources
```

We'll use `measure_cd` as the source for our new column names for now. Instead of choosing new column names, we need to provide the existing columns that define the values (`values_from`) and the column name (`names_from`):

```
tidyr::cms_patient_experience |>
  tidyr::pivot_wider(
    names_from = measure_cd,
    values_from = prf_rate
  )
```

```
# A tibble: 500 x 9
   org_pac_id org_nm           measure_title CAHPS_GRP_1 CAHPS_GRP_2 CAHPS_GRP_3
   <chr>      <chr>            <chr>               <dbl>       <dbl>       <dbl>
 1 0446157747 USC CARE MEDICA~ CAHPS for MI~          63          NA          NA
 2 0446157747 USC CARE MEDICA~ CAHPS for MI~          NA          87          NA
 3 0446157747 USC CARE MEDICA~ CAHPS for MI~          NA          NA          86
 4 0446157747 USC CARE MEDICA~ CAHPS for MI~          NA          NA          NA
 5 0446157747 USC CARE MEDICA~ CAHPS for MI~          NA          NA          NA
 6 0446157747 USC CARE MEDICA~ CAHPS for MI~          NA          NA          NA
 7 0446162697 ASSOCIATION OF ~ CAHPS for MI~          59          NA          NA
 8 0446162697 ASSOCIATION OF ~ CAHPS for MI~          NA          85          NA
 9 0446162697 ASSOCIATION OF ~ CAHPS for MI~          NA          NA          83
10 0446162697 ASSOCIATION OF ~ CAHPS for MI~          NA          NA          NA
# i 490 more rows
# i 3 more variables: CAHPS_GRP_5 <dbl>, CAHPS_GRP_8 <dbl>, CAHPS_GRP_12 <dbl>
```

The output doesn't look quite right; we still seem to have multiple rows for each organization.

That's because, we also need to tell `pivot_wider()` which column or columns have values that uniquely identify each row; in this case those are the variables starting with "org":

```
tidyr::cms_patient_experience |>
  tidyr::pivot_wider(
    id_cols = starts_with("org"),
    names_from = measure_cd,
    values_from = prf_rate
  )
```

```
# A tibble: 95 x 8
   org_pac_id org_nm CAHPS_GRP_1 CAHPS_GRP_2 CAHPS_GRP_3 CAHPS_GRP_5 CAHPS_GRP_8
   <chr>      <chr>        <dbl>       <dbl>       <dbl>       <dbl>       <dbl>
 1 0446157747 USC C~          63          87          86          57          85
 2 0446162697 ASSOC~          59          85          83          63          88
 3 0547164295 BEAVE~          49          NA          75          44          73
 4 0749333730 CAPE ~          67          84          85          65          82
 5 0840104360 ALLIA~          66          87          87          64          87
 6 0840109864 REX H~          73          87          84          67          91
 7 0840513552 SCL H~          58          83          76          58          78
 8 0941545784 GRITM~          46          86          81          54          NA
 9 1052612785 COMMU~          65          84          80          58          87
10 1254237779 OUR L~          61          NA          NA          65          NA
# i 85 more rows
# i 1 more variable: CAHPS_GRP_12 <dbl>
```

9 Subsetting R Objects

9.1 [operator

The `[` **operator** always returns an object of the same class as the original:

```
x <- c("a", "b", "c", "c", "d", "a")
# Extract the first element
x[1]
```

```
[1] "a"
```

```
# other examples
x[1:4]
```

```
[1] "a" "b" "c" "c"
```

```
x[c(1, 3, 49)]
```

```
[1] "a" "c" NA
```

```
x <- 100:110
i <- c(1, 3 ,2)
x[i]
```

```
[1] 100 102 101
```

```
u <- x > "a"
x[u] ## or x[x > "a"]
```

```
integer(0)
```

- Subsetting a **matrix**:

```r
x[1,2] ## Extract the first row, first column
x[1, ] ## Extract the first row
x[, 2] ## Extract the second column
x[1, 2, drop = FALSE] ## keeps the matrix format
```

- It can be used to select **multiple elements** of an object (also in **lists** or **dataframes**):

```r
(x <- list(foo = 1:4, bar = 0.6, baz = "hello"))
```

```
$foo
[1] 1 2 3 4

$bar
[1] 0.6

$baz
[1] "hello"
```

```r
x[c(1, 3)]
```

```
$foo
[1] 1 2 3 4

$baz
[1] "hello"
```

```r
x["foo"]
```

```
$foo
[1] 1 2 3 4
```

```r
class(x["foo"])
```

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

```r
x$foo[1]
```

```
[1] 1
```

```r
x$baz[1]
```

```
[1] "hello"
```

```r
## use of negative subscript removes first element "3"
num10[-1]
```

9.2 [[operator

The [[operator is used to extract elements of a **list or a dataframe**. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.

- **Subsetting a Dataframe**:

  ```r
  # Use the notation [[ ]] to extract columns
  x[["var1"]] = x[, 1] = x$var1 # All are equivalent
  ```

- **Subsetting a List**:

  ```r
  x[[1]] ## Extract single element from a list
  x[["bar"]] ## Extract named index
  x$bar ## Extract named index
  x[[c(1, 3)]] ## Get the 3rd element of the 1st element of the list
  x[[1]][[3]] ## Get the 3rd element of the 1st element of the list
  ```

- Now if we want to get access to the **individual elements** of `list_obj[[2]]`, we have to use the following command:

  ```r
  (data_2variable <- data.frame(x1=c(2,3,4,5,6), x2=c(5,6,7,8,1)))
  ```

  ```
    x1 x2
  1  2  5
  2  3  6
  3  4  7
  4  5  8
  5  6  1
  ```

  ```r
  (list_obj <- list(dat=data_2variable, vec.obj=c(1,2,3)))
  ```

  ```
  $dat
    x1 x2
  1  2  5
  2  3  6
  ```

```
3  4  7
4  5  8
5  6  1

$vec.obj
[1] 1 2 3
```

```
list_obj[[2]][1]
```

```
[1] 1
```

9.3 $ operator

The **$ operator** is used to extract elements of a **list or data frame by literal name**. Its semantics are similar to that of `[[`.

Example:

```
x[["bar"]] ## Extract named index
```

```
[1] 0.6
```

```
x$bar ## Extract named index
```

```
[1] 0.6
```

10 Plyr & Dplyr

The **plyr package** works on **every type** of data structure, whereas the **dplyr package** is designed to work only on **dataframes**.

10.1 Plyr

The most important utility of the **plyr package** is that a single line of code can perform all the `split()`, `apply()`, and `combine()` steps.

The steps for the **split-apply-combine** approach of data analysis are as follows:

1. First, we split the dataset into some mutually exclusive groups.
2. We then apply a task on each group and combine all the results to get the desired output.
3. This group-wise task could be generating new variables, summarizing existing variables, or even performing regression analysis on each group.
4. Finally, combining approaches helps us get a nice output to compare the results from different groups.

Example:

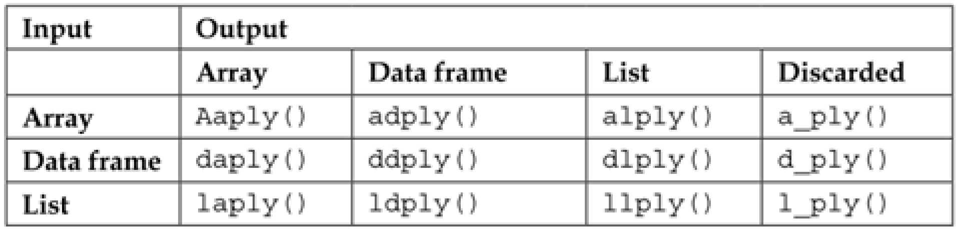
```r
library(plyr)
ddply(iris, .(Species), function(x) colMeans(x[-5]))
```

```
     Species Sepal.Length Sepal.Width Petal.Length Petal.Width
1     setosa        5.006       3.428        1.462       0.246
2 versicolor        5.936       2.770        4.260       1.326
3  virginica        6.588       2.974        5.552       2.026
```

- The first argument is the name of the data frame. We put iris, since the iris dataset is in the data frame structure, and we want to work on it.
- The second argument is for a variable or variables, according to which we want to split our data frame. In this case, we have Species.

- The third argument is a function that defines what kind of task we want to perform on each subset.

Note that the first letter of the function name specifies the input, and the second letter specifies the output type:



Input	Output			
	Array	Data frame	List	Discarded
Array	Aaply()	adply()	alply()	a_ply()
Data frame	daply()	ddply()	dlply()	d_ply()
List	laply()	ldply()	llply()	l_ply()

Figure 10.1: Table 1: Types of funcctions in the plyr package

Note: `mapply()` can take multiple inputs as separate arguments, whereas `a*ply()` takes only a single array argument.

10.2 Dplyr

Quite often, in real-life situations, we start our analysis with a **dataframe-type structure**. What do we do after getting a dataset and what are the basic data-manipulation tasks we usually perform before starting modeling?:

1. Check the validity of a dataset based on conditions.
2. Sort the dataset based on some variables, in ascending or descending order.
3. Create new variables based on existing variables.
4. Finally, summarize them.

dplyr can work with other data frame "backends" such as **SQL databases**. In fact, there is an SQL interface for relational databases via the DBI package

dplyr can also be integrated with the **data.table package** for large fast tables.

10.2.1 dplyr Grammar

All **dplyr functions** have a few common characteristics:

1. The first argument is a dataframe.

2. The subsequent arguments typically describe which columns to operate on using the variable names (without quotes).
3. The output is a new datafram (dplyr doesn't modify the existing original dataset because dplyr functions never modify their inputs).

dplyr's verbs are organized into four groups based on what they operate on: **rows, columns, groups, or tables**.

Table 10.1: Table 2. Dplyr functions

Columns	Rows
`select()`	`filter()` (base: subset)
`rename()`	`arrange()`
`mutate()`	`distinct()`
`relocate()`	`slice()`

Because each **verb** does one thing well, solving complex problems will usually require **combining multiple verbs**, and we'll do so with the **pipe**, `|>`.

For verbs to work at all, **dataframes** must be properly formatted and annotated. In particular, the data must be **tidy**, that is, the data comply with the following three interrelated rules:

- Each **variable** is a column; each column is a variable.
- Each **observation** is a row; each row is an observation.
- Each **value** is a cell; each cell is a single value.

Note that there's a specific **advantage** to placing variables in columns because it allows R's vectorized nature to shine. **dplyr**, **ggplot2**, and all the other packages in the tidyverse are designed to work with tidy data.

10.2.2 The Pipe

The real power of dplyr arises when you start to combine **multiple verbs**. For example, imagine that you wanted to find the fastest flights to Houston's IAH airport: you need to combine `filter()`, `mutate()`, `select()`, and `arrange()`:

The **pipe** takes the thing on its left and passes it along to the function on its right so that `x |> f(y)` is equivalent to f(x, y), and `x |> f(y) |> g(z)` is equivalent to g(f(x, y), z). The easiest way to pronounce the pipe is **"then"**.

Example:

```
flights |>
  filter(dest == "IAH") |>
  mutate(speed = distance / air_time * 60) |>
  select(year:day, dep_time, carrier, flight, speed) |>
  arrange(desc(speed))
```

Even though this pipeline has four steps, it's easy to skim because the verbs come at the start of each line: start with the flights data, then filter, then mutate, then select, then arrange.

IMPORTANT: Chaining (`|>`) is a powerful feature of **dplyr** that allows the output from one verb to be piped into the input of another verb using a short, easy-to-read syntax. To **add the pipe** to your code, we recommend using the built-in keyboard shortcut Ctrl/Cmd + Shift + M. You'll need to make one change to your RStudio options to use `|>` instead of `%>%`.

10.2.3 filter()

The `filter()` function is used to extract **subsets of rows** from a dataframe. This function is similar to the existing `subset()`.

Example:

```
chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80)
summary(chic.f$pm25tmean2)
```

The **tidyverse** alternative writing:

```
# Flights with a departure dely higher than 120 mins
flights |>
  filter(dep_delay > 120)
```

Sometimes, it is more important to subset the dataframe based on **values** of a variable or **multiple** variables.

Example:

```
filter(iris, Species=="virginica")
filter(iris, Species=="virginica" & Sepal.Length <6 & Sepal. Width <=2.7)
```

Tidyverse:

```
# Flights that departed on January 1
flights |>
  filter(month == 1 & day == 1)
# Flights that departed in January or February
flights |>
  filter(month == 1 | month == 2)
```

There's a useful shortcut when you're combining `|` and `==`: `%in%`. It keeps rows where the variable equals one of the values on the right:

```
flights |>
  filter(month %in% c(1, 2))
```

10.2.4 arrange()

The `arrange()` function is used to **change the order of the rows** of a dataframe according to the value of the variables/columns (while preserving corresponding order of other columns).

Example 1:

```
chicago <- arrange(chicago, date)
## Columns can be arranged in descending order
chicago <- arrange(chicago, desc(date))
arrange(iris, Sepal.Length, desc(Sepal.Width))
```

It takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide **more than one** column name, each additional column will be used to break ties in the values of the preceding columns.

Tidyverse:

```
# Sorts by the departure time, which is spread over four columns
# We get the earliest years first, then within a year, the earliest months, etc.
flights |>
  arrange(year, month, day, dep_time)
# Use desc() to re-order the df based on that column in big-to-small order
flights |>
  arrange(desc(dep_delay))
```

Note that the **number of rows has not changed** – we're only arranging the data, we're not filtering it.

10.2.5 distinct()

Sometimes, we might encounter duplicate observations in a data frame. The `distinct()` function helps eliminates these observations (finds all the unique rows in a dataset).

Example:

```
distinct(iris, Species, Petal.Width)
```

Tidyverse:

```
flights |>
  distinct()
```

Most of the time, however, you'll want the distinct combination of some variables, so you can also optionally **supply column names**:

```
flights |>
  distinct(origin, dest)

# Keep other columns when filtering for unique rows
flights |>
  distinct(origin, dest, .keep_all = TRUE)
```

If you want to find the **number of occurrences** instead, you're better off swapping `distinct()` for `count()`. With the `sort = TRUE` argument, you can arrange them in descending order of the number of occurrences.

Example:

```
flights |>
  count(origin, dest, sort = TRUE)
```

10.2.6 slice()

You can extract the subset of a dataframe using the `slice()` function.

Example:

```
slice(iris, 95:105)
```

There are **five handy functions** that allow you to extract specific rows within each group:

```
# Takes the first row from each group.
df |> slice_head(n = 1)
# Takes the last row in each group
df |> slice_tail(n = 1) .
# Takes the row with the smallest value of column x
df |> slice_min(x, n = 1)
# Takes the row with the largest value of column x
df |> slice_max(x, n = 1)
# Takes one random row
df |> slice_sample(n = 1)
```

You can vary `n` to select more than one row, or instead of `n =`, you can use `prop = 0.1` to select (e.g.) 10% of the rows in each group. For example, the following code finds the flights that are most delayed upon arrival at each destination:

Example:

```
flights |>
  group_by(dest) |>
  slice_max(arr_delay, n = 1) |>
  relocate(dest)
```

10.2.7 select()

Most of the time, you do not work on all the variables in a dataframe. Selecting a few columns could make the analysis process less complicated. The `select()` function can be used to select columns of a data frame that you want to focus on.

Example:

```
chicago <- readRDS("chicago.rds")
names(chicago)[1:3]
select(chicago, c("city", "tmpd"))
select(chicago, c(1, 3))
subset <- select(chicago, city:dptp)
```

Tidyverse:

- Select columns by name:

  ```
  flights |>
    select(year, month, day)
  ```

- Select all columns between year and day (inclusive):

  ```
  flights |>
    select(year:day)
  ```

- Select all columns except those from year to day (inclusive):

  ```
  flights |>
    select(!year:day)
  ```

- Select all columns that are characters:

  ```
  flights |>
    select(where(is.character))
  ```

There are a number of **helper functions** you can use within `select()`:

- `starts_with("abc")`: matches names that begin with "abc".
- `ends_with("xyz")`: matches names that end with "xyz".
- `contains("ijk")`: matches names that contain "ijk".
- `num_range("x", 1:3)`: matches x1, x2 and x3.

You can **rename variables** as you `select()` them by using `=`. The new name appears on the left-hand side of the `=`, and the old variable appears on the right-hand side

Example:

```
flights |>
  select(tail_num = tailnum)
```

You can also **omit variables** using the `select()` function by using the negative sign:

Example:

```
select(chicago, -(city:dptp))
```

10.2.8 rename()

Renaming a variable in a dataframe in R is surprisingly hard to do! The `rename()` function is designed to make this process easier.

Example:

```
chicago <- rename(chicago, dewpoint = dptp, pm25 = pm25tmean2)
```

The syntax inside the `rename()` function is to have the **new name** on the left-hand side of the `=` sign and the **old name** on the right-hand side.

Tidyverse

```
flights |>
  rename(tail_num = tailnum)
```

10.2.9 mutate()

The `mutate()` function exists to compute transformations of variables in a dataframe. Very often, you want to create new variables that are derived from existing variables and `mutate()` provides a clean interface for doing that (it adds new columns that are calculated from the existing columns).

Example:

```
# Create a pm25detrend variable that subtracts the mean from the pm25 variable
chicago <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE))
# Other example
mutate(iris, SLm=Sepal.Length/100, SWm= Sepal.Width/100, PLm=Petal. Length/100, PWm= Petal
```

Tidyverse:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60
  )
```

You can use the `.before` argument to add the variables to the **left-hand side**.

Example:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .before = 1
  )
```

You can also use `.after` to add after a variable, and in both `.before` and `.after` you can use the variable name instead of a position.

Note that since we haven't assigned the result of the above computation back to `flights`, the new variables `gain`, and `speed` will only be printed but will not be stored in a dataframe.

Alternatively, you can control which variables are kept with the `.keep` argument. A particularly useful argument is `"used"` which specifies that we only keep the columns that were used in the "create" step with `mutate()`. For example, the following output will contain only the variables dep_delay, arr_delay, air_time, gain, hours, and gain_per_hour.

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours,
    .keep = "used"
  )
```

If you want to keep only the new variables and **drop the old ones**, we could use the `transmute()` function:

Example:

```
## Here we detrend the PM10 and ozone (O3) variables
transmute(chicago, pm10detrend = pm10tmean2 - mean(pm10tmean2, na.rm = TRUE),
    o3detrend = o3tmean2 - mean(o3tmean2, na.rm = TRUE)))
## Note that there are only two columns in the transmuted data frame
```

10.2.10 relocate()

Use `relocate()` to move variables around. You might want to collect related variables together or move important variables to the front. By default `relocate()` moves variables to the front.

Tidyverse

```
flights |>
  relocate(time_hour, air_time)
```

You can also specify where to put them using the `.before` and `.after` arguments, just like in `mutate()`:

```
flights |>
  relocate(year:dep_time, .after = time_hour)
flights |>
  relocate(starts_with("arr"), .before = dep_time)
```

10.2.11 group_by()

The `group_by()` function is used to divide your dataset into groups meaningful for your analysis. You will usually use the `group_by()` function in conjunction with the `summarize()` function.

Example 1:

```
flights |>
  group_by(month)
```

You can create groups using **more than one variable**. For example, we could make a group for each date.

Example:

```
daily <- flights |>
  group_by(year, month, day)
```

Note that `group_by()` doesn't change the data but, if you look closely at the output, you'll notice that the output indicates that it is "grouped by" month. This means subsequent operations will now work "by month". `group_by()` adds this grouped feature (referred to as class) to the dataframe, which changes the behavior of the subsequent verbs applied to the data.

You might also want to **remove grouping** from a dataframe without using summarize(). You can do this with ungroup().

Example:

```
daily |>
  ungroup()
```

10.2.12 summarize()

The `summarize()` function is used to calculate a single summary statistic, and reduces the data frame to have a single row for each group.

Example:

```
flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay)
  )
```

You can create any number of **summaries in a single call** to `summarize()`. One very useful summary is `n()`, which returns the number of rows in each group.

Example:

```
flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = TRUE),
    n = n()
  )
```

10.2.13 Chaining (%>%)

Sometimes, it could be necessary to use multiple functions to perform a single task. The pipeline operator %>% is very handy for stringing together multiple dplyr functions in a sequence of operations.

This nesting is not a natural way to think about a sequence of operations:

```
third(second(first(x)))
```

The %>% operator allows you to string operations in a left-to-right fashion:

```
first(x) %>% second(x) %>% third(x)
```

This way we don't have to create a set of temporary variables along the way or create a massive nested sequence of function calls.

Once you travel down the pipeline with %>%, the first argument is taken to be the output of the previous element in the pipeline.

11 Date and Text Manipulation

11.1 Date Manipulation

In R, **dates** are stored as the number of days elapsed since **January 1, 1970**. The built-in R function `as.Date()` can handle only dates but not time. So if we convert any date object to its internal number, it will show the number of days.

We can reformat the number into a date using the date class.

```r
as.numeric(as.Date("1970-01-01")) ## We can also create a date object with other formats
```

```
[1] 0
```

```r
as.numeric(as.Date("Jan-01-1970", format = "%b-%d-%Y"))
```

```
[1] NA
```

```r
## For the complete list of code that is used to specify date formats
help(strptime)
```

The **chron package** can handle both date and time. However, it cannot work with time zones.

Using the **POSIXct** and **POSIXlt** class objects, we can work with time zones.

The **lubridate package** has a much more user-friendly functionality to process date and time, with time zone support:

- It can process date variables in heterogeneous formats.
- Note that the default time zone in the mdy, dmy, or ymd function is Coordinated Universal Time (UTC)

```r
## creating a heterogeneous date object
library(lubridate)
```

```
hetero_date <- c("second chapter due on 2013, august, 24", "first chapter submitted on 201
ymd(hetero_date)
```

- the sequence of year, month, and day should be similar across all values within the same object, otherwise during date extraction there will be a missing value that will be generated, along with a warning message.

11.2 Text Manipulation

Besides default R functionality (`paste()`, `nchar()`, `substr()` …) , there is one contributed package to deal with character data, which is more user friendly and intuitive: **stringr package**.

Strings can be arranged into **vectors and matrices** just like numbers. We can also paste strings together using `paste(..., sep)`. Here sep is an optional input, with default value " “, that determines which padding character is to be placed between the strings (which are input where … appears).

Example:

```
x <- "Citroen SM"
y <- "Jaguar XK150"
z <- "Ford Falcon GT-HO"
(wish.list <- paste(x, y, z, sep = ", "))
```

```
[1] "Citroen SM, Jaguar XK150, Ford Falcon GT-HO"
```

Special characters can be included in strings using the escape character `\`:

- `\"` for ”
- `\n` for a newline
- `\t` for a tab
- `\b` for a backspace
- `\\` for `\`

Text data can be used to retrieve information in sentiment analysis and even entity recognition.

11.2.1 Sources of Text data

Text data can be found on tweets from any individual, or from any company, Facebook status updates, RSS feeds from any news site, Blog articles, Journal articles, Newspapers, Verbatim transcripts of an in-depth interview.

For example, t extract **Twitter data**, we can use `tweetR()` and, to extract data from **Facebook**, we could use `facebookR()`.

11.2.2 Getting Text Data

The easiest way to get text data is to import from a .csv file where some of the variables contain character data. We have to protect automatic factor conversion by specifying the `stringsAsFactors = FALSE argument`

Example 1: The tweets.txt file is the plain text file. We will import this file using the generic `readLines()` function. It is a vector of characters (not a data.frame).

Example 2: Html (this is also a character string):

```r
conURL <- "http://en.wikipedia.org/wiki/R_%28programming_language%29"
# Establish the connection with the URL
link2URL <- url(conURL)
# Reading html code
htmlCode <- readLines(link2URL)
# Closing the connection
close(link2URL)
# Printing the result
htmlCode
```

The **tm** text mining library has some other functions to import text data from various files such as PDF files, plain text files, and even from doc files.

12 R and Data Bases

- R stores everything in RAM, and a typical personal computer consists of limited RAM. R is RAM intensive, and for that reason, the size of a dataset should be much smaller than its RAM.
- There are two principal ways to connect to a database:
  - the first uses the **ODBC facility** available on many computers and,
  - the second uses the **DBI package** of R along with a specialized package for the particular database needed to be accessed (if there is a specialized package available for a database).

12.0.1 R and Excel/MSAccess

- An Excel file can be imported into R using ODBC. Remember Excel cannot deal with relational databases.
- We will now create an ODBC connection with an MS Excel file with the **connection string xlopen**:
  - In our computer: To use the ODBC approach on an Excel file, we firstly need to create the connection string using the system administrator. We need to open the control panel of the operating system and then open Administrative Tools and then choose ODBC. A dialog box will now appear. Click on the Add button and select an appropriate ODBC driver and then locate the desired file and give a data source name. In our case, the data source name is xlopen.
  - In R:

```r
# calling ODBC library into R
library(RODBC)
# creating connection with the database using odbc package.
xldb <- odbcConnect("xlopen") / odbcConnect("accessdata")
# Now that the connection is created, we will use this connection and import the data
```

12.0.2 Relational databases in R

- There are packages to interface between R and different database software packages that use relational database management systems, such as **MySQL (RMySQL)**, **PostgreSQL (RPgSQL)**, and **Oracle (ROracle)**.
- One of the most popular packages is **RMySQL**. This package allows us to make connections between R and the MySQL server. In order to install this package properly, we need to download both the MySQL server and RMySQL.
- There are several R packages available that allow direct interactions with large datasets within R, such as **filehash**, **ff**, and **bigmemory**. The idea is to avoid loading the whole dataset into memory.

12.0.3 filehash package

- It is used for solving large-data problems. The idea behind the development of this package was to avoid loading the dataset into a computer's virtual memory. Instead, we dump the large dataset into the hard drive and then assign an environment name for the dumped objects.

```r
library(filehash)
dbCreate("exampledb")
filehash_db<- dbInit("exampledb")  ## db needs to be initialized before accessing
dbInsert(filehash_db, "xx", rnorm(50))
value<- dbFetch(filehash_db, "xx")  ## to retrieve db values
summary(value)
```

- This file connection will remain open until the database is closed via dbDisconnect or the database object in R is removed.

12.0.4 ff package

This package extends the R system and stores data in the form of native binary flat files in persistent storage such as hard disks, CDs, or DVDs rather than in the RAM.

This package enables users to work on several large datasets simultaneously. It also allows the allocation of vectors or arrays that are larger than the RAM.

12.0.5 sqldf package

The **sqldf package** is an R package that allows users to run SQL statements within R.

We can perform any type of data manipulation to an R data frame either in memory or during import.

If the dataset is too large and cannot entirely be read into the R environment, we can import a portion of that dataset using sqldf.

Part III

Part 2: Programming in R

13 Good Programming Habits

We find the following to be useful guidelines:

- Start each program with some **comments** giving the name of the program, the author, the date it was written, and what the program does. A description of what a program does should explain what all the inputs and outputs are.
- **Variable names** should be descriptive, that is, they should give a clue as to what the value of the variable represents. Avoid using reserved names or function names as variable names (in particular t, c, and q are all function names in R). You can find out whether or not your preferred name for an object is already in use by the `exists()` function.
- Use **blank lines** to separate sections of code into related parts, and use **indenting** to distinguish the inside part of an `if` statement or a `for` or `while` loop.
- **Document the programs** that you use in detail, ideally with citations for specific algorithms. There is no worse feeling than returning to undocumented code that had been written several years earlier to try to find and then explain an anomaly.

14 Logic

Control (logical) structures mentioned here are primarily useful for writing programs; for command-line interactive work, the “apply” functions are more useful.

14.1 Logical expressions

A **logical expression** is formed using:

- the **comparison operators** `<` , `>` , `<=`, `>=`, `==` (equal to), `!=` (not equal to), `&&`, `||` (sequentially evaluated versions of `&` and `|`, respectively), and
- the **logical operators** `&` (and), `|` (or), and `!` (not).

The order of operations can be controlled using parentheses `( )`.

The value of a logical expression is either **TRUE or FALSE** (the integers 1 and 0 can also be used to represent TRUE and FALSE, respectively).

Example:

```
## Note that A|B is TRUE if A or B or both are TRUE
c(0,0,1,1)|c(0,1,0,1)
```

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

```
## If you want exclusive disjunction, that is either A or B is TRUE but not both, then use
xor(c(0,0,1,1), c(0,1,0,1))
```

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

14.1.1 Sequential `&&` and `||`

To evaluate `x & y`, R first evaluates `x` and `y`, then returns `TRUE` if `x` and `y` are both `TRUE`, `FALSE` otherwise.

To evaluate `x && y`, R first evaluates x. If `x` is `FALSE` then R returns `FALSE` without evaluating `y`. If `x` is `TRUE` then R evaluates `y` and returns `TRUE` if `y` is `TRUE`, `FALSE` otherwise.

Sequential evaluation of `x` and `y` is useful when `y` is not always well defined, or when `y` takes a long time to compute.

Note that `&&` and `||` only work on scalars, whereas `&` and `|` work on vectors on an element-by-element basis.

14.1.2 Index position

If you wish to know the **index positions** of TRUE elements of a logical vector `x`, then use **which(x)**:

Example:

```r
x <- c(1, 1, 2, 3, 5, 8, 13)
which(x %% 2 == 0)
```

```
[1] 3 6
```

14.2 if-else

A natural extension of the if command includes an else part:

```r
if (logical_expression) {
  expression_1 # do something
  ...
} else {
  expression_2 # do something different
  ...
}
```

Braces `{ }` are used to group together one or more expressions. If there is only one expression then the braces are optional.

When an if expression is evaluated, if *logical_expression* is *TRUE* then the first group of expressions is executed and the second group of expressions is not executed. Conversely if *logical_expression* is *FALSE* then only the second group of expressions is executed.

14.2.1 elseif

When the `else` expression contains an `if`, then it can be written equivalently (and more clearly) as follows:

```
if (logical_expression_1) {
  expression_1
  ...
} else if (logical_expression_2) {
  expression_2
  ...
} else {
  expression_3
  ...
}
```

14.3 Loops

Tip: R is set up so that such programming tasks can be accomplished using vector operations rather than looping. Using vector operations is more efficient computationally, as well as more concise literally.

Example:

```
# We could find the sum of the first n squares using a loop as follows:
n <- 100
S <- 0
for (i in 1:n) {
  S <- S + i^2
}
S
```

```
[1] 338350
```

```
# Alternatively, using vector operations we have:
sum((1:n)^2)
```

```
[1] 338350
```

14.3.1 `for` Loops

The `for` command has the following form, where `x` is a simple variable and vector is a vector.

```
for (x in vector) {
  expression_1
  ...
}
```

Example:

```
x <- c("a", "b", "c", "d")
for(i in 1:4) / for(i in seq_along(x)) / for(letter in x) { ## set an iterator variable an
  print(x[i])
  print(letter)
}
```

When executed, the `for` command executes the group of expressions within the braces `{ }` once for each element of vector. The grouped expressions can use `x`, which takes on each of the values of the elements of vector as the loop is repeated.

14.3.2 nested `for` loops

Example:

```
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x))) {
  for(j in seq_len(ncol(x))) {
    print(x[i,j])
  }
}
```

14.3.3 `while` Loops

Often we do not know beforehand how many times we need to go around a loop. That is, each time we go around the loop, we check some condition to see if we are done yet. In this situation we use a `while` loop, which has the form:

```
while (logical_expression) {
  expression_1
  ...
}
```

When a `while` command is executed, *logical_expression* is evaluated first. If it is TRUE then the group of expressions in braces `{ }` is executed. Control is then passed back to the start of the command: if `logical_expression` is still `TRUE` then the grouped expressions are executed again, and so on. Clearly, for the loop to stop eventually, *logical_expression* must eventually be `FALSE`. To achieve this *logical_expression* usually depends on a variable that is altered within the grouped expressions.

Example:

```
time <- 0
debt <- debt_initial
while (debt > 0) {
  time <- time + period
  debt <- debt*(1 + r*period) - repayments
}
```

14.3.4 `repeat` Loops

`repeat` initiates an infinite loop right from the start. The only way to exit a repeat loop is to call break:

```
repeat {
  if() {
    break
  }
}
```

14.3.5 `next`, `break`

next is used to skip an iteration of a loop:

```
for(i in 1:100) {
  if(i <= 20) {
    next ## Skip the first 20 iterations
  }
  ## Do something here
}
```

break is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for(i in 1:100) {
  print(i)
  if(i > 20) {
    break ## Stop loop after 20 iterations
  }
}
```

14.4 Loop Functions

Many R **functions are vectorised**, meaning that given vector input the function acts on each element separately, and a vector output is returned. This is a very powerful aspect of R that allows for compact, efficient, and readable code. Moreover, for many R functions, applying the function to a vector is much faster than if we were to write a loop to apply it to each element one at a time.

Besides, writing **for** and **while loops** is useful when programming but not particularly easy when working interactively on the command line.

R has some functions which implement looping in a compact form to make your life easier:

```
lapply() ## Apply the function FUN to every element of vector X
sapply() ## Same as lapply but try to simplify the result
apply() ## Apply a function that takes a vector argument to each of the rows (or columns)
tapply() ## Apply a function over subsets of a vector
mapply() ## Multivariate version of lapply (to vectorise over more than one argument)
```

`X` can be a **list or an atomic vector**, which is a vector that comprises atomic objects (logical, integer, numeric, complex, character and raw). That is, `sapply(X, FUN)` returns a vector whose i -th element is the value of the expression `FUN(X[i])`. Note that R performs a loop over the elements of `X`, so execution of this code is not faster than execution of an equivalent loop.

If `FUN` has arguments other than `X[i]`, then they can be included using the dots protocol as shown above. That is, `sapply(X, FUN, ...)` returns `FUN(X[i], ...)` as the i -th element.

14.4.1 lapply

The `lapply()` function does the following simple series of operations:

1. It loops over a **list**, iterating over each element in that list
2. It applies a function to each element of the list (a function that you specify)
3. Returns a list (the "l" is for "list").

`lapply()` **always returns a list**, regardless of the class of the input.

```
str(lapply)
```

```
function (X, FUN, ...)
```

Note that the `function()` definition is right in the call to `lapply()`:

```
lapply(x, function(elt) { elt[,1] }
```

Example:

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
$a
[1] 3

$b
[1] 0.3088623
```

Note that when you pass a function to another function, you do not need to include the open and **closed parentheses ()** like you do when you are calling a function.

Once the call to `lapply()` is finished, the function disappears and does not appear in the workspace.

14.4.2 sapply

The `sapply()` function behaves similarly to `lapply()`, the only real difference being that the **return value is a vector or a matrix**.

```
str(sapply)
```

```
function (X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

`sapply()` will try to simplify the result of `lapply()` if possible.

Example:

```r
s <- split(airquality, airquality$Month)
mode(s)
```

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

```r
(q <- sapply(s, function(x) {
              colMeans(x[, c("Ozone", "Solar.R", "Wind")])
          }
))
```

```
               5         6          7        8        9
Ozone         NA        NA         NA       NA       NA
Solar.R       NA 190.16667 216.483871       NA 167.4333
Wind    11.62258  10.26667   8.941935 8.793548  10.1800
```

```r
mode(q)
```

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

14.4.3 apply

The `apply()` function can take a list, matrix, or array. It is most often used to apply a function to the **rows or columns of a matrix** (which is just a 2-dimensional array).

```r
str(apply)
```

```
function (X, MARGIN, FUN, ..., simplify = TRUE)
```

Using `apply()` is not really faster than writing a loop, but it works in one line and is highly compact.

```r
x <- matrix(rnorm(200), 20, 10)
# Take the mean of each column
(k <- apply(x, 2, mean))
```

```
 [1] -0.05896163 -0.42061693  0.13710343  0.25075345  0.17346043 -0.20272873
 [7]  0.15053168  0.09290890  0.18203441 -0.19219215
```

```r
mode(k)
```

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

```r
## Take the mean of each row
apply(x, 1, sum)
```

```
 [1]  2.7626398  0.4275056 -2.2808976  3.1888434 -0.2517535  6.2663674
 [7]  4.3999215 -3.6928629  0.5472288  1.2877206 -4.3892739  4.5661766
[13] -3.7284066 -4.3955953 -2.3021029  1.4905296 -1.4619682 -2.7278041
[19]  6.1565319 -3.6169429
```

For the special case of **column/row sums** and **column/row means** of matrices, there are some useful **shortcuts**:

```r
rowSums = apply(x, 1, sum)
rowMeans = apply(x, 1, mean)
colSums = apply(x, 2, sum)
colMeans = apply(x, 2, mean)
```

14.4.4 tapply

`tapply()` is used to apply a function over subsets of a vector. It can be thought of as a combination of `split()` and `sapply()` for vectors only.

```r
str(tapply)
```

```
function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

Example:

```
(x <- c(rnorm(10), runif(10), rnorm(10, 1)))
```

```
 [1] -0.69486941 -0.40638353 -0.27349063  0.31304461 -0.39084935  1.57816162
 [7] -0.85216944  1.44920253  1.89753729 -1.68443062  0.13719097  0.57267147
[13]  0.41513185  0.09161501  0.84858734  0.11389691  0.97183830  0.47352737
[19]  0.67680390  0.40786657  0.78265156  1.78561430  2.41104568  1.61531871
[25]  0.03061577  0.19932418  1.10485212  0.23091056  0.27503733 -0.06321824
```

```
# Define some groups with a factor variable
(f <- gl(3, 10))
```

```
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
```

```
tapply(x, f, mean)
```

```
         1          2          3 
0.09357531 0.47091297 0.83721520 
```

```
# We can reduce the noise as follows:
round(tapply(x, f, mean), digits=1)
```

```
  1   2   3 
0.1 0.5 0.8 
```

14.4.5 mapply

The `mapply()` function is a multivariate apply of sorts which applies a function in parallel over a set of arguments. Recall that `lapply()` and friends only iterate over a single R object. What if you want to iterate over multiple R objects in parallel? This is what `mapply()` is for.

```
str(mapply)
```

```
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

The `mapply()` function has a different argument order from `lapply()` because the function to apply comes first rather than the object to iterate over. The R objects over which we apply the function are given in the `...` argument because we can apply over an arbitrary number of R objects.

Example 1:

```
mapply(rep, 1:4, 4:1)
```

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

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

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

Example 2:

```
noise <- function(n, mean, sd) {
            rnorm(n, mean, sd)
          }
mapply(noise, 1:5, 1:5, 2)
```

```
[[1]]
[1] 1.531098

[[2]]
[1] 0.9638613 2.7220422

[[3]]
[1] 2.063231448 3.107106359 0.008673902

[[4]]
[1] -0.01631722  1.27070805  3.98643399  4.53642900

[[5]]
[1] 5.877427 4.722149 2.057272 3.404604 8.596655
```

The `mapply()` function can be used to automatically "vectorize" a function. What this means is that it can be used to take a function that typically only takes single arguments and create a new function that can take vector arguments. This is often needed when you want to plot functions.

Example:

```
## Generate some data
x <- rnorm(100)
## This is not what we want
sumsq(1:10, 1:10, x)
## Note that the call to sumsq() only produced one value instead of 10 values.
mapply(sumsq, 1:10, 1:10, MoreArgs = list(x = x))
```

There's even a function in R called `vectorize()` that automatically can create a vectorized version of your function. So, we could create a `vsumsq()` function that is fully vectorized as follows.

Example:

```
vsumsq <- vectorize(sumsq, c("mu", "sigma"))
vsumsq(1:10, 1:10, x)
```

14.4.6 split

The `split()` function takes a vector or other objects and splits it into groups determined by a factor or list of factors.

The form of the `split()` function is this:

```
str(split)
```

```
function (x, f, drop = FALSE, ...)
```

The combination of `split()` and a function like `lapply()` or `sapply()` is a common paradigm in R.

Example 1:

```
x <- c(rnorm(10), runif(10), rnorm(10, 1))
f <- gl(3, 10)
# split() returns a list
```

```
(j <- split(x, f))
```

```
$`1`
 [1]  0.82927744  0.13701989  1.23567957  0.58319080 -0.01399431 -1.39287967
 [7] -0.18183604  0.41847548 -1.10184827  0.25278899

$`2`
 [1] 0.41574371 0.29452943 0.19368774 0.35787964 0.17545609 0.09994724
 [7] 0.28854207 0.30585732 0.89268637 0.12439036

$`3`
 [1]  1.2541046  3.8834575  1.8097673  1.1839733  1.5447306  0.6680359
 [7] -0.6020763 -1.1885185  0.1257680  0.8499511
```

```
mode(j)
```

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

Example 2:

```
# Splitting a Dataframe
s <- split(airquality, airquality$Month)
```

15 Functions

Functions are one of the main building blocks for large programs: they are an essential tool for structuring complex algorithms. Arguably one of R's strengths as a tool for scientific programming is the ease with which it can be extended for specific purposes, using functions written by the **R community** and made available as **R packages**.

The most important advantage of using a function is that once it is loaded, it can be **used again and again** without having to reload it. **User-defined functions** can be used in the same way as predefined functions are used in R. In particular they can be used within other functions.

The second most important use of functions is to **break down** a programming task into smaller logical units. Large programs are typically made up of a number of smaller functions, each of which does a simple well-defined task.

15.1 Building Functions

Functions are often used to encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions. Functions are also often written when code must be shared with others or the public.

The writing of a function allows a developer to create an **interface to the code**, that is explicitly specified with a **set of parameters**.

A function has the general **form**:

```
name <- function(argument_1, argument_2, ...) {
  expression_1
  expression_2
  <some other expressions>
  return(output)
}
```

Here `argument_1`, `argument_2`, etc., are the names of variables and `expression_1`, `expression_2`, and `output` are all regular R expressions.

`name` is the name of the function. Because all function arguments have names, they can be specified using their name. Specifying an **argument by its name** is sometimes useful if a function has many arguments and it may not always be clear which argument is being specified.

Example:

```
f(num = 2)
```

Some functions may have **no arguments**, and that the **braces** are only necessary if the function comprises more than one expression.

Example:

```
f <- function(num = 1) {
## if the function is called without the num argument being explicitly specified, then it
  hello <- "Hello, world!\n"
  for(i in seq_len(num)) {
    cat(hello)
  }
  chars <- nchar(hello) * num
  chars
}
```

In R, the return value of a function is always the very last expression that is evaluated (in the example is `chars`).

The `formals()` function returns a list of all the formal arguments of a function.

Note that functions have their own **class**.

15.2 Call or Run a Function

To **call or run the function** we type (for example) `name(x1, x2)`. The value of this expression is the value of the expression `output`. To calculate the value of `output` the function first copies the value of `x1` to `argument_1`, `x2` to `argument_2`, and so on. The arguments then act as variables within the function. We say that the arguments have been passed to the function. Next the function evaluates the grouped expressions contained in the braces `{ }`; the value of the expression `output` is returned as the value of the function.

To **use the function** we first load it (using source or by copying and pasting into R), then call it, supplying suitable arguments.

Example:

```
rm(list=ls())
source("../scripts/quad3.r")
quad3(1,0,-1)
```

Note that the **name of the function** does not have to match the **name of the program file**, but when a program consists of a single function this is conventional.

15.3 A Function Return

A function always returns a value. For some functions the value returned is unimportant, for example if the function has written its output to a file then there may be no need to return a value as well. In such cases one usually omits the return statement, or returns NULL.

A function may have **more than one return statement**, in which case it stops after executing the first one it reaches. If there is no `return(output)` statement then the value returned by the function is the value of the last expression in the braces.

If, when called, the value returned by a function (or any expression) is not assigned to a variable, then it is printed. The expression `invisible(x)` will return the same value as x, but its value is not printed. For example, some versions of the `summary()` function use invisible on their returned object.

15.4 Arguments

We think about **arguments** both when the functions are written and when they are called. The arguments of an existing function can be obtained by calling the `formals` function.

In order to simplify calling functions, some arguments may be assigned **default values**, which are used in case the argument is not provided in the call to the function:

Example:

```
test3 <- function(x = 1) {
  return(x)
  }
test3(2)
```

```
[1] 2
```

```
test3()
```

```
[1] 1
```

Sometimes you will want to define arguments so that they can take only a **small number** of different values, and the function will stop informatively if an inappropriate value is passed.

Example:

```
funk <- function(vibe = c("Do","Be","Dooby","Dooo")) {
  vibe <- match.arg(vibe)
  return(vibe)
  }
funk()
```

```
[1] "Do"
```

```
funk(Peter)
Error in match.arg(vibe) (from #2) :
'arg' should be one of "Do", "Be", "Dooby", "Dooo"
```

15.4.1 Argument Matching

R functions arguments can be matched (called) **positionally** or **by name**:

- **Positional matching** just means that R assigns the first value to the first argument, the second value to second argument, etc.:

  Example:

  ```
  # Positional match first by argument, default for 'na.rm'
  sd(mydata)
  # Specify 'x' argument by name, default for 'na.rm'
  sd(x = mydata)
  # Specify both arguments by name
  sd(x = mydata, na.rm = FALSE)
  ```

- When specifying the function **arguments by name**, it doesn't matter in what order you specify them. Named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list:

```
## Specify both arguments by name
sd(na.rm = FALSE, x = mydata)
```

- R also provides **partial matching of arguments**, where doing so is not ambiguous. This means that argument names in the call to the function do not have to be complete. Reliance on partial matching makes code more fragile.

```
test6 <- function(a = 1, b.c.d = 1) {
  return(a + b.c.d)
  }
test6()
```

```
[1] 2
```

```
test6(b = 5)
```

```
[1] 6
```

15.4.2 The … Argument

`...`indicates a variable number of arguments that are usually passed on to other functions, it is often used when extending another function and you don't want to copy the entire argument list of the original function.

The `...` argument is necessary when the number of arguments passed to the function cannot be known in advance, e.g. `paste()`, `cat()`.

Any arguments that appear after `...` on the argument list must be named explicitly and cannot be partially matched or matched positionally.

R provides a very useful means of **passing arguments**, unaltered, from the function that is being called to the functions that are called within it. These arguments do not need to be named explicitly in the outer function, hence providing great flexibility. To use this facility you need to include `...` in your argument list. These three dots (an ellipsis) act as a placeholder for any extra arguments given to the function.

Example:

```
test4 <- function(x, ...) {
  return(sd(x, ...))
  }
test4(1:3)
```

```
[1] 1
```

```
# Arguments that do not match those in test4 are provided, in order, to any function withi
test4(c(1:2,NA), na.rm = TRUE)
```

```
[1] 0.7071068
```

Using the dots in this way means that the user has access to all the function arguments without our needing to list them when we define the function.

In general, **naming the arguments** in the function call is good practice, because it increases the readability and eliminates one potential source of errors.

15.5 Scoping Rules

Arguments and variables that are defined within a function exist only within that function. That is, if you define and use a variable `x` inside a function, it does not exist outside the function. If variables with the same name exist inside and outside a function, then they are separate and do not interact at all. You can think of a function as a separate environment that communicates with the outside world only through the values of its arguments and its output expression. For example if you execute the command `rm(list=ls())` inside a function (which is only rarely a good idea), you only delete those objects that are defined inside the function.

Example:

```
test <- function(x) {
  y <- x + 1
  return(y)
  }
test(1)
```

```
[1] 2
```

```
x
Error: Object "x" not found
```

That part of a program in which a variable is defined is called its **scope**. Restricting the scope of variables within a function provides an assurance that calling the function will not modify variables outside the function, except by assigning the returned value.

Beware, however, the **scope of a variable is not symmetric**. That is, variables defined inside a function cannot be seen outside, but variables defined outside the function can be seen inside the function, provided there is not a variable with the same name defined inside.This arrangement makes it possible to write a **function** whose behavior depends on the **context** within which it is run.

Example:

```r
test2 <- function(x) {
  y <- x + z
  return(y)
  }
z <- 1
test2(1)
```

```
[1] 2
```

The moral of this example is that it is generally advisable to ensure that the variables you use in a function either are declared as arguments, or have been defined in the function.

16 Debugging

Often code will be used in circumstances under which you cannot control the type of input (numeric, character, logical, etc.). **Unexpected input** can lead to undesirable consequences, for example, the function could fail to work and the user may not know why. Worse still, the function could seem to work but return plausible nonsense, and the user may be none the wiser

R has a number of ways to indicate to you that **something's not right**. There are different levels of indication that can be used, ranging from mere notification to fatal error.

First, R may **stop processing and report an error**, along with a brief summary of the violation. It can be worth performing simple checks on the input to be sure that it conforms to your expectations. (Useful considerations here are: what will your function do if the input is the wrong type, or the right type but incomplete?)

- The **stop function** is useful in these circumstances: `stop("Your message here.")` will cease processing and print the message to the user.
- The **browser function** is very useful to invoke inside your own functions. The command `browser()` will temporarily stop the program, and allow you to inspect its objects. You can also step through the code, executing one expression at a time.

  Example:

  ```
  my_fun <- function(x) {
    browser()
    y <- x * z
    return(y)
  }
  my_fun(c(1,2,3))
  ```

  ```
  Called from: my_fun(c(1,2,3))
  Browse[1]>
  ```

  `browser` catches the execution and presents us with a prompt. Using `n`, we will step through the function one line at a time. At each point, R shows us the next line to be evaluated. We signify our input using curly braces; thus: `{ Enter }`.

Second, we may identify an error by **examining the output** of a program or function, and noting that it makes no sense.

Finally, R may throw a **warning** upon the execution of some code, and this warning might point to an earlier error.

To ask R to convert warnings into errors, and hence stop processing at the point of warning, type:

```
options(warn = 2)
```

16.1 Show values

You will spend a lot of time correcting errors in your programs. To find an error or bug, you need to be able to see how your variables change as you move through the branches and loops of your code.

An effective and simple way of doing this is to include statements like `cat("var =", var, "\n")` throughout the program, to display the values of variables such as var as the program executes. Once you have the program working you can delete these or just comment them so they are not executed.

It is also very helpful to make **dry runs of your code**, using simple starting conditions for which you know what the answer should be. These dry runs should ideally use short and simple versions of the final program, so that analysis of the output can be kept as simple as possible.

Graphs and summary statistics of intermediate outcomes can be very revealing, and the code to create them is easily commented out for production runs.

16.2 Message

We can fix this problem by anticipating the possibility of NA values and checking to see if the input is NA with the `is.na()` function:

```
printmessage3 <- function(x) {
    if(length(x) > 1L)
        stop("'x' has length > 1")
        if(is.na(x))
            print("x is a missing value!")
    else if(x > 0)
```

```
        print("x is greater than zero")
    else
        print("x is less than or equal to zero")
    invisible(x)
}
```

16.3 traceback()

The `traceback()` command prints out the function call stack **after an error** has occurred. The function call stack is the sequence of functions that was called before the error occurred.

The `traceback()` command must be called **immediately after** an error occurs. Once another function is called, you lose the traceback.

```
lm(y ~ x)
```

```
Error in eval(expr, envir, enclos) : object 'y' not found
```

```
traceback()
```

```
7: eval(expr, envir, enclos)
6: eval(predvars, data, env)
5: model.frame.default(formula = y \~ x, drop.unused.levels = TRUE)
4: model.frame(formula = y \~ x, drop.unused.levels = TRUE)
3: eval(expr, envir, enclos)
2: eval(mf, parent.frame())
1: lm(y \~ x)
```

Looking at the **traceback** is useful for figuring out roughly where an error occurred but it's not useful for more detailed debugging. For that you might turn to the `debug()` function.

16.4 debug()

The `debug()` function takes a function as its first argument. Here is an example of debugging the `lm()` function.

```
debug(lm)    ## Flag the 'lm()' function for interactive debugging
lm(y ~ x) debugging
```

Now, every time you call the `lm()` function it will launch the interactive debugger. To turn this behavior off you need to call the `undebug()` function.

The debugger calls the browser at the very top level of the function body. From there you can step through each expression in the body. There are a few **special commands** you can call in the browser:

- **n**: executes the current expression and moves to the next expression.
- **c**: continues execution of the function and does not stop until either an error or the function exits.
- **Q**: quits the browser.

You can turn off interactive debugging with the `undebug()` function like this:

```
undebug(lm)  ## Unflag the 'lm()' function for debugging
```

16.5 recover()

The `recover()` function can be used to modify the error behavior of R when an error occurs. Normally, when an error occurs in a function, R will print out an error message, exit out of the function, and return you to your workspace to await further commands.

With `recover()` you can tell R that when an error occurs, it should halt execution at the exact point at which the error occurred.

```
options(error = recover)  ## Change default R error behavior
read.csv("nosuchfile")  ## This code doesn't work
```

17 Profiling R Code

R comes with a **profiler** to help you optimize your code and improve its performance. Better to get all the bugs out first, then focus on optimizing.

Of course, when it comes to **optimizing code**, the question is what should you optimize? Well, clearly should optimize the parts of your code that are running slowly, but how do we know what parts those are?

Profiling is a systematic way to examine how much time is spent in different parts of a program.

17.1 system.time()

The `system.time()` function computes the time (in seconds) needed to execute an expression and if there's an error, gives the time until the error occurred. Using `system.time()` allows you to test certain functions or code blocks to see if they are taking excessive amounts of time.

The function returns an object of **class proc_time** which contains two useful bits of information:

- user time: time charged to the CPU(s) for this expression.
- elapsed time: "wall clock" time, the amount of time that passes for you as you're sitting there.

The elapsed time may be greater than the user time if the CPU spends a lot of time waiting around. The elapsed time may be smaller than the user time if your machine has multiple cores/processors (and is capable of using them).

Here's an example of where the elapsed time is greater than the user time:

```r
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
```

```
   user  system elapsed 
   0.00    0.02    0.86 
```

You can time longer expressions by wrapping them in curly braces within the call to `system.time()`.

```
system.time({
  ## expression or loop or function
  })
```

17.2 The R Profiler

`system.time()` assumes that you already know where the problem is and can call `system.time()` on that piece of code. What if you don't know where to start? This is where the profiler comes in handy.

In conjunction with `Rprof()`, we will use the `summaryRprof()` function which summarizes the output from `Rprof()`.

`Rprof()` keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent inside each function.

The profiler is started by calling the `Rprof()` function:

```
Rprof()  ## Turn on the profiler
```

Once you call the `Rprof()` function, everything that you do from then on will be measured by the profiler. Therefore, you usually only want to run a single R function or expression once you turn on the profiler and then immediately turn it off.

The profiler can be turned off by passing NULL to `Rprof()`:

```
Rprof(NULL)  ## Turn off the profiler
summaryRprof()
```

The `summaryRprof()` function tabulates the R profiler output and calculates how much time is spend in which function. There are two methods for normalizing the data:

- **"by.total"**: divides the time spend in each function by the total run time.
- **"by.self"**: does the same as "by.total" but first subtracts out time spent in functions above the current function in the call stack.

The final bit of output that `summaryRprof()` provides is the sampling interval and the total runtime.

```
$sample.interval
```

[1] 0.02

```
$sampling.time
```

[1] 7.41

Part IV

Part 3: Statistics in R

18 Descriptive Statistics

Descriptive statistics summarizes the data set, lets us have a feel and understanding of the data and variables. Descriptive statistics usually focuses on: - the **distribution**: can be a normal distribution, binomial distribution, and other distributions like Bernoulli distribution. - the **central tendency**: can be the mean, median, and mode of the data. - the **dispersion of the data**: it describes the spread of the data, and dispersion can be the variance, standard deviation, and interquartile range.

Descriptive statistics allows us to decide or determine whether we should use inferential statistics to identify the relationship between data sets, or use regression analysis instead to identify the relationships between variables.

18.1 summary() and str()

The `summary()` and `str()` functions are the fastest ways to get descriptive statistics of the data.

The `summary()` function gives the basic descriptive statistics of the data.

The `str()` function gives the structure of the variables.

18.2 Measures of Centrality

18.2.1 Mode

The **mode** is a value in data that has the highest frequency:

Example:

```r
a <- c(1, 2, 3, 4, 5, 5, 5, 6, 7, 8)
y <- table(a)  ## To get mode in a vector, you create a frequency table
y
```

```
a
1 2 3 4 5 6 7 8
1 1 1 1 3 1 1 1
```

```
names(y)[which(y==max(y))]
```

```
[1] "5"
```

18.2.2 Median

The **median** is the middle or midpoint of the data and is also the 50 percentile of the data.

The **median** is affected by the outliers and skewness of the data.

```
median(A)
```

18.2.3 Mean

The **mean** is the average of the data. The mean works best if the data is distributed in a normal distribution or distributed evenly.

```
mean(A)
```

18.3 Measures of Variability

The **measures of variability** are the measures of the spread of the data.

The **measures of variability* can be: - variance - standard deviation - range - interquartile range - and more.

18.3.1 Variance

The **variance** is the average of squared differences from the mean, and it is used to measure the spreadness of the data:

- **Population variance**:

```r
A <- c(1, 2, 3, 4, 5, 5, 5, 6, 7, 8)
N <- length(A)
var(A) * (N - 1) / N
```

```
[1] 4.24
```

- **Sample variance**:

```r
var(A)
```

```
[1] 4.711111
```

18.3.2 Standard Deviation

The **standard deviation** is the square root of a variance and it measures the spread of the data.

- **Population standard deviation**:

```r
A <- c(1, 2, 3, 4, 5, 5, 5, 6, 7, 8)
N <- length(A)
variance <- var(A) * (N - 1) / N
sqrt(variance)
```

```
[1] 2.059126
```

- **Sample standard deviation**:

```r
sd(A)
```

```
[1] 2.170509
```

18.3.3 range()

The **range** is the difference between the largest and smallest points in the data:

```r
range(A)
```

```
[1] 1 8
```

```r
res <- range(A)
diff(res)
```

```
[1] 7
```

```r
min(A)
```

```
[1] 1
```

```r
max(A)
```

```
[1] 8
```

18.3.4 Interquartile Range

The **interquartile range** is the measure of the difference between the 75 percentile or third quartile and the 25 percentile or first quartile.

```r
IQR(A)
```

```
[1] 2.5
```

You can get the quartiles by using the quantile() function:

```r
quantile(A)
```

```
  0%  25%  50%  75% 100% 
1.00 3.25 5.00 5.75 8.00 
```

18.4 Distributions

18.4.1 Normal Distribution

If the points do not deviate away from the line, the data is normally distributed.

To see whether data is normally distributed, you can use the `qqnorm()` and `qqline()` functions:

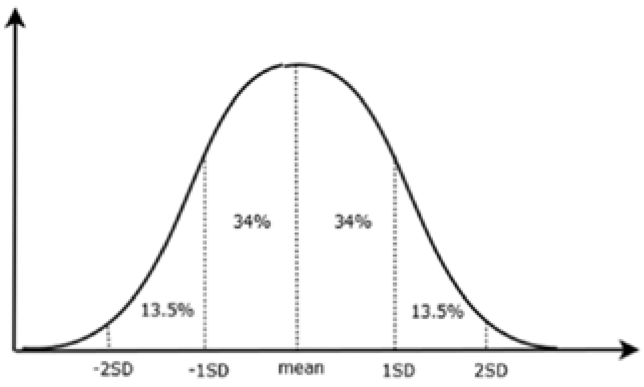


Figure 18.1: Figure 1: The normal distribution

```r
qqnorm(data$x) #You must first draw the distribution to draw the line afterwards
qqline(data$x)
```

You can also use a **Shapiro Test** to test whether the data is normally distributed. If the p-value is more than 0.05, you can conclude that the data does not deviate from normal distribution:

```r
shapiro.test(data$x)
```

18.4.2 Modality

The **modality** of a distribution can be seen by the number of peaks when we plot the histogram.

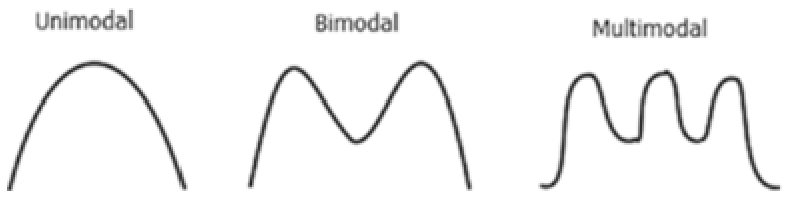


Figure 18.2: Figure 2: The modality of a distribution

18.4.3 Skewness

Skewness is a measure of how symmetric a distribution is and how much the distribution is different from the normal distribution.

Negative skew is also known as left skewed, and **positive skew** is also known as right skewed: - A positive skewness indicates that the size of the right-handed tail is larger than the left-handed tail. - A negative skewness indicates that the left-hand tail will typically be longer than the right-hand tail.

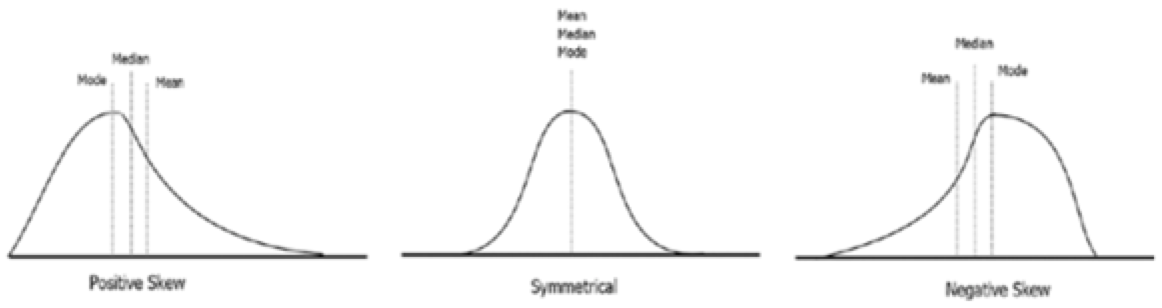


Figure 18.3: Figure 3: Skewness of a distribution

The **Pearson's Kurtosis measure** is used to see whether a dataset is heavy tailed, or light tailed. High kurtosis means heavy tailed, so there are more outliers in the data.

- When kurtosis is close to 0, then a normal distribution is often assumed. These are called mesokurtic distributions.

- When kurtosis>0, then the distribution has heavier tails and is called a leptokurtic distribution.
- When kurtosis<0, then the distribution is light tails and is called a platykurtic distribution.

To find the kurtosis and skewness in R, you must install the **moments package**:

```
install.packages("moments")
skewness(data$x)
kurtosis(data$x)
```

18.4.4 Binomial Distribution

A **binomial distribution** has two outcomes, success or failure, and can be thought of as the probability of success or failure in a survey that is repeated various times.

```
dbinom(32, 100, 0.5)
```

```
[1] 0.000112817
```

19 Exploratory Analysis

19.1 Analysis Pipeline

19.1.1 Formulate your question

A sharp question or hypothesis can serve as a dimension reduction tool that can eliminate variables that are not immediately relevant to the question.

It's usually a good idea to spend a few minutes to figure out what is the question you're really interested in and narrow it down to be as specific as possible (without becoming uninteresting).

One of the most important questions you can answer with an exploratory data analysis is: **"Do I have the right data to answer this question?"**

19.1.2 Read in your data

Sometimes the data will come in a very messy format, and you'll need to do some cleaning.

The **readr package** by Hadley Wickham is a nice package for reading in flat files very fast, or at least much faster than R's built-in functions

19.1.3 Check the packaging

Assuming you don't get any warnings or errors when reading in the dataset, you should now have an object in your workspace named e.g. "ozone". It's usually a good idea to poke at that object a little bit before we break open the wrapping paper.

For example, you can check the number of rows and columns.

19.1.4 Run str()

This is usually a safe operation in the sense that even with a very large dataset, running str() shouldn't take too long.

You can examine the classes of each of the columns to make sure they are correctly specified (i.e., numbers are numeric, and strings are character, etc.).

19.1.5 Look at the top and the bottom of your data

This lets me know if the data were read in properly, things are properly formatted, and that everything is there. If your data are time series data, then make sure the dates at the beginning and end of the dataset match what you expect the beginning and ending time period are.

You can peek at the top and bottom of the data with the `head()` and `tail()` functions. Sometimes there's weird formatting at the end or some extra comment lines that someone decided to stick at the end.

19.1.6 Check your "n"s (frequency)

To do this properly, you need to identify some landmarks that can be used to check against your data. For example, if you are collecting data on people, such as in a survey or clinical trial, then you should know how many people there are in your study (i.e., in an ozone monitoring data system we can take a look at the Time.Local variable to see what time measurements are recorded as being taken.)

```
table(ozone$Time.Local)
```

19.1.7 Validate with at least one external data source

External validation can often be as simple as checking your data against a single number.

The data are at least of the right order of magnitude (i.e., the units are correct) or, the range of the distribution is roughly what we'd expect, given the regulation around ambient pollution levels.

19.1.8 Try the easy solution first

Because we want to know which counties have the highest levels, it seems we need a list of counties that are ordered from highest to lowest with respect to their levels of ozone.

```r
ranking <- group_by(ozone, State.Name, County.Name) %>%
  summarize(ozone = mean(Sample.Measurement)) %>%
  as.data.frame %>%
  arrange(desc(ozone))
```

19.1.9 Challenge your solution/Bootstrap sample

The easy solution is nice because it is, well, easy, but you should never allow those results to hold the day. You should always be thinking of ways to challenge the results, especially if those results comport with your prior expectation.

How stable are the rankings from year to year? We can imagine that from year to year, the ozone data are somewhat different randomly, but generally follow similar patterns across the country. So, the shuffling process could approximate the data changing from one year to the next. It's not an ideal solution, but it could give us a sense of how stable the rankings are.

First, we set our random number generator and resample the indices of the rows of the data frame with replacement. The statistical jargon for this approach is a **bootstrap sample**:

1. We use the resampled indices to create a new dataset, ozone2, that shares many of the same qualities as the original but is randomly perturbed.

   ```r
   set.seed(10234)
   N <- nrow(ozone)
   idx <- sample(N, N, replace = TRUE)
   ozone2 <- ozone[idx, ]
   ```

2. We reconstruct our rankings of the counties based on this resampled data:

   ```r
   ranking2 <- group_by(ozone2, State.Name, County.Name) %>%
     summarize(ozone = mean(Sample.Measurement)) %>%
     as.data.frame %>%
     arrange(desc(ozone))
   ```

3. We can then compare the top 10 counties from our original ranking and the top 10 counties from our ranking based on the resampled data.

   ```r
   cbind(head(ranking, 10)
   head(ranking2, 10))
   ```

4. We can see that the rankings based on the resampled data are very close to the original, with the first 7 being identical. Numbers 8 and 9 get flipped in the resampled rankings but that's about it. This might suggest that the original rankings are somewhat stable.

19.1.10 Follow up

At this point it's useful to consider a few follow up questions:

1. Do you have the right data?
2. Do you need other data?
3. Do you have the right question?

The goal of exploratory data analysis is to get you thinking about your data and reasoning about your question. At this point, we can refine our question or collect new data, all in an iterative process to get at the truth.

19.2 Hierarchical Clustering

20 Inferential Statistics

Inferential statistics describes and makes inferences about the population from the sampled data, using hypothesis testing and estimating of parameters.

- In **hypothesis testing**, a research question is a hypothesis asked in question format
  - Based on the research question, the hypothesis can be a null hypothesis, $H_0(\mu_1 = \mu_2)$ and an alternate hypothesis, H_a $(\mu_1 \quad \mu_2)$.
  - If the **p-value** is less than or equal to alpha, which is usually 0.05, you reject the null hypothesis and say that the alternate hypothesis is true at the 95% confidence interval. If the p-value is more than 0.05, you fail to reject the null hypothesis.
- By **estimating parameters**, you try to answer the population parameters.
  - For estimating parameters, the parameters can be the mean, variance, standard deviation, and others.
  - E.g. you calculate the mean of the height of the samples and then make an inference on the mean of the height of the population. You can then construct the confidence intervals, which is the range in which the mean of the height of the population will fall.,

20.1 Correlation

Correlations are statistical associations to find how close two variables are and to derive the linear relationships between them.

You can use **correlation** to find which variables are more related to the target variable and use this to reduce the number of variables.

Correlation does not mean a causal relationship, it does not tell you the how and why of the relationship.

```r
cor(data$var1, data$var2)
```

The correlation has a range from -1.0 to 1.0.

20.2 Covariance

Covariance is a measure of variability between two variables.

The greater the value of one variable and the greater of other variable means it will result in a covariance that is positive.

```
cov(data$var1, data$var2)
```

Covariance does not have a range. When two variables are independent of each other, the covariance is zero.

20.3 Hypothesis Testing and P-Value

Based on the research question, the hypothesis can be a null hypothesis, H0 (1= 2) and an alternate hypothesis, Ha (1 2).

For data normally distributed:

- **p-value**:
  - A small p-value <= alpha, which is usually 0.05, indicates that the observed data is sufficiently inconsistent with the null hypothesis, so the null hypothesis may be rejected. The alternate hypothesis is true at the 95% confidence interval.
  - A larger p-value means that you failed to reject null hypothesis.
- **t-test** continuous variables of data.
- **chi-square test** for categorical variables or data.
- **ANOVA**

For data not normally distributed:

- **non-parametric** tests.

20.4 T-Test

A **t-test** is used to determine whether the mean between two data points or samples are equal to each other.

- H_0 $(\mu_1 = \mu_2)$: The null hypothesis means that the two means are equal.
- H_a $(\mu_1 \quad \mu_2)$: The alternative means that the two means are different.

In **t-test** there are two assumptions:

- The population is normally distributed.
- The samples are randomly sampled from their population.

Type I and Type II Errors:

- A **Type I error** is a rejection of the null hypothesis when it is really true.
- A **Type II error** is a failure to reject a null hypothesis that is false.

20.4.1 One-Sample T-Test

A **one-sample t-test** is used to test whether the mean of a population is equal to a specified mean.

You can use the t statistics and the degree of freedom $(df = n - 1)$ to estimate the p-value using a t-table.

```
t.test(data$var1, mu=0.6)
```

20.4.2 Two-Sample Independent T-Test (unpaired, paired = FALSE)

The two-sample unpaired t-test is when you compare two means of two independent samples. The degrees of freedom formula is $df = nA–nB–2$

In the two-sample unpaired t-test, when the variance is unequal, you use the **Welch t-test**.

```
t.test(data$var1, data\$var2, var.equal=FALSE, paired=FALSE)
```

20.4.3 Two-Sample Dependent T-Test (paired = TRUE)

A two-sample paired t-test is used to test the mean of two samples that depend on each other. The degree of freedom formula is $df = n - 1$

```
t.test(data$var1, data$var2, paired=TRUE)
```

20.5 Chi-Square Test

The **chi-square test** is used to compare the relationships between two categorical variables.

The null hypothesis means that there is no relationship between the categorical variables.

20.5.1 Goodness of Fit Test

When you have only **one categorical variable** from a population and you want to compare whether the sample is consistent with a hypothesized distribution, you can use the goodness of fit test.

- H_0: No significant difference between the observed and expected values.
- H_A: There is a significant difference between the observed and expected values.

To use the **goodness of fit chi-square test** in R, you can use the `chisq.test()` function:

```
data <- c(B=200, c=300, D=400)
chisq.test(data)
```

20.5.2 Contingency Test

If you have **two categorical variables** and you want to compare whether there is a relationship between two variables, you can use the **contingency test**.

- H_0: the two categorical variables have no relationship. The two variables are independent.
- H_A: the two categorical variables have a relationship. The two variables are not independent.

```
var1 <- c("Male", "Female", "Male", "Female", "Male")
var2 <- c("chocolate", "strawberry", "strawberry", "strawberry", "chocolate")
data <- data.frame(var1, var2)
```

```
data.table <- table(data$var1, data$var2)
data.table > chisq.test(data.table)
```

20.6 ANOVA

ANOVA is the process of testing the means of two or more groups. ANOVA also checks the impact of factors by comparing the means of different samples.

In **ANOVA**, you use two kinds of means:

- Sample means.
- Grand mean (the mean of all of the samples' means).

Hypothesis: - H_0: $\mu_1 = \mu_2 = ... = \mu_L$; the sample means are equal or do not have significant differences. - H_A: $\mu_1 \quad \mu_m$; is when the sample means are not equal.

You assume that the variables are sampled, independent, and selected or sampled from a population that is normally distributed with unknown but equal variances.

20.6.1 Between Group Variability

The distribution of two samples, when they overlap, their means are not significantly different. Hence, the difference between their individual mean and the grand mean is not significantly different.

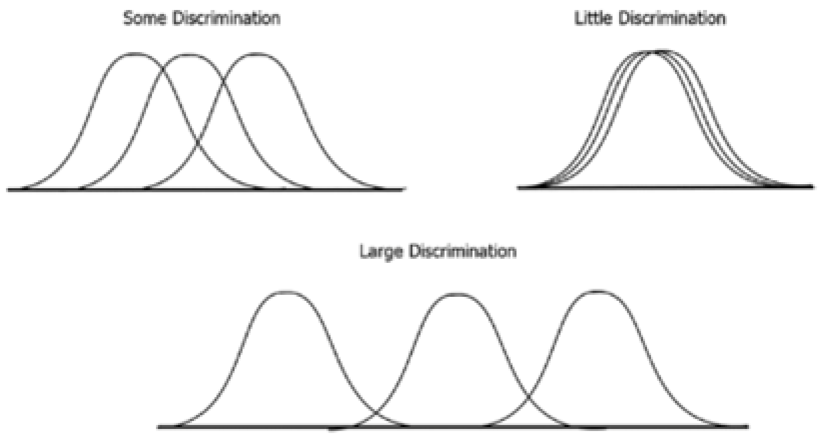


Figure 20.1: Figure 6: Between group variability

This variability is called the **between-group variability**, which refers to the variations between the distributions of the groups or levels.

20.6.2 Within Group Variability

For the following distributions of samples, as their variance increases, they overlap each other and become part of a population.

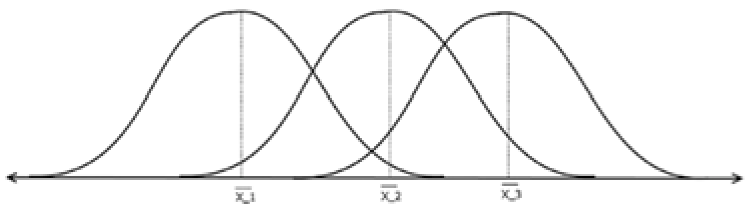


Figure 20.2: Figure 7: Within group variability

The **F-statistics** are the measures if the means of samples are significantly different. The lower the F-statistics, the more the means are equal, so you cannot reject the null hypothesis.

20.6.3 One-Way ANOVA

One-way ANOVA is used when you have only one independent variable.

```r
library(graphics)
set.seed(123)
var1 <- rnorm(12, 2, 1)
var2 <- c("B", "B", "B", "B", "C", "C", "C", "C", "C", "D", "D", "B")
data <- data.frame(var1, var2)
fit <- aov(data$var1 ~ data$var2, data = data)
fit
```

```
Call:
   aov(formula = data$var1 ~ data$var2, data = data)

Terms:
                data$var2 Residuals
Sum of Squares   0.162695  9.255706
Deg. of Freedom         2         9

Residual standard error: 1.014106
Estimated effects may be unbalanced
```

```r
summary(fit)
```

```
            Df Sum Sq Mean Sq F value Pr(>F)
data$var2    2  0.163  0.0813   0.079  0.925
Residuals    9  9.256  1.0284
```

20.6.4 Two-Way ANOVA

Two-way ANOVA is used when you have two independent variables. (continuar en el ejemplo anterior):

```
var3 <- c("D", "D", "D", "D", "E", "E", "E", "E", "E", "F", "F", "F")
data <- data.frame(var1, var2, var3)
fit <- aov(data$var1 ~ data$var2 + data$var3, data=data)
fit
```

```
Call:
   aov(formula = data$var1 ~ data$var2 + data$var3, data = data)

Terms:
                data$var2 data$var3 Residuals
Sum of Squares   0.162695  0.018042  9.237664
Deg. of Freedom         2         1         8

Residual standard error: 1.074573
1 out of 5 effects not estimable
Estimated effects may be unbalanced
```

```
summary(fit)
```

```
            Df Sum Sq Mean Sq F value Pr(>F)
data$var2    2  0.163  0.0813   0.070  0.933
data$var3    1  0.018  0.0180   0.016  0.904
Residuals    8  9.238  1.1547
```

```
## var1 does not depend on var2's mean and var3's mean
```

20.6.5 MANOVA

The multivariate analysis of variance is when there are multiple response variables that you want to test.

Example:

```
res <- manova(cbind(iris$Sepal.Length, iris$Petal.Length) ~ iris$Species, data=iris)
summary(res)
```

```
              Df Pillai approx F num Df den Df    Pr(>F)    
iris$Species   2 0.9885   71.829      4    294 < 2.2e-16 ***
Residuals    147                                            
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
summary.aov(res)
```

```
 Response 1 :
              Df Sum Sq Mean Sq F value    Pr(>F)    
iris$Species   2 63.212  31.606  119.26 < 2.2e-16 ***
Residuals    147 38.956   0.265                      
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

 Response 2 :
              Df Sum Sq Mean Sq F value    Pr(>F)    
iris$Species   2 437.10 218.551  1180.2 < 2.2e-16 ***
Residuals    147  27.22   0.185                      
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
## Hence, you have two response variables, Sepal.Length and Petal.Length
```

The **p-value** is 2.2e-16, which is less than 0.05. Hence, you reject the null hypothesis

20.7 Nonparametric Test

The **nonparametric test** is a test that does not require the variable and sample to be normally distributed.

You use nonparametric tests when you do not have normally distributed data and the sample data is big.

Table 20.1: Table 3: Types of nonparametric tests

Nonparametric Test	Function	Method replaced
Wilcoxon Signed Rank Test	`wilcox.test(data[,1], mu=0, alternatives="two.sided")`	one-sample
t-testundefined		
Wilcoxon-Mann-Whitney Test	`wilcox.test(data[,1], data[,2], correct=FALSE)`	substitute
to the two-sample t-test		
Kruskal-Wallis Test	`kruskal.test(airquality$Ozone ~ airquality$Month)`	one-way

20.7.1 Wilcoxon Signed Rank Test

The **Wilcoxon signed rank test** is used to replace the **one-sample t-test**.

Hypothesis:

- H_0: $\mu_1 = \mu_o$; the null hypothesis is that the population median has the specified value of μ_0
- H_a: $\mu_1 \quad \mu_o$

To use the **Wilcoxon signed rank test** in R, you can first generate the data set using **random.org packages**, so that the variables are not normally distributed.

Example:

```
install.packages("random")
library(random)
var1 <- randomNumbers(n=100, min=1, max=1000, col=1)
var2 <- randomNumbers(n=100, min=1, max=1000, col=1)
var3 <- randomNumbers(n=100, min=1, max=1000, col=1)
data <- data.frame(var1[,1], var2[,1], var3[,1])
```

```r
wilcox.test(data[,1], mu=0, alternatives="two.sided")
```

20.7.2 Wilcoxon-Mann-Whitney Test

The **Wilcoxon-Mann-Whitney test** is a nonparametric test to compare two samples. It is a powerful substitute to the two-sample t-test.

To use the **Wilcoxon-Matt-Whitney test** (or the Wilcoxon rank sum test or the Mann-Whitney test) in R, you can use the `wilcox.test()` function:

```r
wilcox.test(data[,1], data[,2], correct=FALSE)
```

There are not significant differences in the median for first variable median and second variable median.

20.7.3 Kruskal-Wallis Test

The **Kruskal-Wallis test** is a nonparametric test that is an extension of the **Mann-Whitney U test** for three or more samples.

The test requires samples to be identically distributed.

Kruskal-Wallis is an alternative to **one-way ANOVA**.

The **Kruskal-Wallis** test tests the differences between scores of k independent samples of unequal sizes with the ith sample containing li rows:

- H_0: $\mu_o = \mu_1 = \mu_2 = ... = \mu_k$; The null hypothesis is that all the medians are the same.
- H_a: $\mu_1 \quad \mu_k$; The alternate hypothesis is that at least one median is different.

```r
kruskal.test(airquality$Ozone ~ airquality$Month)
```

```

	Kruskal-Wallis rank sum test

data:  airquality$Ozone by airquality$Month
Kruskal-Wallis chi-squared = 29.267, df = 4, p-value = 6.901e-06
```

21 Regression Analysis

Regression analysis is a form of predictive modelling techniques that identifies the relationships between dependent and independent variables(s). The technique is used to find causal effect relationships between variables.

Regression analysis identifies the significant relationships between dependent and independent variables and the strength of the impact of multiple independent variables on dependent variables.

21.1 Linear Regressions

The linear regression equation is $y = b_0 + b_1 x$, where **y** is the dependent variable, **x** is the independent variable, b_0 is the intercept, and b_1 is the slope.

To use linear regression in R, you use the `lm()` function:

```r
set.seed(123)
x <- rnorm(100, mean=1, sd=1)
y <- rnorm(100, mean=2, sd=2)
data <- data.frame(x, y);
mod <- lm(data$y ~ data$x, data=data)
mod
```

```

Call:
lm(formula = data$y ~ data$x, data = data)

Coefficients:
(Intercept)       data$x
     1.8993      -0.1049
```

```r
summary(mod)
```

```
Call:
lm(formula = data$y ~ data$x, data = data)

Residuals:
   Min     1Q Median     3Q    Max 
-3.815 -1.367 -0.175  1.161  6.581 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   1.8993     0.3033   6.261 1.01e-08 ***
data$x       -0.1049     0.2138  -0.491    0.625    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.941 on 98 degrees of freedom
Multiple R-squared:  0.002453,  Adjusted R-squared:  -0.007726 
F-statistic: 0.241 on 1 and 98 DF,  p-value: 0.6246
```

When the **p-value** is less than 0.05, the model is significant:

- H_0: Coefficient associated with the variable is equal to zero
- H_a: Coefficient is not equal to zero (there is a relationship)

Furthermore:

- The higher the **R-squared** and the **adjusted R-squared**, the better the linear model.
- The lower the **standard error**, the better the model

21.2 Multiple Linear Regressions

Multiple linear regression is used when you have more than one independent variable.

The equation of a multiple linear regression is:

$y = b_0 + b_1x_1 + b_2x_2 + ... + b_kx_k + \epsilon$

When you have n observations or rows in the data set, you have the following model:

Using a matrix, you can represent the equations as: $y = Xb + \epsilon$

To calculate the coefficients: ^b = (X' X)-1 X' y

$$
\begin{aligned}
y_1 &= b_0 + b_1 x_{11} + b_2 x_{12} + \ldots + b_k x_{1k} + \epsilon_1 \\
y_2 &= b_0 + b_1 x_{21} + b_2 x_{22} + \ldots + b_k x_{2k} + \epsilon_2 \\
y_3 &= b_0 + b_1 x_{31} + b_2 x_{32} + \ldots + b_k x_{3k} + \epsilon_3 \\
&\ldots \\
y_n &= b_0 + b_1 x_{n1} + b_2 x_{n2} + \ldots + b_k x_{nk} + \epsilon_n
\end{aligned}
$$

Figure 21.1: Figure 4: Model for multiple regressions

$$
y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{bmatrix}
\quad
X = \begin{bmatrix}
1 & x_{11} & x_{12} & . & . & . & x_{1k} \\
1 & x_{21} & x_{22} & . & . & . & x_{2k} \\
. & . & . & & & & . \\
. & . & . & & & & . \\
. & . & . & & & & . \\
1 & x_{n1} & x_{n2} & . & . & . & x_{nk}
\end{bmatrix}
$$

$$
b = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ b_n \end{bmatrix}
\quad and \quad
\epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ . \\ . \\ . \\ \epsilon_n \end{bmatrix}
$$

Figure 21.2: Figure 5: Representation of equations as a matrix

```r
set.seed(123)
x <- rnorm(100, mean=1, sd=1)
x2 <- rnorm(100, mean=2, sd=5)
y <- rnorm(100, mean=2, sd=2)
data <- data.frame(x, x2, y)
mod <- lm(data$y ~ data$x + data$x2, data=data)
mod
```

```

Call:
lm(formula = data$y ~ data$x + data$x2, data = data)

Coefficients:
(Intercept)       data$x      data$x2  
   2.517425    -0.266343     0.009525  
```

```r
summary(mod)
```

```

Call:
lm(formula = data$y ~ data$x + data$x2, data = data)

Residuals:
    Min      1Q  Median      3Q     Max 
-3.7460 -1.3215 -0.2489  1.2427  4.1597 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept)  2.517425   0.305233   8.248 7.97e-13 ***
data$x      -0.266343   0.209739  -1.270    0.207    
data$x2      0.009525   0.039598   0.241    0.810    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.903 on 97 degrees of freedom
Multiple R-squared:  0.01727,	Adjusted R-squared:  -0.00299 
F-statistic: 0.8524 on 2 and 97 DF,  p-value: 0.4295
```

Part V

Part 4: Graphics

22 Graphs

22.1 Principles of Graphics

1. Show comparisons

You should always be comparing at least two things.

2. Show causality, mechanism, explanation, systematic structure

Generally, it's difficult to prove that one thing causes another thing even with the most carefully collected data. But it's still often useful for your data graphics to indicate what you are thinking about in terms of cause.

3. Show multivariate data

The point is that data graphics should attempt to show this information as much as possible, rather than reduce things down to one or two features that we can plot on a page… From the plot it seems that there is a slight negative relationship between the two variables….This example illustrates just one of many reasons why it can be useful to plot multivariate data and to show as many features as intelligently possible. In some cases, you may uncover unexpected relationships depending on how they are plotted or visualized.

4. Integrate evidence

Data graphics should make use of many modes of data presentation simultaneously, not just the ones that are familiar to you or that the software can handle. One should never let the tools available drive the analysis; one should integrate as much evidence as possible on to a graphic as possible.

5. Describe and document the evidence

Data graphics should be appropriately documented with labels, scales, and sources. A general rule for me is that a data graphic should tell a complete story all by itself. Is there enough information on that graphic for the person to get the story? While it is certainly possible to be too detailed, I tend to err on the side of more information rather than less.

6. Content, Content, Content

Analytical presentations ultimately stand or fall depending on the quality, relevance, and integrity of their content. This includes the question being asked and the evidence presented in favor of certain hypotheses. Starting with a good question, developing a sound approach, and only presenting information that is necessary for answering that question, is essential to every data graphic.

22.2 Types of Graphs

Visualizing the data via graphics can be important at the beginning stages of data analysis to understand basic properties of the data, to find simple patterns in data, and to suggest possible modeling strategies. In later stages of an analysis, graphics can be used to "debug" an analysis, if an unexpected (but not necessarily wrong) result occurs, or ultimately, to communicate your findings to others.

We will make a **distinction between exploratory graphs and final graphs**:

Exploratory graphs are usually made very quickly and a lot of them are made in the process of checking out the data, developing a personal understanding of the data and to prioritize tasks for follow up. Details like axis orientation or legends, while present, are generally cleaned up and prettified if the graph is going to be used for communication later.

22.2.1 One-dimension graphs

Graph	Code	Use
Five-number summary	`fivenum()`, `summary()`	
Boxplots	`boxplot(pollution$pm25, col = "blue")`	Commonly plot outliers that go beyond the bulk of the data
Barplot	`table(pollution$region) %>% barplot(col = "wheat")`	For visualizing categorical data, with the number of entries for each category being proportional to the height of the bar
Histogram	`hist(pollution$pm25, col = "green", breaks = 100)` `rug(pollution$pm25)` `abline(v = 12, lwd = 2)` `abline(v = median(pollution$pm25), col = "magenta", lwd = 4)`	Check skewness of the data, symmetry, multi-modality, and other features

Graph	Code	Use

Density plot	`a <- density(airquality$Ozone, na.rm = TRUE)`
	`plot(a)`
		Computes a non-parametric estimate of the distribution of a variables

22.2.2 Multi-dimension graphs

Graph	Code	Use

Multiple 1-D plots	`boxplot(pm25 ~ region, data = pollution, col = "red")`
	`par(mfrow = c(2, 1), mar = c(4, 4, 2, 1))`
	`hist(subset(pollution, region == "east")$pm25, col = "green")`
	`with(subset(airquality, Month == 5), points(Wind, Ozone, col = "blue"))`
	`hist(subset(pollution, region == "west")$pm25, col = "green")> boxplot(pm25 ~ region, data = pollution, col = "red")`
	`par(mfrow = c(2, 1), mar = c(4, 4, 2, 1))`
	`hist(subset(pollution, region == "east")$pm25, col = "green")`
	`with(subset(airquality, Month == 5), points(Wind, Ozone, col = "blue"))`
	`hist(subset(pollution, region == "west")$pm25, col = "green")`
		For seeing the relationship between two variables, especially when one is naturally **categorical**

Graph	Code	Use
Scatterplots	`with(pollution, plot(latitude, pm25, col = region)) > abline(h = 12, lwd = 2, lty = 2) levels(pollution$region) par(mfrow = c(2, 1), mar = c(4, 4, 2, 1)) hist(subset(pollution, region == "east")$pm25, col = "green") hist(subset(pollution, region == "west")$pm25, col = "green")`	Visualizing two continuous variables
Scatter plot matrix	`pairs(~var1+var2+var3+var4+var5, data=data, main="scatterplot matrix")`	It is used to find the correlation between a variable and other variables, and to select the important variables, which is also known as variable selection
Smooth scatterplots	`with(airquality, {plot(Temp, Ozone), lines(loess.smooth(Temp, Ozone))`	Similar to scatterplots but rather plots a 2-D histogram. Can be useful for scatterplots with many many data points

22.3 The Base Plotting System

The **base plotting system** is the original plotting system for R. The basic model is sometimes referred to as the "artist's palette" model. The idea is you start with blank canvas and build up from there.

You will typically start with a plot function (or similar plot creating function) to initiate a plot and then annotate the plot. If you don't have a completely well-formed idea of how you want your data to look, you'll often start by "throwing some data on the page" and then slowly add more information to it as our thought process evolves.

There are a few **downsides** though:

- You can't go backwards once the plot has started.
- While the base plotting system is nice in that it gives you the flexibility to specify these kinds of details to painstaking accuracy, sometimes it would be nice if the system could just figure it out for you.

- It's difficult to describe or translate a plot to others because there's no clear graphical language or grammar that can be used to communicate what you've done.

22.3.1 How to Create a Base Plot?

1. First, you must read the data into R with `read.csv()`. For example, the *avgpm25.csv* dataset contains the annual mean PM2.5 averaged over the period 2008 through 2010

```
class <- c("numeric", "character", "factor", "numeric", "numeric")
pollution <- read.csv("data/avgpm25.csv", colClasses = class)
```

2. Explicitly launch a graphics device.
3. Call a plotting function to make a plot (Note: if you are using a file device, no plot will appear on the screen).
4. Annotate the plot if necessary.
5. Explicitly close graphics device with dev.off() (this is very important!).

Example:

```
# Open PDF device; create 'myplot.pdf' in my working directory
pdf(file = "myplot.pdf", width = 4, height = 3) # The height and width arguments are in un
# Create plot and send to a file (no plot appears on screen)
with(faithful, plot(eruptions, waiting))
# Annotate plot; still nothing on screen
title(main = "Old Faithful Geyser data")
# Close the PDF file device
dev.off()
# Now you can view the file 'myplot.pdf' on your computer
```

Let's further detail each of the steps involved in creating a **base plot**.

22.3.2 Graphics Devices

We can think of a **graphics device** as being a platform upon which the plot is created. If we create a plot, then a default graphics device is automatically opened for the plot to appear upon. In other words, when you make a plot in R, it has to be "sent" to a specific **graphics device**.

The most common place for a plot to be "sent" is the **screen** device. Functions like `plot()` in base, `xyplot()` in lattice, or `qplot` in ggplot2 will default to sending a plot to the screen device. Therefore, when making a plot, you need to consider how the plot will be used to determine what device the plot should be sent to.

The **list of devices** supported by your installation of R is found in `?Devices`.

- Using the commands `pdf`, `postscript`, `jpeg`, `png`, or `bmp`, we can also produce graphics in the formats that correspond to these names.
- The `jpeg`, `png`, and `bmp` graphics are all **raster-style** graphics, which may translate poorly when included in a document.
- In contrast, the `pdf`, `postscript`, and `windows metafile` (win.metafile, available on Windows) formats allow for **vector-style** graphics, which are scaleable, and better suited to integration in documents.

Vector	Bitmap
Good for line drawings and plots with solid colors using a modest number of points	good for plots with a large number of points, natural scenes or web-based plots
pdf (line-type graphics, resizes well, usually portable)	png (good for line drawings or images with solid colors)
svg (XML-based scalable vector graphics; supports animation and interactivity)	jpeg (good for plotting many many many points, does not resize well)
win.metafile	tiff
postscript	bmp

It is possible to **open multiple graphics devices** (screen, file, or both), for example when viewing multiple plots at once. Plotting can only occur on one graphics device at a time, though.

One way of having more than one plot visible is to open additional graphics devices. In a Windows environment this is done by using the command `windows()` before each additional plot.

To create a graphics device without a plot, we call the function that is specific to our operating system (that is, `windows` for Windows, `quartz` for Mac, and `X11` for Unix).

The **currently active graphics device** can be found by calling `dev.cur()`

Every open graphics device is assigned an integer starting with 2 (there is no graphics device 1). You can change the active graphics device with `dev.set(<integer>)`.

22.3.3 Graphics Parameters

Graphics parameters control how output appears on graphics devices. To get a complete list with their current values, type `par()`. Some of the parameters mentioned above, namely `pch`, lwd and col, are examples of graphics parameters. To get the value of a specific parameter, for example pch, type par("pch").

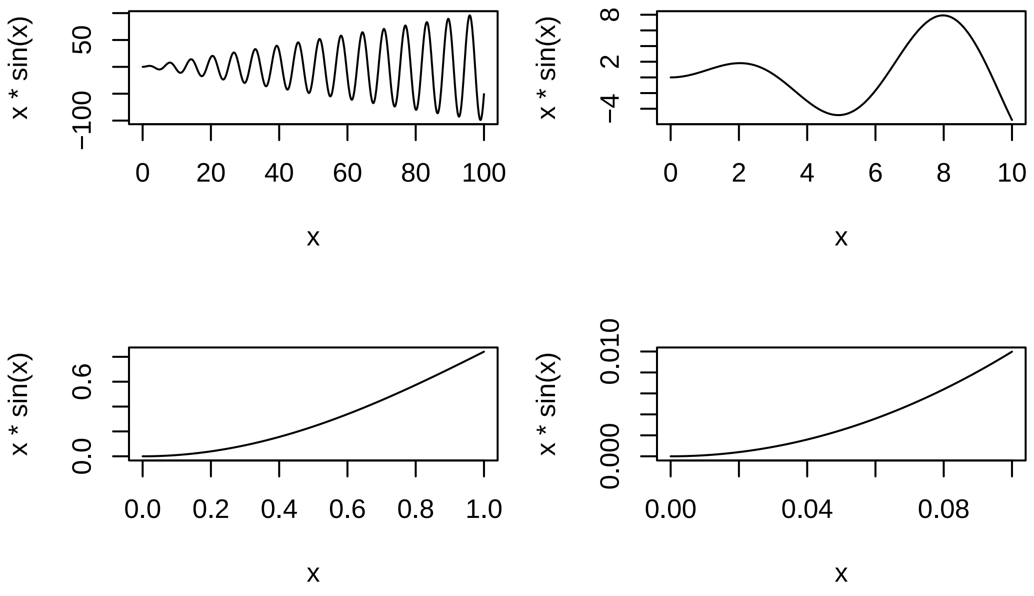
Some **graphics parameters** can apply to **one or more plots**, and others only make sense when applied to graphics devices. For example, to change the symbol for a single plot, we could include the argument pch = 2 in the call to the plot function. However, we could also make this change for all graphics that are produced on the device.

To change a graphics parameter for the graphics device, we use the par command. These are some examples:

- par(mfrow = c(a,b)) where a and b are integers, will create a matrix of plots on one page, with a rows and b columns. These will be filled by rows; use mfcol if you wish to fill them by columns.

  Example:

```
par(mfrow = c(2, 2), mar=c(5, 4, 2, 1))
curve(x*sin(x), from = 0, to = 100, n = 1001)
curve(x*sin(x), from = 0, to = 10, n = 1001)
curve(x*sin(x), from = 0, to = 1, n = 1001)
curve(x*sin(x), from = 0, to = 0.1, n = 1001)
```



- `par(mar = c(bottom, left, top, right))` will create space around each plot, in which to write axis labels and titles. Measurements are in units of character widths.
- `par(oma = c(bottom, left, top, right))` will create space around the matrix of plots (an outer margin). Measurements are in units of character widths.
- `par(las = 1)` rotates labels on the y-axis to be horizontal rather than vertical.
- `par(pty = "s")` forces the plot shape to be square. The alternative is that the plot shape is mutable, which is the default, and corresponds to `pty ="m"`.

22.3.4 Make a Plot with `plot(x,y)`

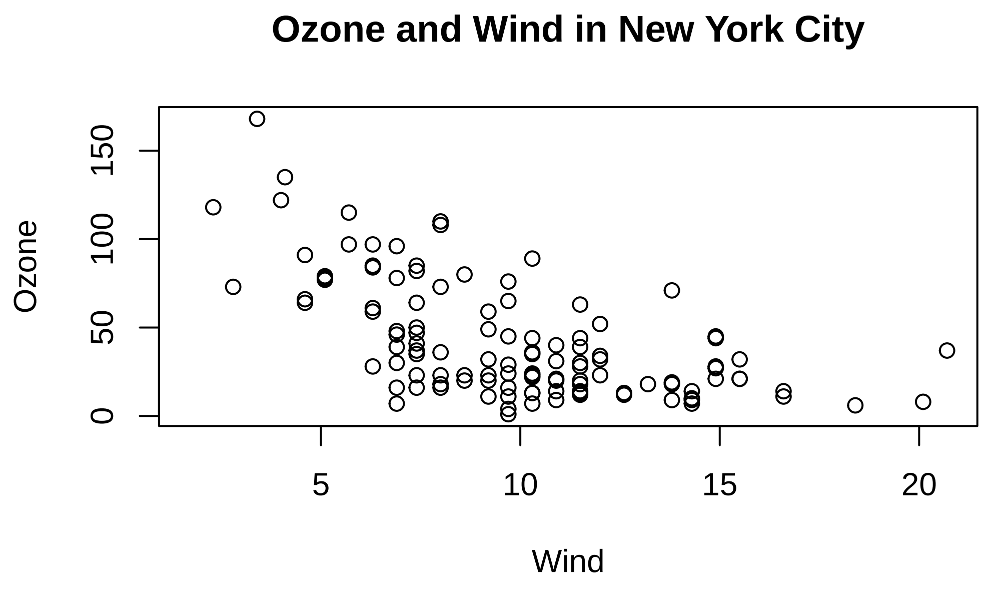
`plot(x, y)` is used to plot one vector against another, with the `x` values on the x-axis and the `y` values on the y-axis.

The `plot` command offers a wide variety of **options for customizing** the graphic. Each of the following arguments can be used within the plot statement, singly or together, separated by commas

- `type = "p"`: Determines the type of plot, with options:
  - "p" for points (the default);
  - "l" for lines;
  - "b" for both, with gaps in the lines for the points; …

  Example:

  ```
  with(airquality, plot(Wind, Ozone,
      main = "Ozone and Wind in New York City",
      type = "p")
  )
  ```



- `points(x, y)`: To add points (`x[1]`, `y[1]`), (`x[2]`, `y[2]`), … to the current plot.
- `lines(x, y)`: To add lines.
- `abline(v = xpos)` and `abline(h = ypos)`: to draw vertical or horizontal lines. - `col`: Both points and lines take the optional input from `col` (e.g. "red", "blue", etc.). The complete list of available colours can be obtained by the `colours` function (or `colors`).

  Example: # Plot with a Regression Line

  - First make the plot (as above).
  - Fit a simple linear regression model using the `lm()` function.
  - Take the output of `lm()` and pass it to the `abline()` function which automatically takes the information from the model object and calculates the corresponding regression line
- `text(x, y, labels)`: To add the text labels [i] at the point (`x[i]`, `y[i]`). The optional input `pos` is used to indicate where to position the labels in relation to the points.
- `title(text)`: To add a title, where `text` is a character string.
- `main = "Plot title goes in here"`: provides the plot title.
- `xlab = " "` / `ylab = " "`: To add axis labels.

Note: In a call to plot, the arguments for **main**, **sub**, and **xlab** and **ylab** can be character strings or expressions that contain a mathematical expression. For **mathematical typesetting** you can use the functions `expression`(require graphics) and `bquote`(base):

Example:

```
...
xlab = expression(alpha),
ylab = expression(100 %*% (alpha^3 - alpha^2) + 15),
# Mix of mathematical expressions and character strings into a single expression by u
main = expression(paste("Function: ",
   f(alpha) == 100 %*% (alpha^3 - alpha^2) + 15)),
...
```

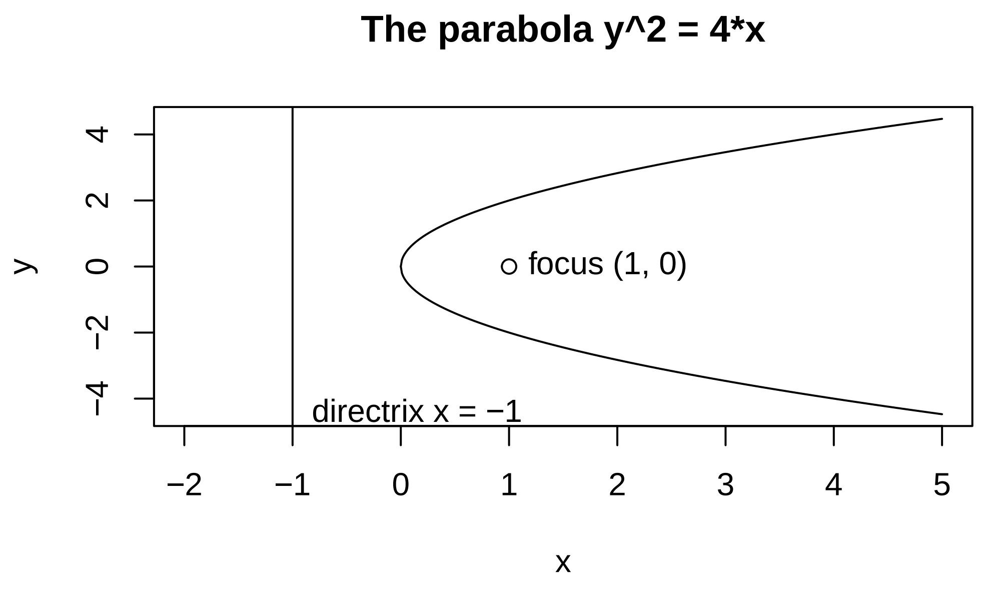
`xlab = expression(alpha)` tells R to interpret alpha in the context of the MML (mathematical markup language), producing an   as the label for the x-axis.

- `pch = k`: Determines the shape of points, with k taking a value from 1 to 25.
- `lwd = 1`: line width, default 1.
- `xlim = c(a,b)`/ ‘`ylim = c(a,b)`: Will set the lower and upper limits of the x-axis/y-axis to be a and b, respectively.

As an **example** we plot part of the parabola $y^2 = 4x$, as well as its focus and directrix. We make use of the surprisingly useful input `type = "n"`, which results in the graph dimensions being established, and the axes being drawn, but nothing else.

Example:

```
x <- seq(0, 5, by = 0.01)
y.upper <- 2*sqrt(x)
y.lower <- -2*sqrt(x)
y.max <- max(y.upper)
y.min <- min(y.lower)
plot(c(-2, 5), c(y.min, y.max), type = "n", xlab = "x", ylab = "y")
lines(x, y.upper)
lines(x, y.lower)
abline(v=-1)
points(1, 0)
text(1, 0, "focus (1, 0)", pos=4)
text(-1, y.min, "directrix x = -1", pos = 4)
title("The parabola y^2 = 4*x")
```



22.3.5 Augmenting a Plot

A traditional plot can be augmented using any of a number of different tools after its creation. A number of these different steps are detailed below:

1. Start by creating the **plot object**, which sets up the dimensions of the space, but omit any plot objects for the moment:

   Example:

   ```
   opar1 <- par(las = 1, mar=c(4,4,3,2))
   plot(ufc$dbh.cm, ufc$height.m, axes=FALSE,
       xlab="",
       ylab="",
       type="n"
   )
   ```

2. Next, we **add the points**. Here we use different colours and symbols for different heights of trees: those that are realistic, and those that are not, which may reflect measurement errors. We use the vectorised `ifelse()` function:

   Example:

```r
points(ufc$dbh.cm, ufc$height.m,
  col = ifelse(ufc$height.m > 4.9, "darkseagreen4", "red"),
  pch = ifelse(ufc$height.m > 4.9, 1, 3)
)
```

3. Then we **add axes**. The following are the simplest possible calls. We can also control the locations of the tickmarks, and their labels; we can overlay different axes, change colour, and so on. `?axis` provides the details:

   Example:

   ```r
   axis(1)
   axis(2)
   ```

4. We can next **add axis labels** using margin text (switching back to vertical direction for the y-axis text):

   Example:

   ```r
   opar2 <- par(las=0)
   mtext("Diameter (cm)", side=1, line=3)
   mtext("Height (m)", side=2, line=3)
   ```

5. **Wrap the plot** in the traditional frame. As before, we can opt to use different line types and different colours:

   Example:

   ```r
   box()
   ```

6. Finally, we **add a legend**:

   Example:

   ```r
   legend(x = 60, y = 15, c("Normal trees", "A weird tree"),
       col=c("darkseagreen3", "red"),
       pch=c(1, 3),
       bty="n"
   )
   ```

   Note the first two arguments: the location of the legend can also be expressed relative to the graph components, for example, by "bottomright".

7. If we wish, we can return the graphics environment to a previous state:

```r
par(opar1)
```

Check out the **playwith package**, which provides interaction with graphical objects at a level unattainable in base R.

22.3.6 Color in Plots

Typically we add color to a plot, not to improve its artistic value, but to add another dimension to the visualization. It makes sense that the range and palette of colors you use will depend on the kind of data you are plotting. Careful choices of plotting color can have an impact on how people interpret your data and draw conclusions from them.

The function `colors()` lists the names of (657) colors you can use in any plotting function. Typically, you would specify the color in a (base) plotting function via the **col** argument.

The **grDevices package** has two functions, they differ only in the type of object that they return:

- **colorRamp**: Take a palette of colors and return a function that takes values between 0 and 1.

  ```r
  pal <- colorRamp(c("red", "blue"))
  pal(0)
  ```

  The numbers in the matrix will range from 0 to 255 and indicate the quantities of red, green, and blue (RGB) in columns 1, 2, and 3 respectively. there are over 16 million colors that can be expressed in this way.

  The idea here is that you do not have to provide just two colors in your initial color palette; you can start with multiple colors and `colorRamp()` will interpolate between all of them.

- **colorRampPalette**: Takes a palette of colors and returns a function that takes integer arguments and returns a vector of colors interpolating the palette (like `heat.colors()` or `topo.colors()`).

  ```r
  pal <- colorRampPalette(c("red", "yellow"))
  pal(3)
  ```

  Returns 3 colors in between red and yellow. Note that the colors are represented as hexadecimal strings.

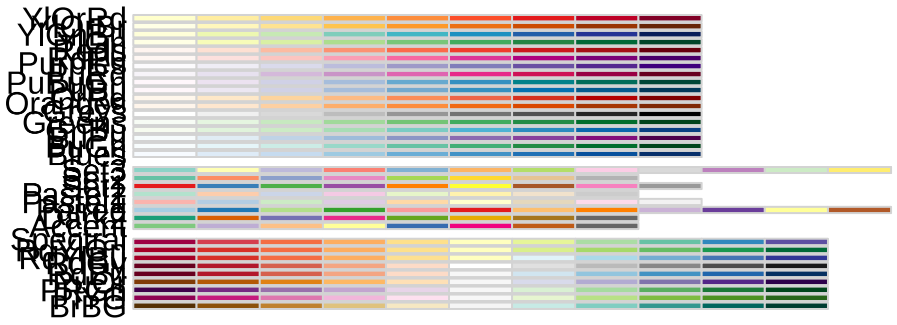
- Note that the `rgb()` function can be used to produce any color via red, green, blue proportions and return a hexadecimal representation:

```
rgb(0, 0, 234, maxColorValue = 255)
```

- Part of the art of creating good color schemes in data graphics is to start with an appropriate color palette that you can then interpolate with a function like `colorRamp()` or `colorRampPalette()`.

For improved color palettes you can use the **RColorBrewer package**. Here is a display of all the color palettes available from this package:

```
library(RColorBrewer)
display.brewer.all()
```



- The `brewer.pal()` function creates nice looking color palettes especially for thematic maps:

```
library(RColorBrewer)
(cols <- brewer.pal(3, "BuGn"))
```

```
[1] "#E5F5F9" "#99D8C9" "#2CA25F"
```

These three colors make up your initial palette. Then you can pass them to `colorRampPalette()` to create my interpolating function.

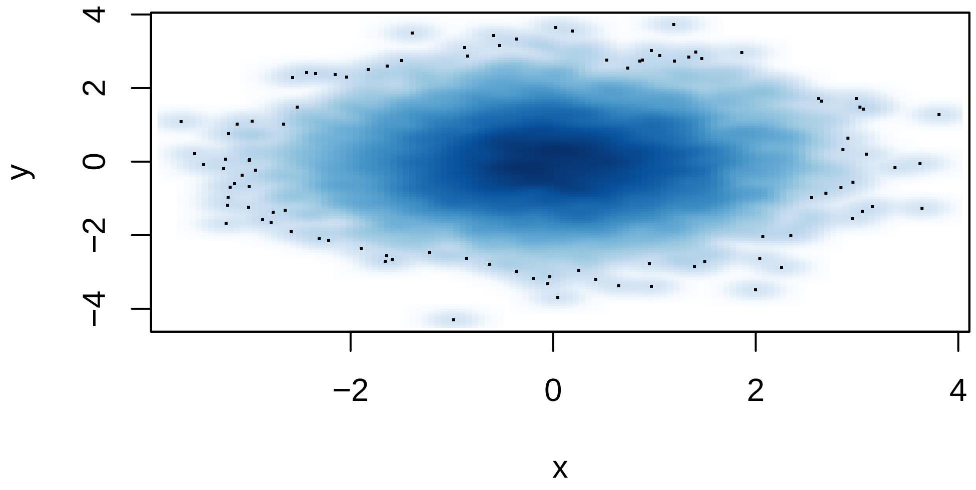
```
pal <- colorRampPalette(cols)
```

Now you can plot your data using this newly created color (ramp) palette:

```
image(volcano, col = pal(20))
```

- The `smoothScatter()` function is very useful for making **scatterplots** of very large datasets:

```
set.seed(1)
x <- rnorm(10000)
y <- rnorm(10000)
smoothScatter(x, y)
```



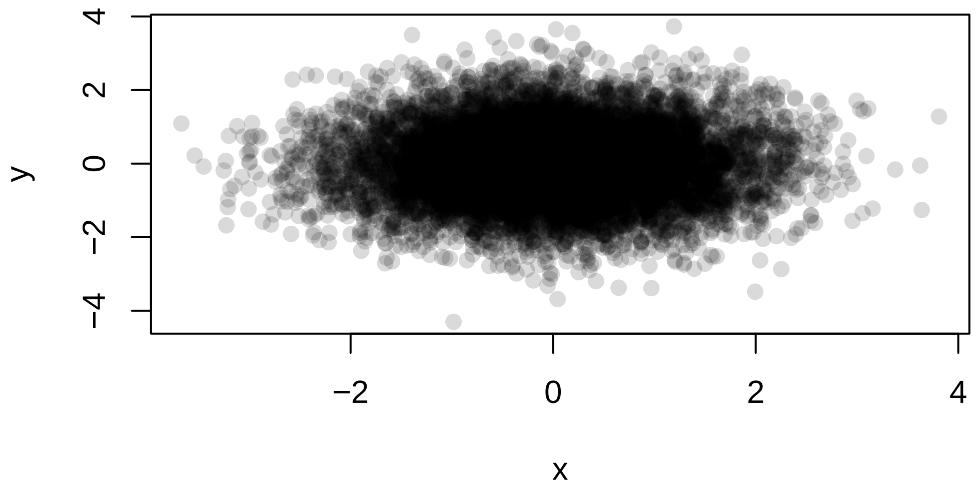
- **Adding transparency**: Color transparency can be added via the alpha parameter to `rgb()` to produce color specifications with varying levels of transparency.

```
rgb(1, 0, 0, 0.1)
```

```
[1] "#FF00001A"
```

Transparency can be useful when you have plots with a high density of points or lines. If you add some transparency to the black circles, you can get a better sense of the varying density of the points in the plot.

```
# x, y from the previous example
plot(x, y, pch = 19, col = rgb(0, 0, 0, 0.15))
```



22.3.7 Copying Plots

Note that copying a plot is not an exact operation, so the result may not be identical to the original:

```
## Copy my plot to a PNG file  (follows the previous code from pdf creation)
dev.copy(png, file = "geyserplot.png")

## Don't forget to close the PNG device!
dev.off()
```

22.3.8 Complementary Packages

- **graphics**: contains plotting functions for the "base" graphing systems, including `plot`, `hist`, `boxplot` and many others
- **grdevices**: contains all the code implementing the various graphics devices, including X11, PDF, PostScript, PNG, etc.

22.4 The Lattice System

Trellis graphics are a data visualisation framework developed at the Bell Labs, which have been implemented in R as the **lattice package**.

Trellis graphics are a set of techniques for displaying multidimensional data. They allow great flexibility for producing **conditioning plots**; that is, plots obtained by conditioning on the value of one of the variables.

To use Trellis graphics you must first you must load the **lattice package** with the library function: `library(lattice)`.

22.4.1 When to Use Trellis Plots

- **Lattice plots** tend to be most useful for **conditioning types of plots**, i.e. looking at how y changes with x across levels of z. These types of plots are useful for looking at **multidimensional data** and often allow you to squeeze a lot of information into a single window or page.
- With the lattice system, plots are created with a single function call, such as `xyplot()` or `bwplot()`.
- Note that there is no real distinction between the functions that create or initiate plots and the functions that annotate plots because **it all happens at once**.
- Another difference from base plotting is that things like **margins and spacing** are set automatically. This is possible because entire plot is specified at once via a single function call.
- The notion of **panels** comes up a lot with lattice plots because you typically have many panels in a lattice plot (each panel typically represents a condition, like "region").
- Once a plot is created, you **cannot "add"** to the plot (but of course you can just make it again with modifications).

22.4.2 How to Use Lattice

To illustrate the use of the **lattice package** we use the *ufc dataset*, where *dbh (diameter at breast height)* and *height* vary by *species*. That is, the plots are conditioned on the value of the variable species.

Example:

```
# These plots display a distribution of values taken on by the variable dbh, divided up ac
densityplot(~ dbh.cm | species, data = ufc) # A density plot
bwplot(~ dbh.cm | species, data = ufc) # # A box and whiskers plot
histogram(~ dbh.cm | species, data = ufc) # A histogram
# We plot height as a function of dbh.
xyplot(height.m ~ dbh.cm | species, data = ufc) # A scatterplot
```

All four commands require a formula object, which is described using ~and |.

- If a **dataframe** is passed to the function, using the argument `data`, then the column names of the dataframe can be used for describing the model.
- We interpret `y ~ x | a` as saying we want `y` as a function of `x`, divided up by the different levels of `a`.
- If ‘`a` is not a factor then a factor will be created by coercion.
- If we are just interested in `x` we still include the ~ symbol, so that R knows that we are specifying a model.
- If we wish to provide within-panel conditioning on a second variable, then we use the group argument.

In order to display **numerous lattice objects** on a graphics device, we call the `print` function with the `split` and more arguments.

Example:

```
# Place a lattice object (called my.lat) in the top-right corner of a 3-row, 2-column grap
print(my.lat, split = c(2,1,2,3), more = TRUE)
```

See `?print.trellis` for further details.

Graphics produced by lattice are **highly customisable**.

Example:

```
xyplot(height.m ~ dbh.cm | species,
    data = ufc,
    subset = species %in% list("WC", "GF"),
    panel = function(x, y, ...) {
        panel.xyplot(x, y, ...),
        panel.abline(lm(y~x), ...)
    },
    xlab = "Diameter (cm)",
    ylab = "Height (m)"
    )
```

Note that the `subset` argument is a logical expression or a vector of integers, and is used to restrict the data that are plotted. We can also change the order of the panels using the `index.cond` argument (see `?xyplot` for more details). The panels are plotted bottom left to top right by default, or top left to bottom right if the argument `as.table = TRUE` is supplied.

The `panel` argument accepts a function, the purpose of which is to control the appearance of the plot in each panel. The panel function should have one input argument for each variable in the model, not including the conditioning variable.

22.5 3D Plots

R provides considerable functionality for constructing **3D graphics**, using either the **base** graphics engine or the **lattice package**. However, its is recommended to use the **lattice package**, because the data can be supplied to the lattice functions in a familiar structure: observations in rows and variables in columns, unlike that required by the base 3D graphics engine (observations in a grid).

Example:

```
ufc.plots <- read.csv("../data/ufc-plots.csv")
str(ufc.plots)
library(lattice)
wireframe(vol.m3.ha ~ east * north,
    main = expression(paste("Volume (", m^3, ha^{-1}, ")", sep =         "")),
    xlab = "East (m)", ylab = "North (m)",
    data = ufc.plots
    )
```

To learn more about **base-graphics 3D plots**, run the demonstrations `demo(persp)` and `demo(image)` and look at the examples presented.

22.6 ggplot2

ggplot2 is one of the most elegant and most versatile systems for making graphs. **ggplot2** implements the grammar of graphics, a coherent system for describing and building graphs.

ggplot2 splits the difference between base and lattice in a number of ways. Taking cues from lattice, the ggplot2 system automatically deals with spacings, text, titles but also allows you to annotate by "adding" to a plot.

You can find more information in the Graphs section here.

22.6.1 Installation

```r
install.packages("ggplot2")
library(ggplot2)
```

A typical plot with the ggplot package looks as follows:

```r
data(mpg)
qplot(displ, hwy, data = mpg)
```

In **ggplot2**, aesthetics are the things we can see (e.g. position, color, fill, shape, line type, size). You can use aesthetics in ggplot2 via the `aes()` function:

```r
ggplot(data, aes(x=var1, y=var2))
```

22.6.2 Geometric objects

Geometric objects are the plots or graphs you want to put in the chart.

You can use `geom_point()` to create a scatterplot, `geom_line()` to create a line plot, and `geom_boxplot()` to create a boxplot in the chart.

```r
help.search("geom_", package="ggplot2")
```

In ggplot2, `geom` is also the layers of the chart. You can add in one geom object after another, just like adding one layer after another layer.

```r
ggplot(data, aes(x=var1, y=var2)) +
  geom_point(aes(color="red"))
```

22.6.3 Plotly

Plotly JS allows you to create interactive, publication-quality charts. You can create a Plotly chart using **ggplot**:

```r
install.packages("plotly")
set.seed(12)
var1 <- rnorm(100, mean=1, sd=1)
var2 <- rnorm(100, mean=2, sd=1)
data <- data.frame(var1, var2)
gg <- ggplot(data) + geom_line(aes(x=var1, y=var2))
g <- ggplotly(gg)
```

23 Creating a ggplot

The best place to continue learning about ggplot is the **ggplot2 book**: ggplot2: Elegant graphics for data analysis.

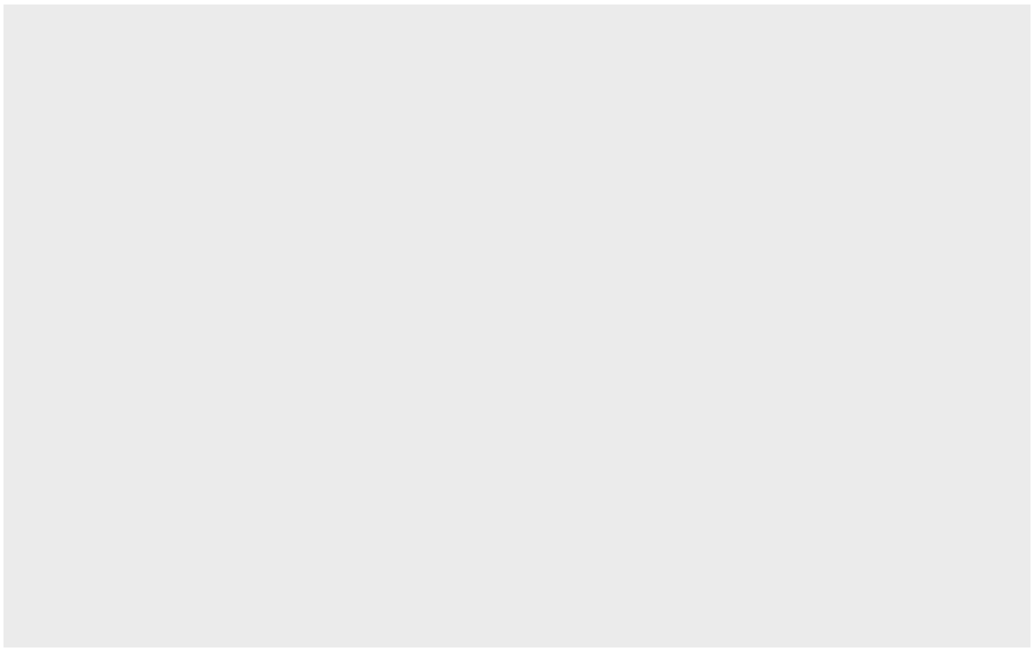
23.1 Define a Plot Object

With **ggplot2**, you begin a plot with the function `ggplot()`, defining a plot object that you then add layers to.

The first argument of `ggplot()` is the **dataset** to use in the graph and so `ggplot(data = penguins)` creates an empty graph that is primed to display the penguins data, but since we haven't told it how to visualize it yet, for now it's empty (it's like an empty canvas you'll paint the remaining layers of your plot onto).

Example:

```r
ggplot(data = penguins)
```



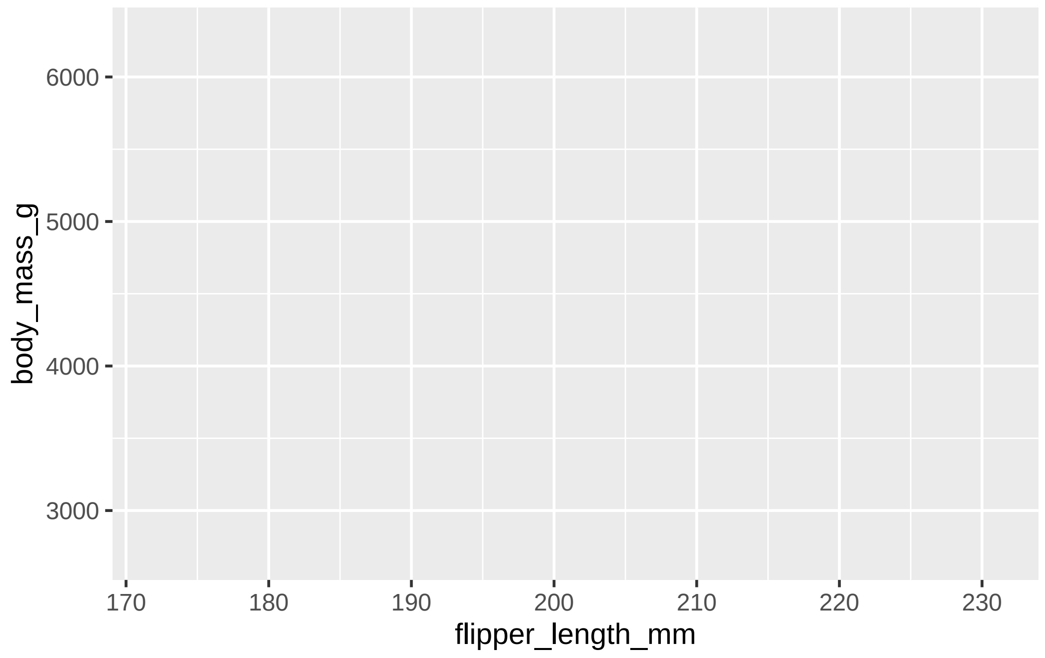
23.2 Map the Data

Next, we need to tell `ggplot()` how the information from our data will be visually represented. The `mapping` argument of the `ggplot()` function defines how variables in your dataset are mapped to visual properties (aesthetics) of your plot.

The `mapping` argument is always defined in the `aes()` function, and the `x` and `y`arguments of `aes()` specify which variables to map to the x and y axes.

Example:

```
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
)
```



Our empty canvas now has more structure – it's clear where flipper lengths will be displayed (on the x-axis) and where body masses will be displayed (on the y-axis). But the penguins themselves are not yet on the plot. This is because we have not yet determine how to represent the observations from our dataframe on our plot.

23.3 Represent the Data

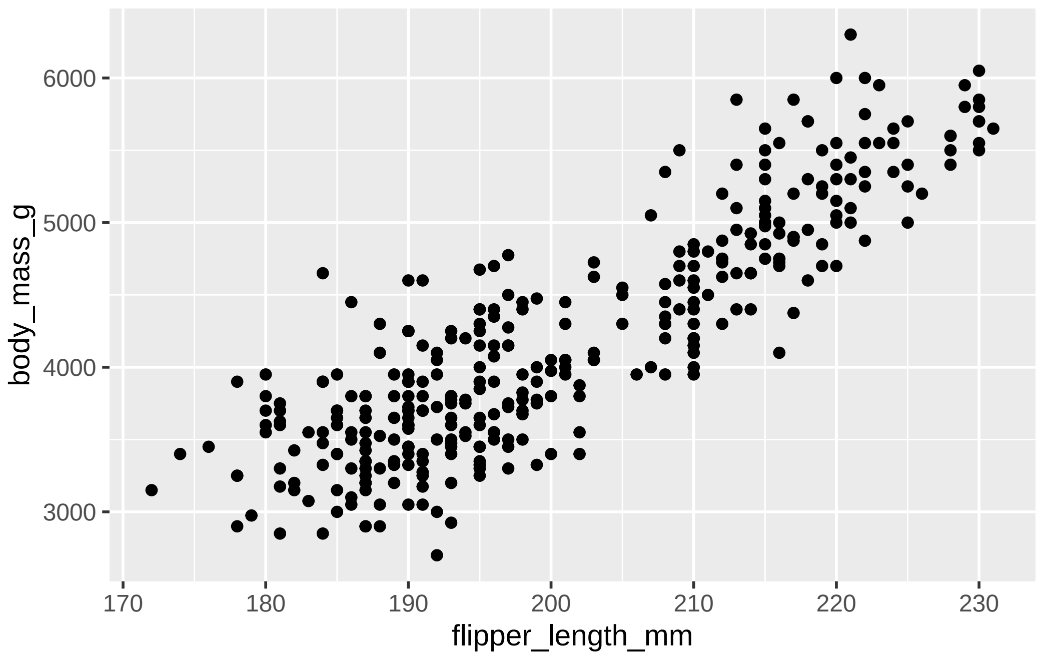
To do so, we need to define a `geom`: the geometrical object that a plot uses to represent data. These geometric objects are made available in ggplot2 with functions that start with `geom_`.

For example, bar charts use `geom_bar()`, line charts use `geom_line()`, boxplots use `geom_boxplot()`, scatterplots use `geom_point()`, and so on.

Example:

```r
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

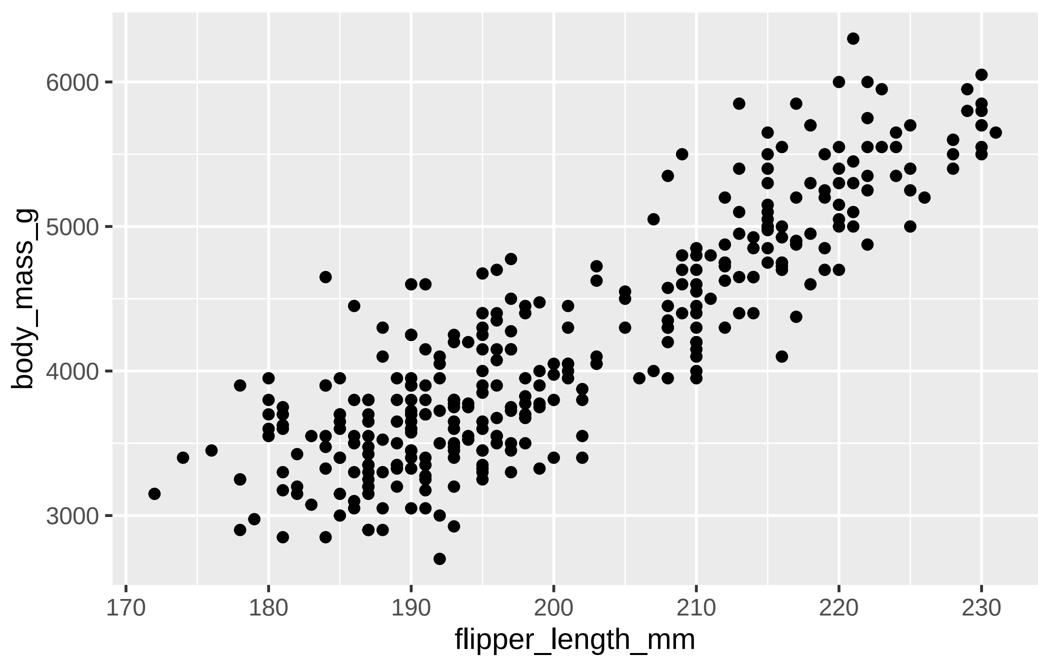


Note: One common problem when creating ggplot2 graphics is to put the **+** in the wrong place: it has to come at the end of the line, not the start.

You can **rewrite the previous plot** more concisely:

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point()
```

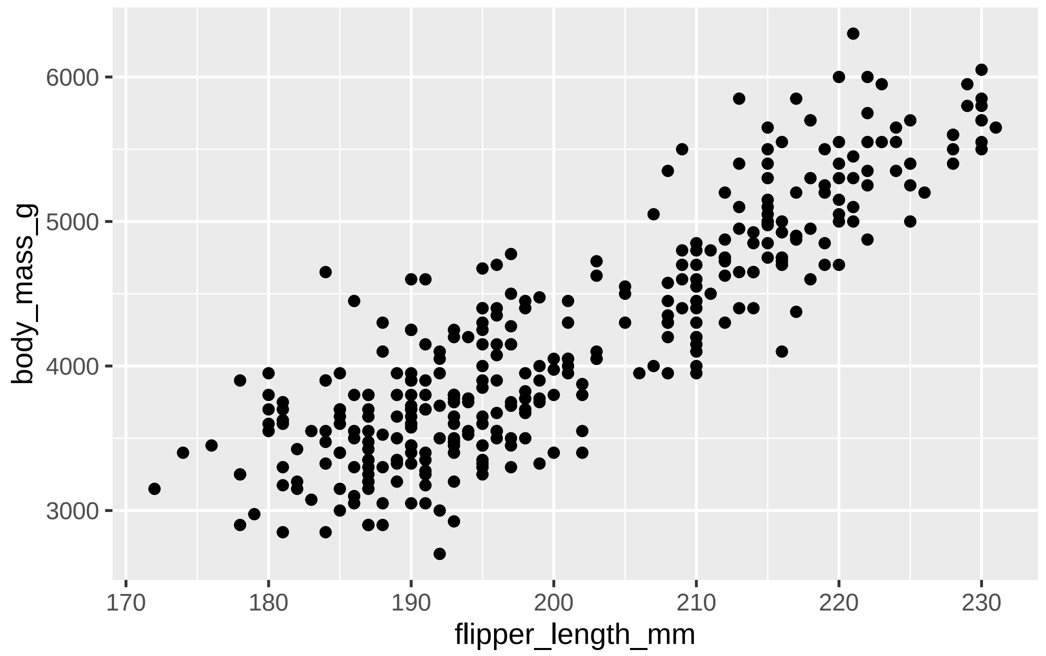
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Or using the pipe, `|>`:

```
penguins |>
  ggplot(aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point()
```

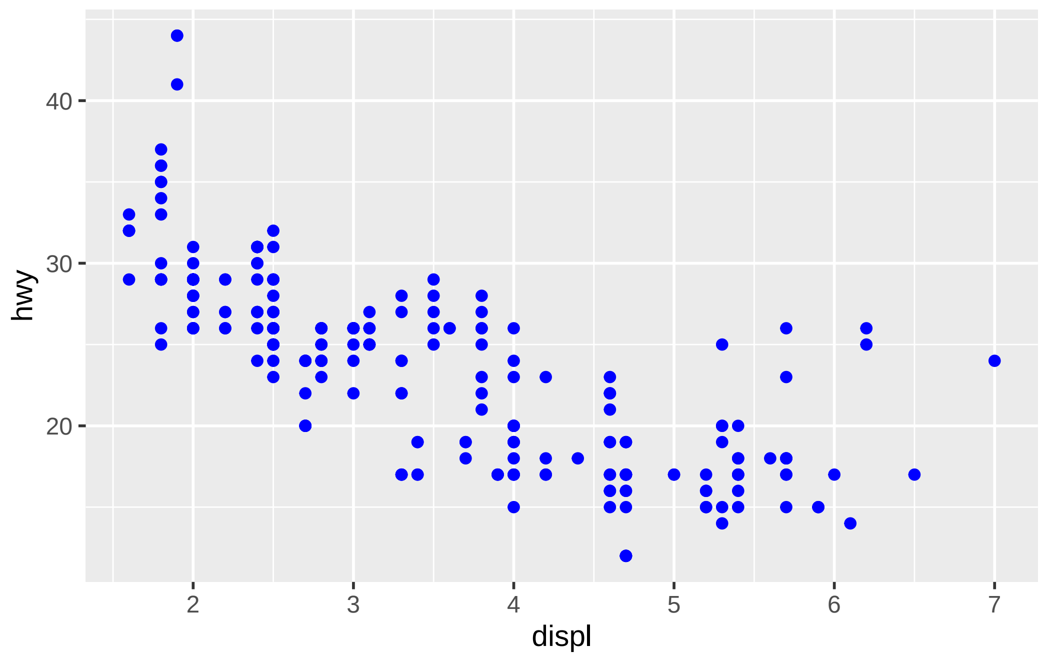
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Now we have something that looks like what we might think of as a "scatterplot".

You can also set the visual **properties of your geom** manually as an argument of your `geom` function (outside of `aes()`) instead of relying on a variable mapping to determine the appearance. For example, we can make all of the points in our plot blue:

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(color = "blue")
```



23.3.1 Help with geoms

The best place to get a comprehensive **overview of all of the geoms** ggplot2 offers, as well as all functions in the package, is this page: https://ggplot2.tidyverse.org/reference. To learn more about any single geom, use the help (e.g., `?geom_smooth`). See also https://exts.ggplot2.tidyverse.org/gallery/ for a sampling of community made geoms.

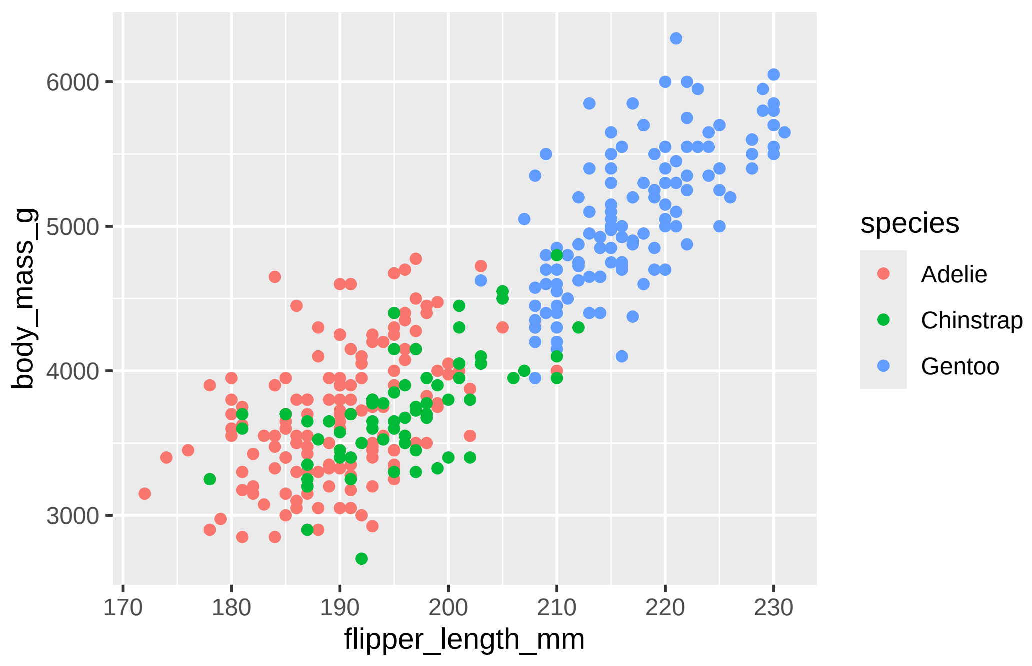
23.4 Add Aesthetics and Layers

You will we need to modify the aesthetic mapping, inside of `aes()`.

Example:

```
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g, color = species)
) +
  geom_point()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Once you **map an aesthetic**, ggplot2 takes care of the rest. It selects a reasonable scale to use with the aesthetic, and it constructs a **legend** that explains the mapping between levels and values. For x and y aesthetics, ggplot2 does not create a legend, but it creates an **axis line** with tick marks and a label. The axis line provides the same information as a legend; it explains the mapping between locations and values.

When a **categorical variable** is mapped to an aesthetic, ggplot2 will automatically assign a unique value of the aesthetic (here a unique color) to each unique level of the variable (each of the three species), a process known as scaling. ggplot2 will also add a **legend** that explains which values correspond to which levels.

Now let's add **one more layer**: a **smooth curve** displaying the relationship between body mass and flipper length. Since this is a new geometric object representing our data, we will add **a new geom** as a layer on top of our point geom: `geom_smooth()`. And we will specify that we want to draw the line of best fit based on a linear model with `method = "lm"`.

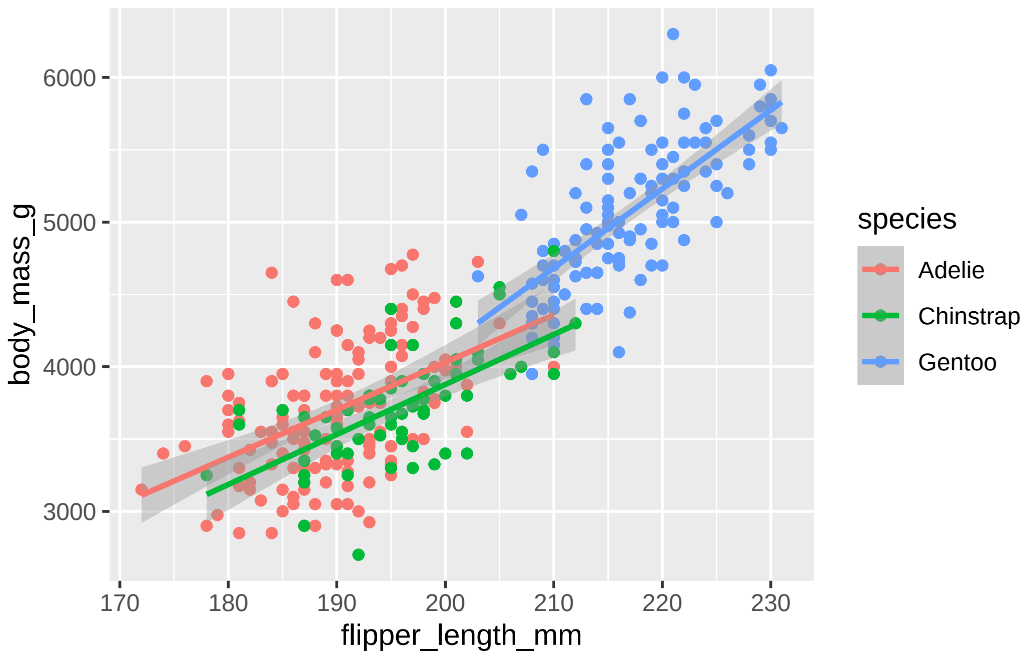
Example:

```r
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g, color = species)
) +
  geom_point() +
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Note that When aesthetic mappings are defined in `ggplot()`, at the **global level**, they're passed down to each of the subsequent geom layers of the plot. However, each geom function in ggplot2 can also take a mapping argument, which allows for aesthetic mappings at the **local level** that are added to those inherited from the global level. Since we want points to be colored based on species but don't want the lines to be separated out for them, we should specify `color = species` for `geom_point()` only.

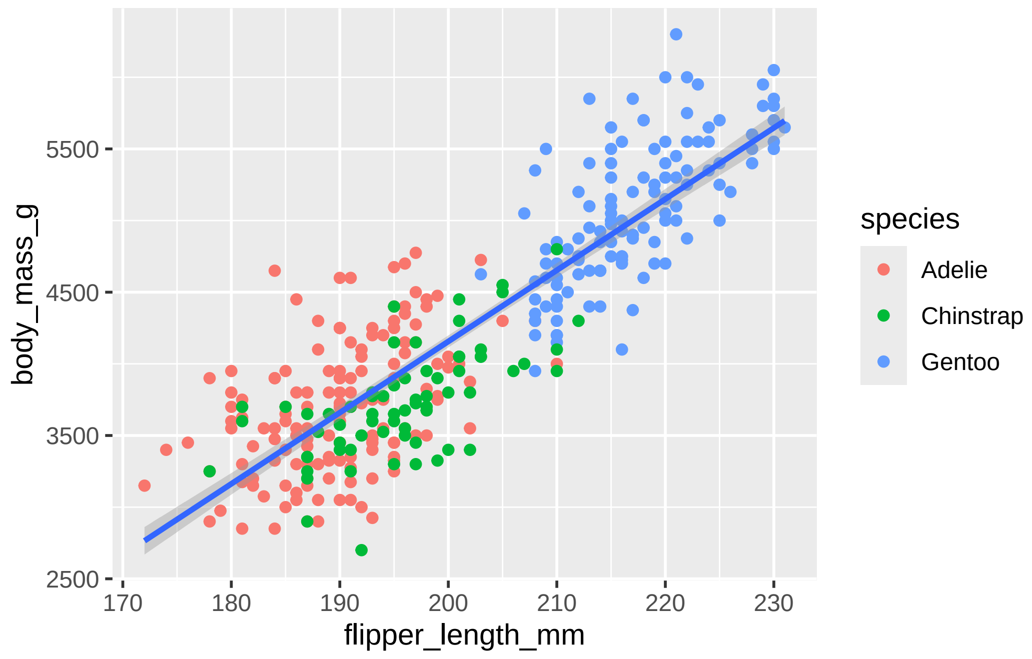
Example:

```r
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point(mapping = aes(color = species)) +
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



It's generally not a good idea to represent information using only colors on a plot, as people perceive colors differently due to color blindness or other color vision differences. Therefore, in addition to color, we can also map species to the **shape aesthetic**.

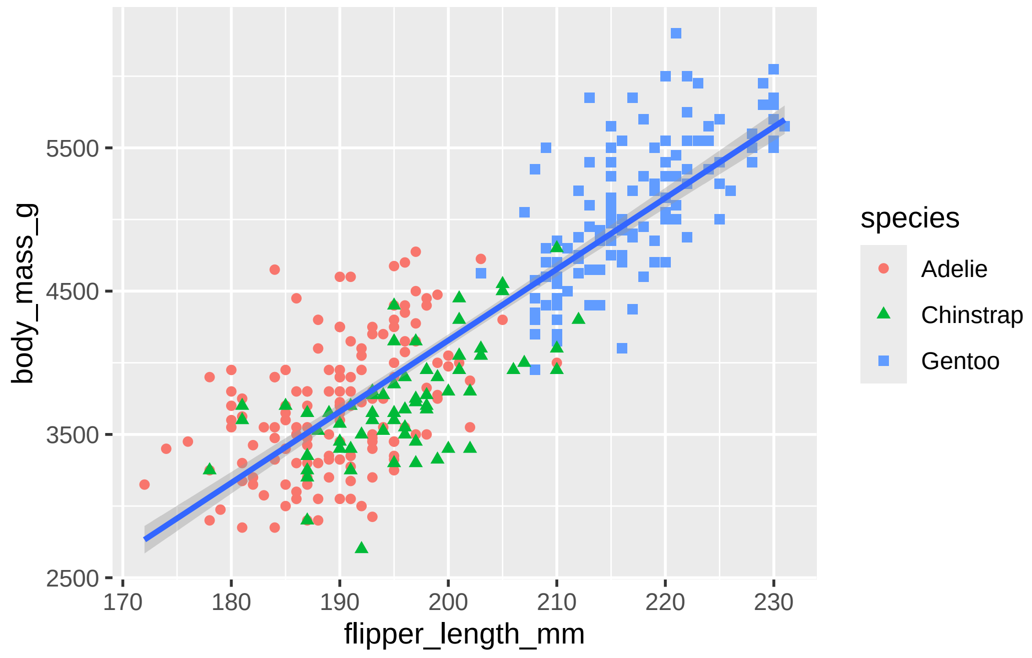
Example:

```r
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point(mapping = aes(color = species, shape = species)) +
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



You can learn more about all possible **aesthetic mappings** in the aesthetic specifications vignette at https://ggplot2.tidyverse.org/articles/ggplot2-specs.html. Remember that the specific aesthetics you can use for a plot depend on the geom you use to represent the data.

23.5 Improve Labels

And finally, we can improve the **labels** of our plot using the `labs()` function in a new layer:

- `title`, adds a title.
- `subtitle`, adds a subtitle to the plot.
- `x`, is the x-axis label.
- `y`, is the y-axis label.
- `color` and `shape` define the label for the legend.

In addition, we can improve the color palette to be **colorblind safe** with the `scale_color_colorblind()` function from the **ggthemes package**.

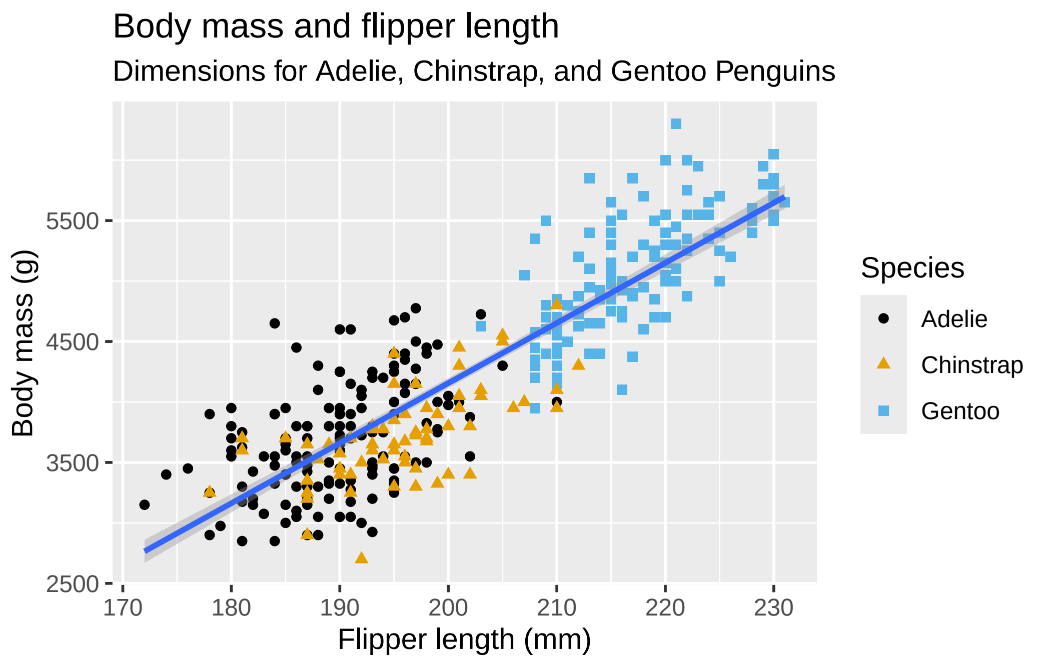
Example:

```r
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point(aes(color = species, shape = species)) +
  geom_smooth(method = "lm") +
  labs(
    title = "Body mass and flipper length",
    subtitle = "Dimensions for Adelie, Chinstrap, and Gentoo Penguins",
    x = "Flipper length (mm)", y = "Body mass (g)",
    color = "Species", shape = "Species"
  ) +
  scale_color_colorblind()
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

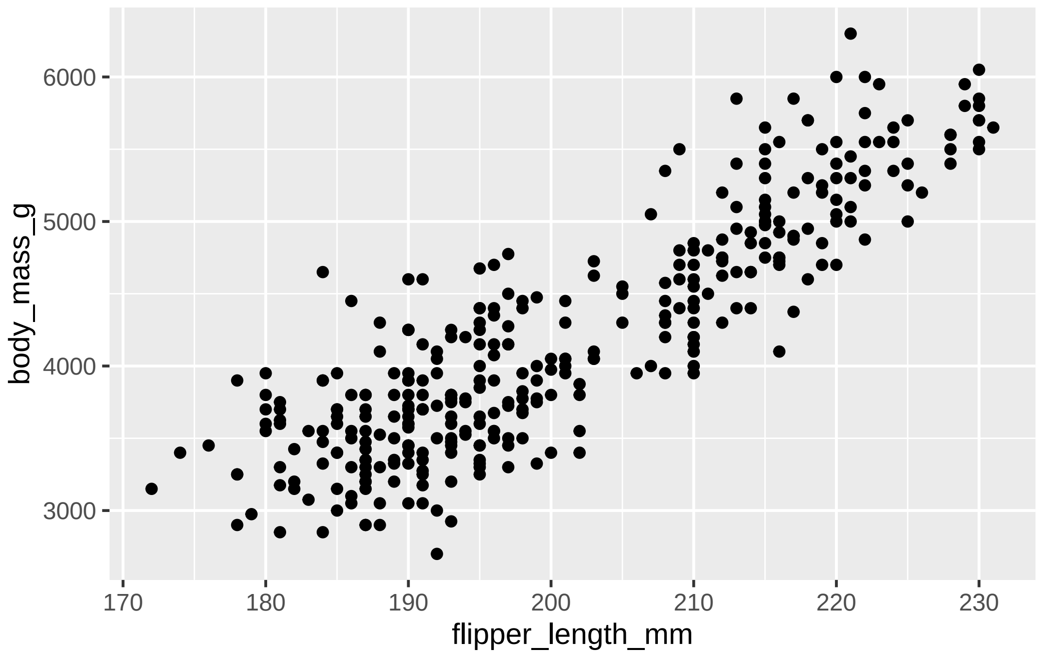


23.6 Save Your Plot

Once you've made a plot, you might want to get it out of R by saving it as an **image** that you can use elsewhere. That's the job of `ggsave()`, which will save the plot most recently created to disk:

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



```
ggsave(filename = "penguin-plot.png")
```

```
Saving 5.5 x 3.5 in image
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

This will save your plot to your **working directory**.

24 Types of Graphs

24.1 Visualizing Categorial Variables

A variable is **categorical** if it can only take one of a small set of values.

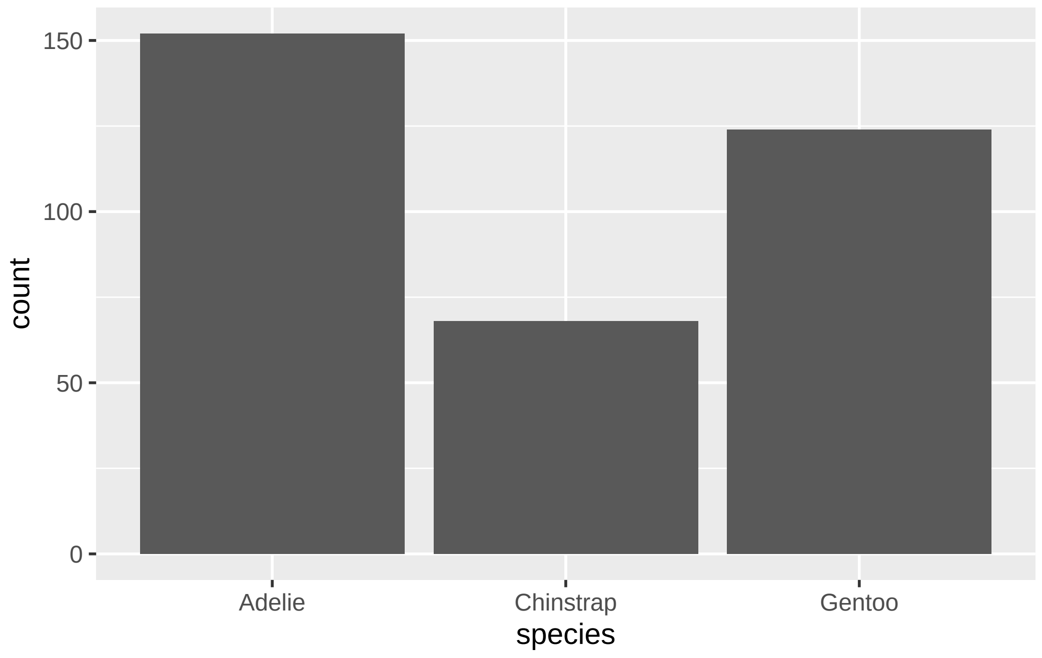
24.1.1 Bar Chart

To examine the distribution of a categorical variable, you can use a bar chart. The height of the bars displays how many observations occurred with each `x` value.

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

Example:

```
ggplot(penguins, aes(x = species)) +
  geom_bar()
```

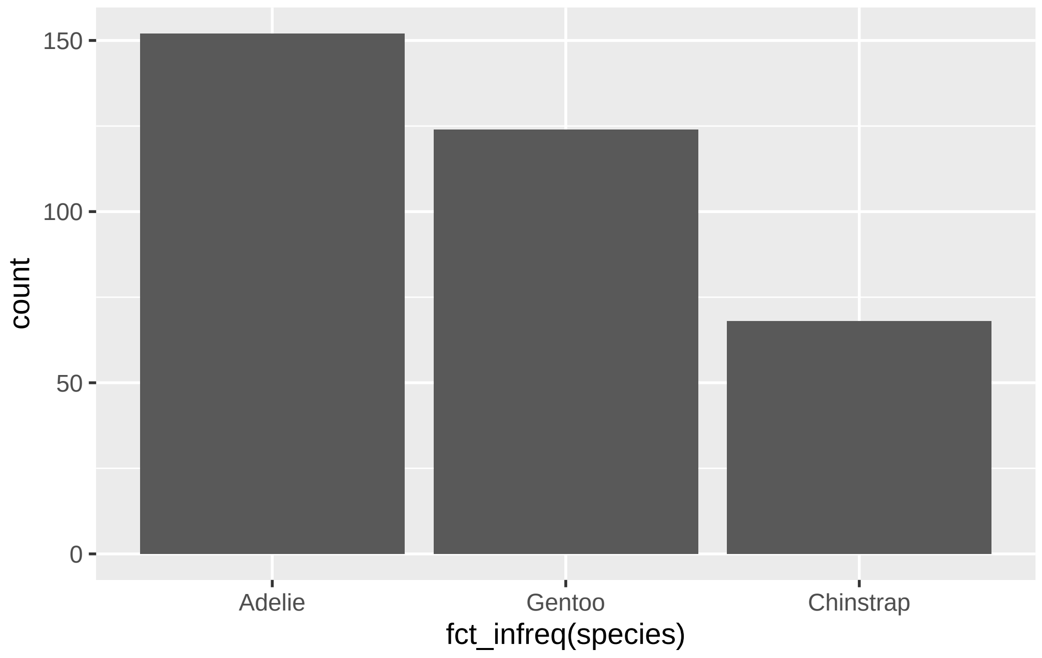


24.1.2 Bar Plots

In bar plots of categorical variables with non-ordered levels, like the penguin species above, it's often preferable to reorder the bars based on their frequencies. Doing so requires transforming the variable to a factor (how R handles categorical data) and then reordering the levels of that factor.

Example:

```r
ggplot(penguins, aes(x = fct_infreq(species))) +
  geom_bar()
```



24.2 Visualizing Numerical Variables

A variable is numerical (or quantitative) if it can take on a wide range of numerical values, and it is sensible to add, subtract, or take averages with those values. Numerical variables can be **continuous or discrete**.

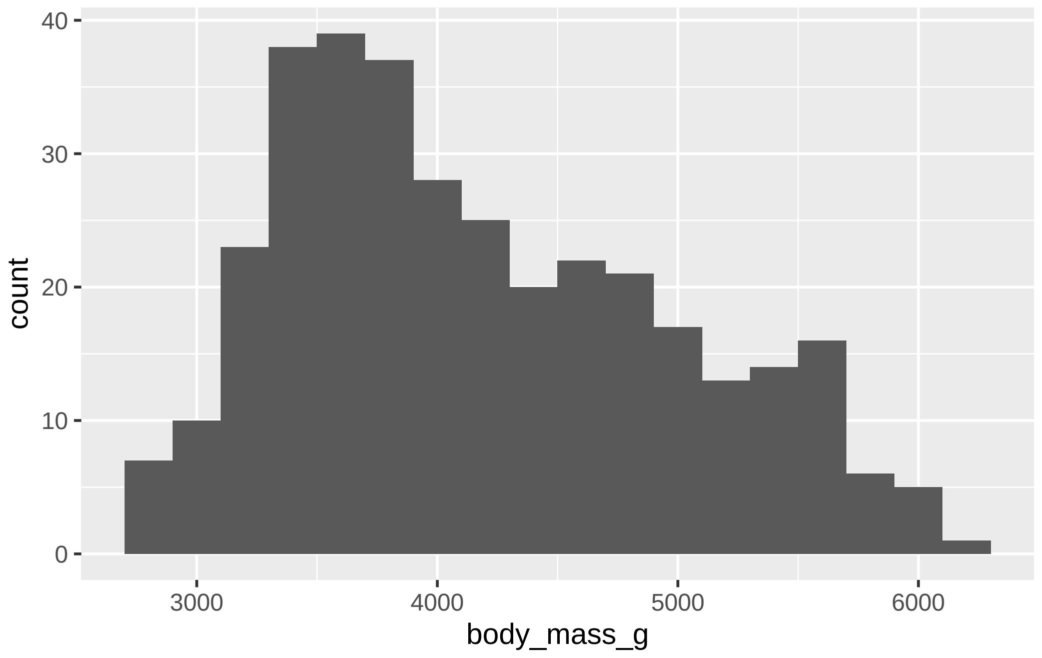
24.2.1 Histogram

One commonly used visualization for distributions of **continuous variables** is a histogram.

Example:

```r
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(binwidth = 200)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```



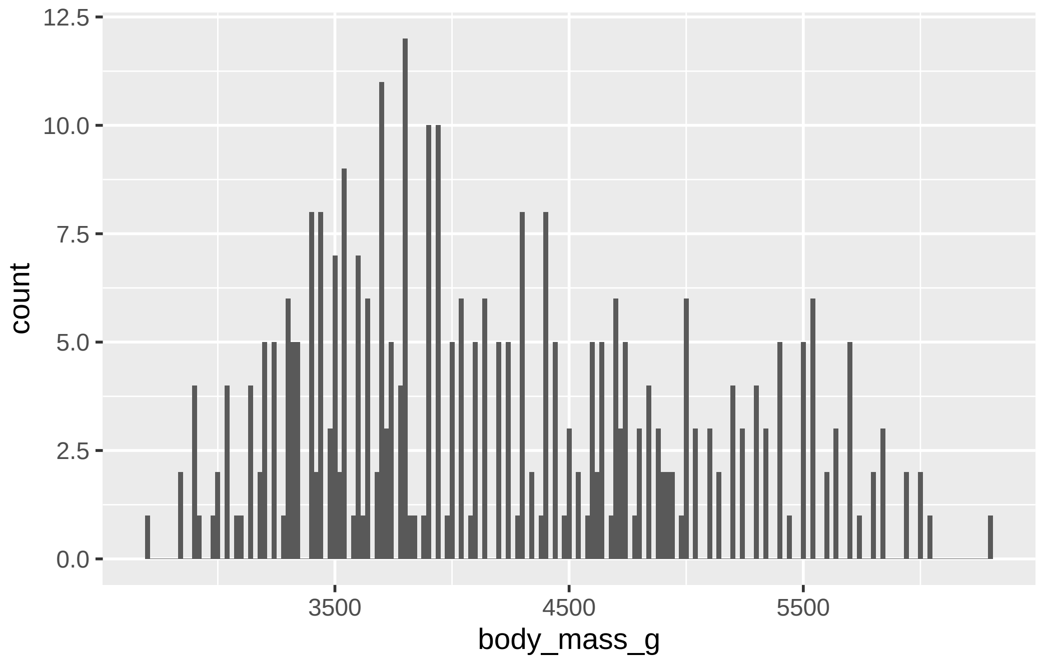
A **histogram** divides the x-axis into equally spaced bins and then uses the height of a bar to display the number of observations that fall in each bin. In the graph above, the tallest bar shows that 39 observations have a body_mass_g value between 3,500 and 3,700 grams, which are the left and right edges of the bar.

You can set the width of the **intervals** in a histogram with the binwidth argument, which is measured in the units of the x variable.

Example:

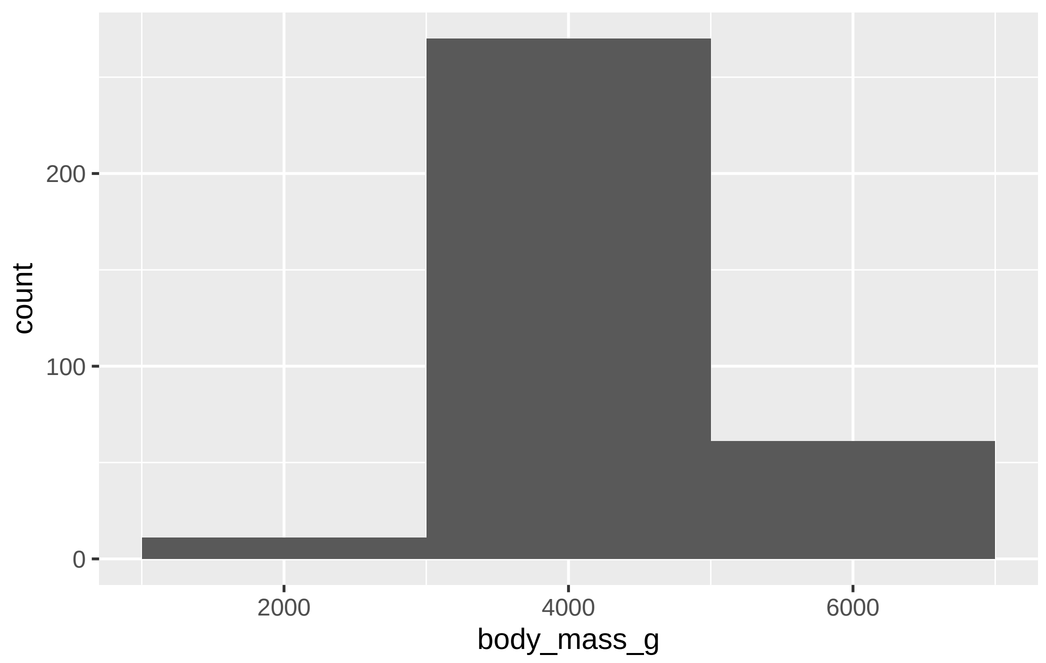
```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(binwidth = 20)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```



```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(binwidth = 2000)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```

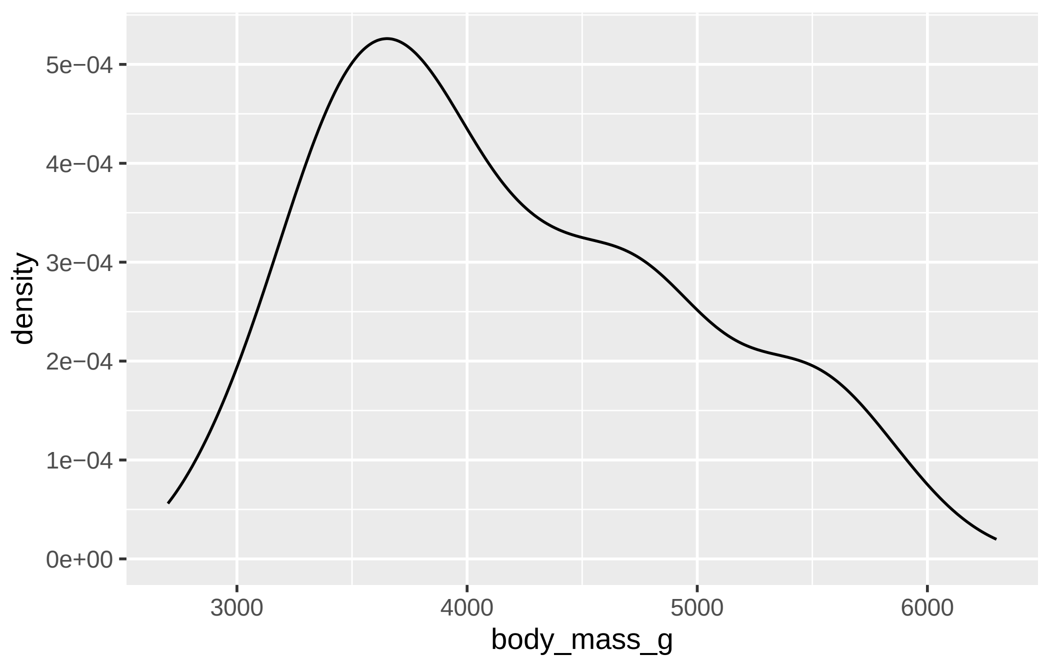


An alternative visualization for distributions of numerical variables is a **density plot**. A density plot is a smoothed-out version of a histogram and a practical alternative, particularly for continuous data that comes from an underlying smooth distribution.

Example:

```r
ggplot(penguins, aes(x = body_mass_g)) +
  geom_density()
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_density()`).
```



24.3 Visualizing Relationships

To visualize a relationship we need to have at least two variables mapped to aesthetics of a plot.

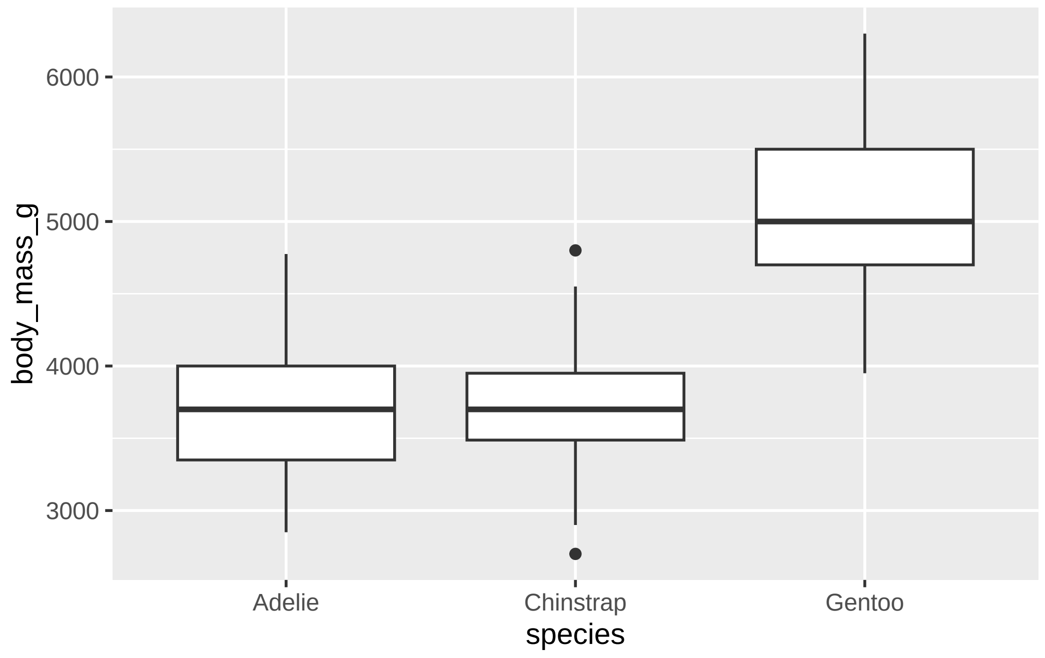
24.3.1 Numerical and Categorical Variable

You can use **side-by-side box plots**. A boxplot is a type of visual shorthand for measures of position (percentiles) that describe a distribution. It is also useful for identifying potential outliers.

Example:

```r
ggplot(penguins, aes(x = species, y = body_mass_g)) +
  geom_boxplot()
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_boxplot()`).
```

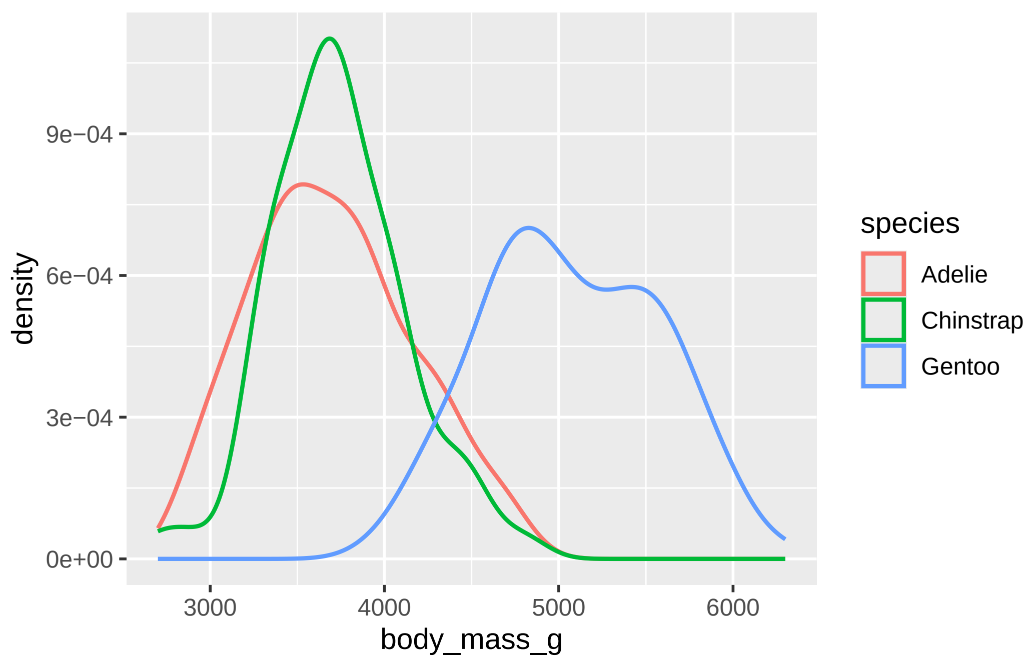


Alternatively, we can make density plots with `geom_density()`.

Example:

```
ggplot(penguins, aes(x = body_mass_g, color = species)) +
  geom_density(linewidth = 0.75)
```

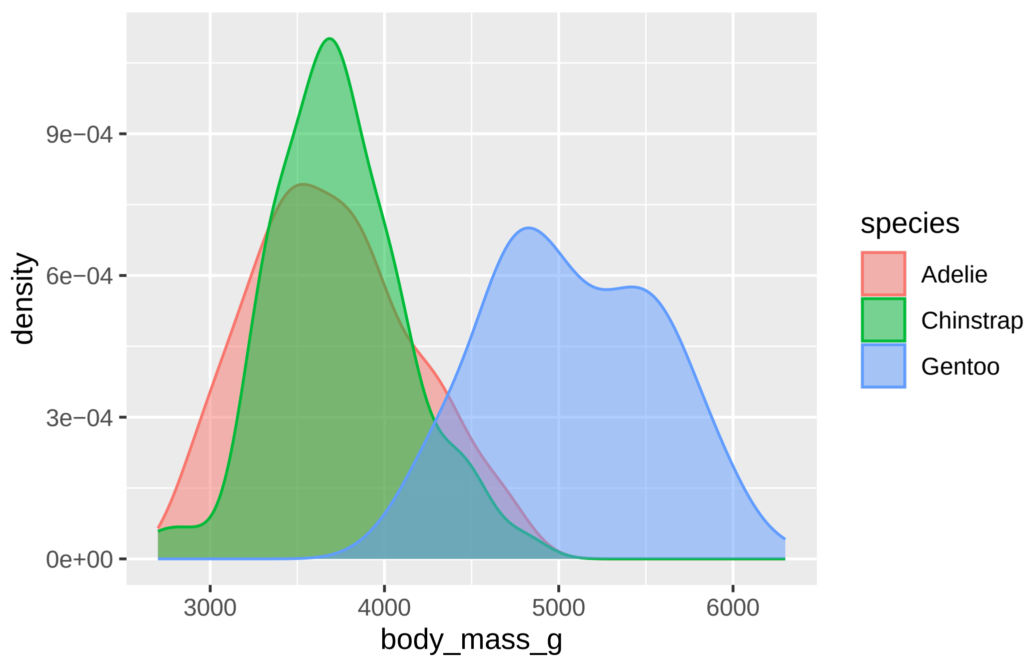
```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_density()`).
```



Additionally, we can map species to both **color and fill** aesthetics and use the alpha aesthetic to add transparency to the filled density curves. This aesthetic takes values between 0 (completely transparent) and 1 (completely opaque). In the following plot it's set to 0.5.

```
ggplot(penguins, aes(x = body_mass_g, color = species, fill = species)) +
  geom_density(alpha = 0.5)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_density()`).
```

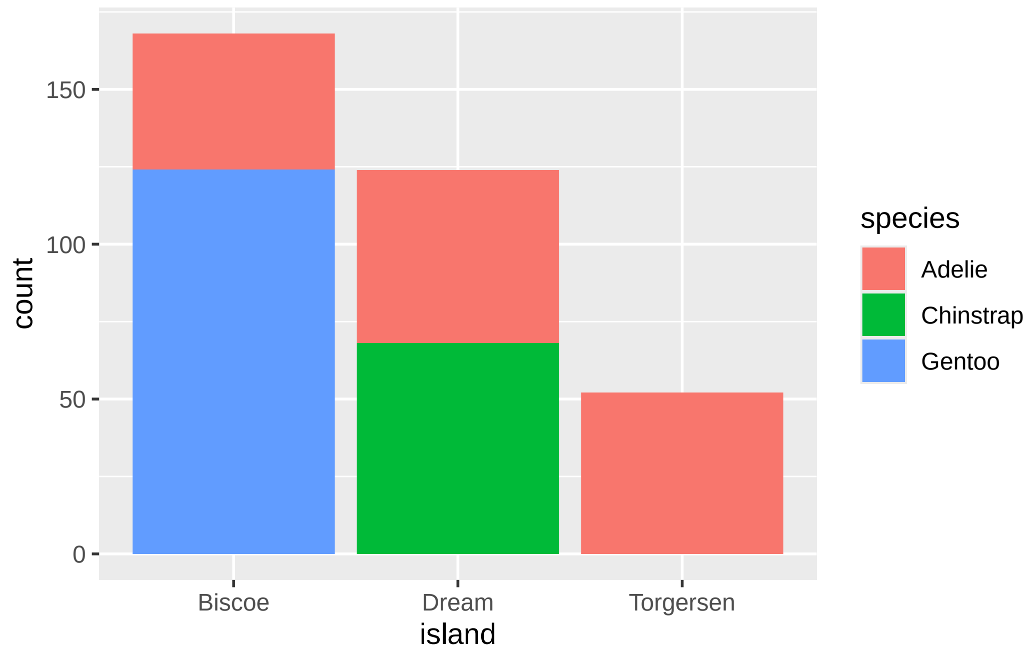


24.3.2 Two Categorical Variables

We can use **stacked bar plots** to visualize the relationship between two categorical variables. For example, the following two stacked bar plots both display the relationship between `island` and `species`, or specifically, visualizing the distribution of `species` within each `island`.

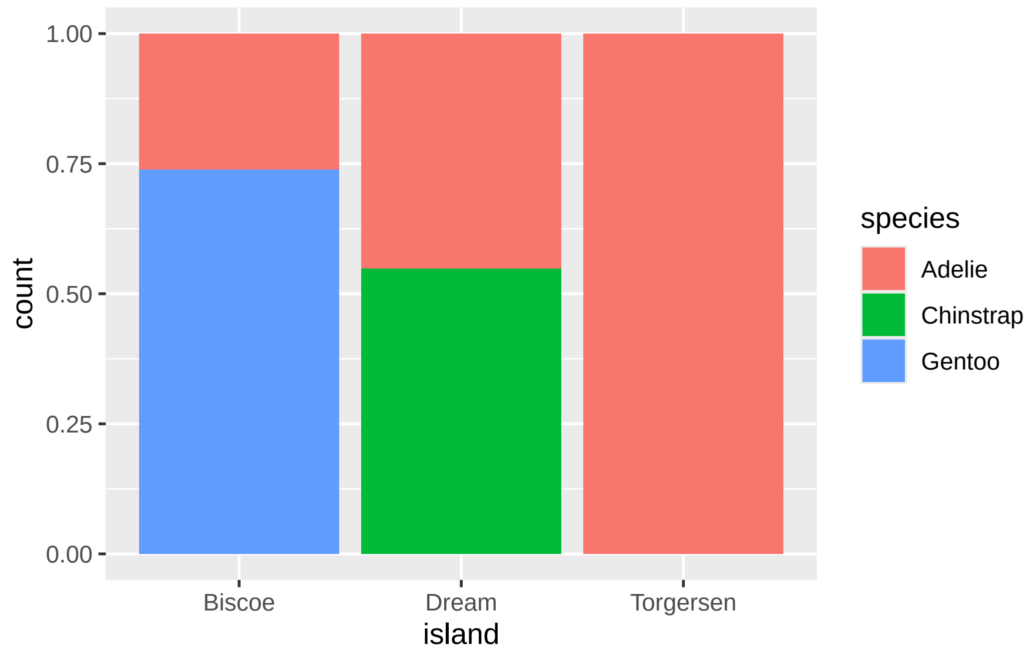
Example:

```
ggplot(penguins, aes(x = island, fill = species)) +
  geom_bar()
```



The following plot, a **relative frequency plot** created by setting `position = "fill"` in the `geom`, is more useful for comparing species distributions across islands since it's not affected by the unequal numbers of penguins across the islands.

```
ggplot(penguins, aes(x = island, fill = species)) +
  geom_bar(position = "fill")
```



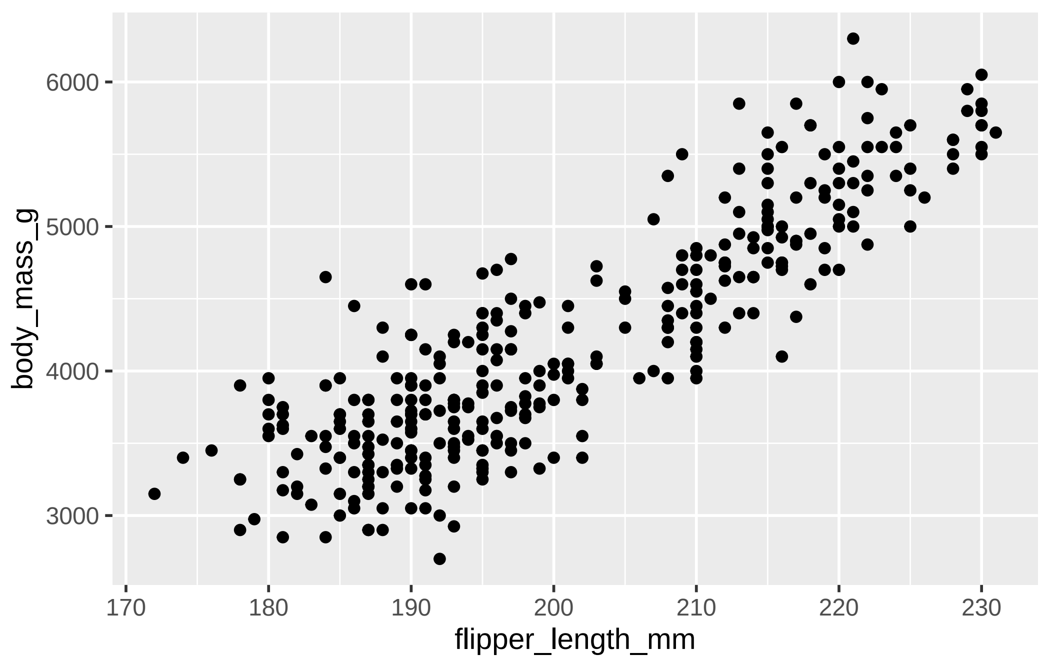
In creating these bar charts, we map the variable that will be separated into bars to the `x` aesthetic, and the variable that will change the colors inside the bars to the `fill` aesthetic.

24.3.3 Two Numerical Variables

A **scatterplot** is probably the most commonly used plot for visualizing the relationship between two numerical variables.

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

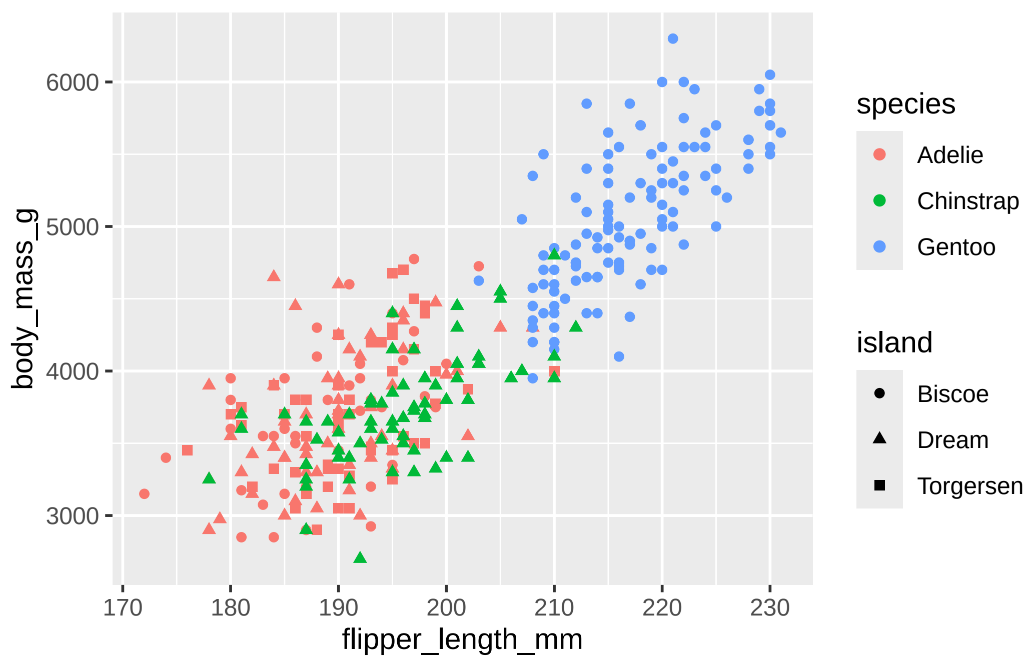


24.3.4 Three or More Variables

We can incorporate more variables into a plot by mapping them to additional aesthetics. For example, in the following **scatterplot** the colors of points represent `species` and the shapes of points represent `islands`.

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point(aes(color = species, shape = island))
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



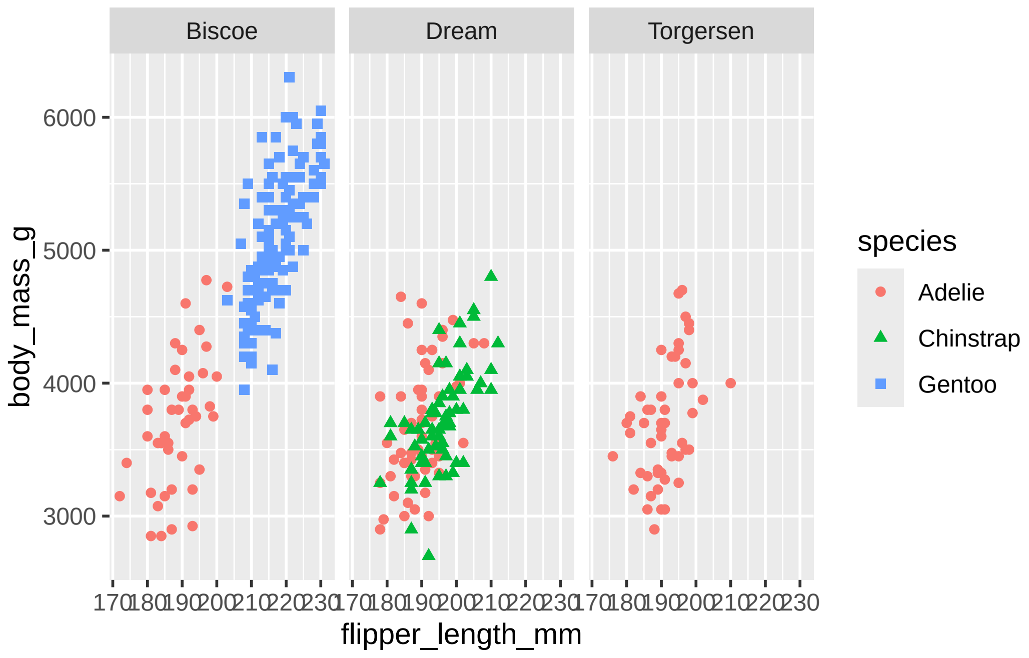
24.3.5 Facets

Adding too many aesthetic mappings to a plot makes it cluttered and difficult to make sense of. Another way, which is particularly useful for categorical variables, is to split your plot into **facets**, subplots that each display one subset of the data.

To facet your plot by a single variable, use `facet_wrap()`. The first argument of `facet_wrap()` is a formula3, which you create with `~` followed by a variable name. The variable that you pass to `facet_wrap()` should be categorical.

```r
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point(aes(color = species, shape = species)) +
  facet_wrap(~island)
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



If you don't specify the `width` and `height` they will be taken from the dimensions of the current plotting device. For reproducible code, you'll want to specify them.

Part VI

Part 5: Modeling

25 Random Numbers

Simulation is an important (and big) topic for both statistics and for a variety of other areas where there is a need to introduce randomness.

Sometimes you want to implement a statistical procedure that requires random number generation or sample (i.e. Markov chain Monte Carlo, the bootstrap, random forests, bagging), or you want to simulate a system, then **random number generators** can be used to model random inputs.

Some **example functions** of random numbers probability distributions in R are these:

```r
rnorm(n, mean = 0, sd = 1) ## generate random Normal variates with a given mean and standa
dnorm(x, mean = 0, sd = 1, log = FALSE) ## evaluate the Normal probability density (with a
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)  ## evaluate the cumulative d
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE) ## generate random Poisson va
```

Maybe the most common probability distribution to work with is the **normal distribution** (also known as the Gaussian).

Example:

```r
x <- rnorm(10, 20, 2)
summary(x)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  14.42   18.27   19.22   19.49   20.99   24.35 
```

25.1 Always Remember to Set Your Seed!

When simulating any random numbers it is essential to set the **random number seed**. Setting the random number seed with `set.seed()` ensures reproducibility of the sequence of random numbers.

Example:

```r
set.seed(1)
rnorm(5)
```

```
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

Example: Simulating a Linear Model

Suppose we want to simulate from the following linear model $y = \beta_0 + \beta_1 x + \varepsilon$ where $\varepsilon \sim N(0, 22)$

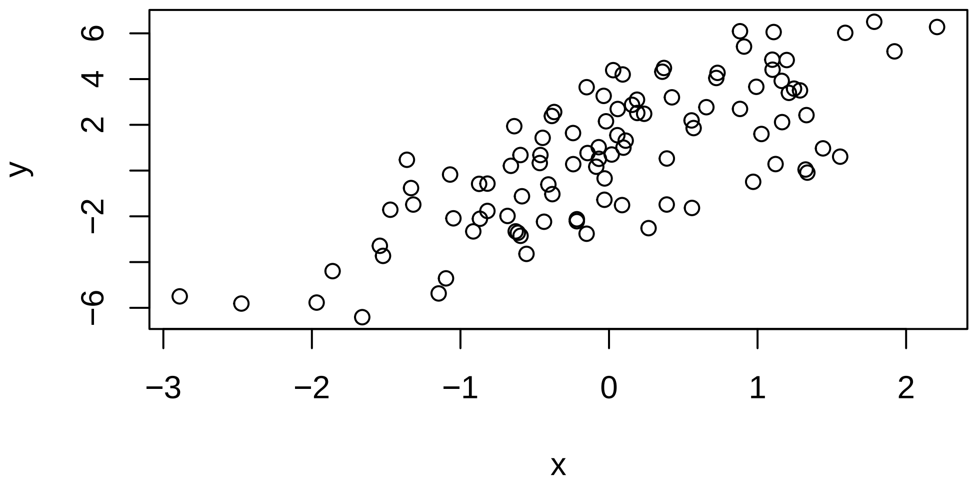
Assume $x \sim N(0, 12)$, $\beta_0 = 0.5$ and $\beta_1 = 2$. The variable `x` might represent an important predictor of the outcome `y`.

Here's how you could do that in R:

```r
## Always set your seed!
set.seed(20)
## Simulate predictor variable
x <- rnorm(100)
## Simulate the error term
e <- rnorm(100, 0, 2)
## Compute the outcome via the model
y <- 0.5 + 2 * x + e
summary(y)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
-6.4084 -1.5402  0.6789  0.6893  2.9303  6.5052 
```

```r
plot(x, y)
```



25.2 Random Sampling

The `sample()` function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions of numbers.

Example:

```r
set.seed(1)
sample(1:10,4)
```

```
[1] 9 4 7 1
```

```r
# Doesn't have to be numbers
sample(letters, 5)
```

```
[1] "b" "w" "k" "n" "r"
```

To sample more complicated things, such as rows from a **dataframe or a list**, you can sample the indices into an object rather than the elements of the object itself:

```r
set.seed(20)
idx <- seq_len(nrow(airquality))## Create index vector
samp <- sample(idx, 6) ##Sample from the index vector
airquality[samp, ]
```

```
    Ozone Solar.R Wind Temp Month Day
107    NA      64 11.5   79     8  15
120    76     203  9.7   97     8  28
130    20     252 10.9   80     9   7
98     66      NA  4.6   87     8   6
29     45     252 14.9   81     5  29
45     NA     332 13.8   80     6  14
```

Part VII

Part 6: Data Products

26 Packages for Data Products

26.1 Manipulate

The **manipulate package** creates a quick interactive graphic that offers simple controls, including sliders, pickers and checkboxes.

```r
library(manipulate)
manipulate(plot(1:x), x = slider(1, 100))
```

With **manipulate** you can have more than one set of controls by simply adding more arguments to the manipulate function.

Note that it's difficult to share a manipulate interactive graphic.

This is a link on how **manipulate** actually works: link

26.2 Shiny

It is described by RStudio as "A web application framework for R". "Turn your analyses into interactive web applications No HTML, CSS, or JavaScript knowledge required".

Only those who have **shiny** installed and have access to your code could run your web page. However, RStudio offers a service for hosting shiny apps (their servers) on a platform called **shinyapps.io**.

If on Windows, make sure that you have Rtools installed. Then, you can install shiny with:

```r
install.packages("shiny")
library(shiny)
```

A **shiny app** consists of two files. First, a file called **ui.R** that controls the User Interface (hence the ui in the filename) and secondly, a file **server.R** that controls the shiny server (hence the server in the filename).

26.2.1 Create ui.R file:

```
library(shiny)
shinyUI(
  pageWithSidebar(
    headerPanel("Hello Shiny!"),
    sidebarPanel(
      h3('Sidebar text')
    ),
    mainPanel(
      h3('Main Panel text')
    )
  )
)
```

26.2.2 Create server.R file:

```
library(shiny)
shinyServer(
  function(input, output) {
  }
)
```

The current version of Rstudio has a **"Run app" button** in the upper right hand corner of the editor if a ui.R or server.R file is open.

To get used to programming shiny apps, you need to throw away a little of your thinking about R; it's a different style of programming.

26.2.3 Types of inputs

- Numeric input:

  ```
  numericInput('id1', 'Numeric input, labeled id1', 0, min = 0, max = 10, step= 1)
  ```

- Checkbox input:

  ```
  checkboxGroupInput("id2", "Checkbox", c("Value 1" = "1", "Value 2" = "2", "Value 3" =
  ```

- Date input:

```
dateInput("date", "Date:")
```

26.2.4 Sharing your app

Now that we have a working app we'd like to share it with the world. It's much nicer to have it display as a standalone web application.

This requires running a shiny server to host the app. Instead of creating and deploying our own shiny server, we'll rely on RStudio's service: link

- After login in shinyapps:

  ```
  install.packages("devtools")
  install.packages("shinyapps")
  ```

- Run code:

  ```
  shinyapps::setAccountInfo(name='<ACCOUNT NAME>',
        token='<TOKEN>',
        secret='<SECRET>')
  ```

- Submit code:

  ```
  deployApp(appName = "myFirstApp")
  ```

26.2.5 Build a GUI with html

Check out this link

26.3 Reproducible presentations

26.3.1 Slidify

Slidify is for building reproducible presentations.

```
install.packages("devtools")
library(devtools)/ require(devtools)
install_github('ramnathv/slidify')
install_github('ramnathv'/'slidifyLibraries') o devtools::install_github(pkgs, force = TRU
```

```r
library(slidify)
```

26.3.2 R studio presenter

For more information about R studio presenter, chek out this link

Tip: if you're sort of a hacker type and you like to tinker with things, use slidify. If you just want to get it done and not worry about it, use RPres. Either way, you really can't go wrong.

26.4 Interactive Graphs

26.4.1 rCharts

link to rCharts

rCharts is a way to create interactive javascript visualizations using R.

```r
require(devtools)
install_github('rCharts', 'ramnathv')
```

26.4.2 googleVis

link to googlevis

Google has some nice visualization tools built into their products (e.g. Google Maps). These include maps and interactive graphs.

```r
install.packages("googleVis")
library(googlevis)
```

26.4.3 leaflet

link to leaflet

leaflet seems to be emerging as the most popular R package for creating **interactive maps**.

The map widget (the `leaflet()` command) starts out a map and then you add elements or modify the map by passing it as arguments to mapping functions.

26.4.4 plot.ly

link to plot.ly

plotly relies on the platform/website plot.ly for creating interactive graphics.

```r
require(devtools)
install_github("ropensci/plotly")
```

Plotly will give you the json data and gives a tremendous number of options for publishing the graph.

Notably, **plotly** allows for integration with **ggplot2**.

For interactive graphics, learning some javascript and following that up with D3 would be the logical next step

27 Reproducible Reporting

Reproducibility of computational research is very important, primarily to increase transparency and to improve knowledge sharing.

27.1 Goal

The goal is to have independent people to do independent things with different data, different methods, and different laboratories and see if you get the same result. But the problem is that it's becoming more and more challenging to do replication or to replicate other studies. Part of the reason is because studies are getting bigger and bigger.

The idea behind a **reproducible reporting** is to create a kind of minimum standard or a middle ground where we won't be replicating a study, but maybe we can do something in between.

You need to make the data available for the original study and the computational methods available so that other people can look at your data and run the kind of analysis that you've run, and come to the same findings that you found. If you can take someone's data and reproduce their findings, then you can, in some sense, validate the data analysis.

Understanding what someone did in a data analysis now requires looking at code and scrutinizing the computer programs that people used.

27.2 The Data Science Pipeline

The basic idea behind **reproducibility** is to focus on the elements in the blue blox: the **analytic data** and the **computational results**.

With **reproducibility** the goal is to allow the author of a report and the reader of that report to "meet in the middle".

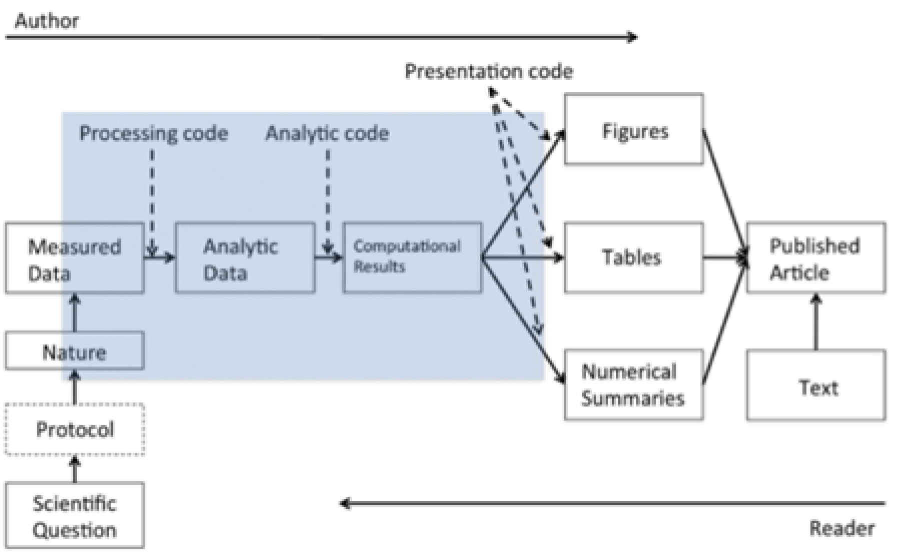


Figure 27.1: Figure 8: The Data Science Pipeline

27.3 Organizing a Data Analysis

- **Raw Data:**
  - You want to store this raw data in your analysis folder.
  - If the data were accessed from the web you want to include things like the URL, where you got the data, what the data set is, a brief description of what it's for, the date that you accessed the URL on, the website, etc.
  - You may want this in a README file.
  - If you're using git to track things that are going on in your project, add your raw data, if possible. In the log message, when you add it you can talk about what the website was where you got it, what the URL was, etc.
- **Processed Data:**
  - Your processed data should be named so that you can easily see what script generated what data.
  - In any README file or any sort of documentation, it's important to document what code files were used to transform the raw data into the processed data.
- **Figures:**
  - Exploratory figures.
  - Final figures. The final figures usually make a very small subset of the set of exploratory figures that you might generate. You typically don't want to inundate

people with a lot of figures because then the ultimate message of what you're trying to communicate tends to get lost in a pile of figures.

- **Scripts:**
  - Final scripts will be much more clearly commented. You'll likely have bigger comment blocks for whole sections of code.
- **R Markdown files:**
  - They may not be exactly required, but they can be very useful to summarize parts of an analysis or an entire analysis.
  - You can embed code and text into the same document and then you process the document into something readable like a webpage or a PDF file.
- **Final Report:**
  - The point of this is to tell the final story of what you generated here.
  - You'll have a title, an introduction that motivates your problem, the methods that you used to refine, the results and any measures of uncertainty, and then any conclusions that you might draw from the data analysis that you did, including any pitfalls or potential problems.

27.4 Structure of a Data Analysis

1. Defining the question:

- A proper data analysis has a scientific context, and at least some general question that we're trying to investigate which will narrow down the kind of dimensionality of the problem. Then we'll apply the appropriate statistical methods to the appropriate data.
- Defining a question is the most powerful dimension reduction tool you can ever employ.
- The idea is, if you can narrow down your question as specifically as possible, you'll reduce the kind of noise that you'll have to deal with when you're going through a potentially very large data set.
- Think about what type of question you're interested in answering before you go delving into all the details of your data set. That will lead you to the data. Which may lead you to applied statistics, which you use to analyze the data.

2. Defining the ideal dataset/Determining what data you can access (the real data set):

- sometimes you have to go for something that is not quite the ideal data set.
- You might be able to find free data on the web. You might need to buy some data from a provider.

- If the data simply does not exist out there, you may need to generate the data yourself in some way.

3. **Obtaining the data:**

   - You have to be careful to reference the source, so wherever you get the data from, you should always reference and keep track of where it came from.
   - If you get data from an Internet source, you should always make sure at the very minimum to record the URL, which is the web site indicator of where you got the data, and the time and date that you accessed it.

4. **Cleaning the data:**

   - Raw data typically needs to be processed in some way to get it into a form where you can model it or feed it into a modeling program.
   - If the data is already pre-processed, it's important that you understand how it was done. Try to get some documentation about what the pre-processing was and how the sampling was done.
   - It is very important that anything you do to clean the data is recorded.
   - Once you have cleaned the data and you have gotten a basic look at it, it is important to determine if the data are good enough to solve your problems.
   - If you determine the data are not good enough for your question, then you've got to quit, try again, change the data, or try a different question. It is important to not simply push on with the data you have, just because that's all that you've got, because that can lead to inappropriate inferences or conclusions.

5. **Exploratory data analysis:**

   - It would be useful to look at what are the data, what did the data look like, what's the distribution of the data, what are the relationships between the variables.
   - You want to look at basic summaries, one dimensional, two dimensional summaries of the data and we want to check for is there any missing data, why is there missing data, if there is, create some exploratory plots and do a little exploratory analyses. o Split the data set into Train and Test data sets:

     ```
     library(kernlab)
     data(spam)
     set.seed(3435)
     trainIndicator = rbinom(4601, size = 1, prob = 0.5) table(trainIndicator)
     trainSpam = spam[trainIndicator == 1, ]
     testSpam = spam[trainIndicator == 0, ]
     ```

- We can make some plots and we can compare, what are the frequencies of certain characteristics between the spam and the non spam emails:

  ```
  boxplot(capitalAve ~ type, data = trainSpam)
  pairs(log10(trainSpam[, 1:4] + 1))   ## pairs plot of the first four variables
  ```

- You can see that some of them are correlated, some of them are not particularly correlated, and that's useful to know.
- Explore the predictors space a little bit more by doing a hierarchical cluster analysis, e.g. the Dendrogram just to see how what predictors or what words or characteristics tend to cluster together

  ```
  hCluster = hclust(dist(t(trainSpam[, 1:57])))
  plot(hCluster)
  ```

6. **Statistical prediction/modeling:**

- Any statistical modeling that you engage in should be informed by questions that you're interested in, of course, and the results of any exploratory analysis. The exact methods that you employ will depend on the question of interest.
- we're just going to cycle through all the variables in this data set using this for-loop to build a logistic regression model, and then subsequently calculate the cross validated error rate of predicting spam emails from a single variable.
- Once we've done this, we're going to try to figure out which of the individual variables has the minimum cross validated error rate. It turns out that the predictor that has the minimum cross validated error rate is this variable called charDollar. This is an indicator of the number of dollar signs in the email.
- We can actually make predictions now from the model on the test data (now we're going to predict the outcome on the test data set to see how well we do).
- we can take a look at the predicted values from our model, and then compare them with the actual values from the test data set, because we know which was spam, and which was not. Now we can just calculate the error rate.

7. **Interpretation of results:**

- Think carefully about what kind of language you use to interpret your results. It's also good to give an explanation for why certain models predict better than others, if possible.
- If there are coefficients in the model that you need to interpret, you can do that here.
- And in particular it's useful to bring in measures of uncertainty, to calibrate your interpretation of the final results.

8. Challenging of results:

- It's good to challenge everything, the whole process by which you've gone through this problem. Is the question even a valid question to ask? Where did the data come from? How did you get the data? How did you process the data? How did you do the analysis and draw any conclusions?
- And if you built models, why is your model the best model? Why is it an appropriate model for this problem? How do you choose the things to include in your model?
- All these things are questions that you should ask yourself and should have a reasonable answer to, so that when someone else asks you, you can respond in kind.

9. Synthesis and write up:

- Typically in any data analysis, there are going to be many, many, many things that you did. And when you present them to another person or to a group you're going to want to have winnowed it down to the most important aspects to tell a coherent story.
- Typically you want to lead with the question that you were trying to address.
- It's important that you don't include every analysis that you ever did, but only if its needed for telling a coherent story. Talk about the analyses of your data set in the order that's appropriate for the story you're trying to tell.
- Include very well done figures so that people can understand what you're trying to say in one picture or two.

10. Creating reproducible code:

- You can use tools like RMarkdown and knitr and RStudio to document your analyses as you do them.
- You can preserve the R code as well as any kind of a written summary of your analysis in a single document using knitr.
- If someone cannot reproduce your data analysis then the conclusions that you draw will be not as worthy as an analysis where the results are reproducible.

27.5 R Markdown

link to R Markdown guide

The benefit of **Markdown** for writers is that it allows one to focus on writing as opposed to formatting. It has simple and minimal yet intuitive formatting elements and can be easily converted to valid HTML (and other formats) using existing tools.

R markdown is the integration of R code with Markdown. Documents written in R Markdown have R coded nested inside of them, which allows one to create documents containing "live" R code.

R markdown can be converted to standard markdown using the **knitr package** in R. Markdown can subsequently be converted to HTML using the markdown package in R.

27.6 Knitr

For literate statistical programming, the idea is that a report is viewed as a stream of text and code.

Analysis code is divided into code chunks with text surrounding the code chunks explaining what is going on: - In general, literate programs are weaved to produce human-readable documents - and tangled to produce machine- readable documents

The requirements for writing literate programs are a documentation language (**Markdown**) and a programming language (**R**).

My First knitr Document:

- Open an R Markdown document.
- RStudio will prompt you with a dialog box to set some of the metadata for the document.
- When you are ready to process and view your R Markdown document the easiest thing to do is click on the Knit HTML button that appears at the top of the editor window.
- Note here that the the code is echoed in the document in a grey background box and the output is shown just below it in a white background box. Notice also that the output is prepended with two pound symbols.
- Code chunks begin with `{r} and end with just`. Any R code that you include in a document must be contained within these delimiters, unless you have inline code.
- Hiding code:

  ```
  {r pressure, echo=FALSE}
  ```

- Hiding results:

  ```
  {r pressure, echo=FALSE, results = "hide"}
  ```

- Rather than try to copy and paste the result into the paragraph, it's better to just do the computation right there in the text:

  My favourite random number is `r rnorm(1)`

- Tables can be made in R Markdown documents with the help of the xtable package.
- The opts_chunk variable sets an option that applies to all chunks in your document. For example, if we wanted the default to be that all chunks do NOT echo their code and always hide their results, we could set:

  knitr::opts_chunk$set(echo = FALSE, results = "hide")
- Global options can always be overridden by any specific options that are set in at the chunk level:

  ```
  {r pressure, echo=FALSE, results = "asis"}
  ```
- Chunk caching. If you have a long document or one involving lengthy computations, then every time you want to view your document in the pretty formatted version, you need to re-compile the document, meaning you need to re- run all the computations. Chunk caching is one way to avoid these lengthy computations.

  `cache = TRUE`
- Including a call to `sessionInfo()` at the end of each report written in R (perhaps with markdown or knitr) can be useful for communicating to the reader what type of environment is needed to reproduce the contents of the report.

References