Joe's R Guide

Joe Mirza

2021-06-04

Contents

Chapter 1

R Studio Environment

Understanding the R Studio environment.

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
library(tinytex)
library(tidyverse)
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.org/tinytex/.

1.1 R Studio vs. R and R Markdown

The file type this is being written in is called R Markdown. It's a Markdown file that
lets you insert and run R code in two types of places that are…not Markdown.

The first type is a code chunk. It looks like this:

````
```{r}

```
````

This is the most minimal sort of chunk. Three backticks, followed by r in brackets, ending in three backticks. Other instructions can follow the `r`. The most commonly used ones are:

1. Instructions that suppress certain types of content from being displayed (e.g. `echo = FALSE` means prevents code, but not the results from appearing in the finished file.)
2. Captions for graphical results using `fig.cap = "..."`.
3. The ability to set the size and layout of the images. Much more about this in plotting.

Link to specifics here: Code Chunks

The other way you can insert R code into your markdown is 'inline'. Say you had a numeric variable you'd calculated above called `varname` and wanted to use the value inline. You'd do that like so:

```
`r varname`
```

Where the numeric value of `varname` would be inserted in that location.

1.2 RStudio and the File System

- Get the working directory: `getwd()`
- Set the working directory: `setwd("/home/jovyan/RBridge/")`
- Note that by hitting backslash in rstudio, it will provide you options for the next directory in the file hierarchy
- List of files in current working directory: `list.files()`

1.3 `ls()` vs. `list.files()`

- `ls()` gives you the list of variables in the global environment.
- If you want to clear the local environmental variables, you would type `rm(list = ls())`. You probably could be forgiven for thinking it would've been `rm(ls())`, but no dice!
- Note that `ls()` is different from `list.files()`. `list.files()` lists the files in the *working directory.* `ls()` provides the variables in *working memory.*

1.4 Handy Shortcuts

Changing Focus between Source and Console Panels

- To change focus to the source (where you write the code as full-fledged scripts): `ctrl-1`
- To change focus to the console (where you see the output or write one-off commands): `ctrl-2`

What kind of object is this? (e.g. data.frame, data.table, vector?)

```
typeof(squirrel_subset)
```

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

```
class(squirrel_subset)
```

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

Comment/Uncomment Code:

```
`shift + command + C`
```

Keyboard shortcut to insert a code chunk:

````
`Cmd + Option + I`

Gives you an empty code chunk:

```{r}

```
````

Also note how one can display the code for a code chunk and not the result.(Look in the source code to see how.)

Getting Help

- You can use `?` before a command to get help on a topic. For example: `?geom_point` returns a help page for that topic.
- You can also type `help(geom_point)` to achieve the same effect.

Installing and Loading Packages

- To install a package (note the quotes around `zoo`): `install.packages('zoo')`
- To load that package and make it available (no quotes): `library(zoo)`
- The `::` specifies the package an object comes from, for example: `dplyr::mutate()`

Clearing the console: `Ctrl-L`.

Chapter 2

Summary & Overview Functions

Basic dataframe for some of the examples below.

```
d <- data.frame(
  id = 1:1000,
  x = rnorm(1000, mean = 0, sd = 1),
  y = rnorm(1000, mean = 10, sd = 2),
  color = sample(c('red', 'blue'), size = 1000, replace = TRUE)
)
```

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter 2. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter 4.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 2.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 2.1.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

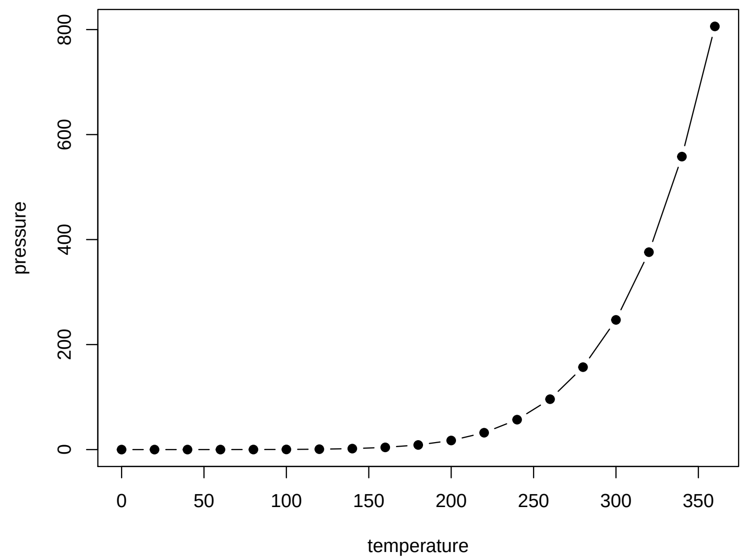


Figure 2.1: Here is a nice figure!

Table 2.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

You can write citations, too. For example, we are using the **bookdown** package (Xie, 2021) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

2.1 RStudio Cheatsheets (master list)

These seem handy.

Some functions that can give you a sense of the data you're working with.

List of these functions:
- summary - glimpse

- table
- dim
- nrow
- ncol
- count
- names

2.2 summary

Provides a summary of each data series in a dataframe. Good for getting a sense of the range of the data as well as outliers and missing values.

```
summary(nyc_license)
```

```
##  animal_name        animal_gender      animal_birth_month     breed_rc
##  Length:122203      Length:122203      Min.   :1999-01-01   Length:122203
##  Class :character   Class :character   1st Qu.:2007-10-01   Class :character
##  Mode  :character   Mode  :character   Median :2011-05-01   Mode  :character
##                                        Mean   :2010-09-05
##                                        3rd Qu.:2014-05-01
##                                        Max.   :2017-03-01
##
##    borough             zip_code     community_district census_tract2010
##  Length:122203      Min.   :  121   Min.   :101.0      Min.   :     1
##  Class :character   1st Qu.:10029   1st Qu.:108.0      1st Qu.:   126
##  Mode  :character   Median :10465   Median :302.0      Median :   251
```

```
##                        Mean   :10678   Mean   :265.2      Mean   :  7339
##                        3rd Qu.:11228   3rd Qu.:403.0      3rd Qu.:   912
##                        Max.   :94608   Max.   :595.0      Max.   :157903
##                                        NA's   :3341       NA's   :3341
##      nta               city_council_district congressional_district
##  Length:122203      Min.   : 1.00         Min.   : 3.00
##  Class :character   1st Qu.: 6.00         1st Qu.: 8.00
##  Mode  :character   Median :22.00         Median :11.00
##                     Mean   :22.83         Mean   :10.27
##                     3rd Qu.:37.00         3rd Qu.:12.00
##                     Max.   :51.00         Max.   :16.00
##                     NA's   :3341          NA's   :3341
##  state_senatorial_district license_issued_date  license_expired_date
##  Min.   :10.00             Min.   :2014-09-12   Min.   :2016-01-01
##  1st Qu.:18.00             1st Qu.:2015-08-24   1st Qu.:2016-10-13
##  Median :25.00             Median :2016-03-25   Median :2017-05-07
##  Mean   :23.54             Mean   :2016-02-13   Mean   :2017-05-24
##  3rd Qu.:28.00             3rd Qu.:2016-07-30   3rd Qu.:2017-09-27
##  Max.   :36.00             Max.   :2016-12-31   Max.   :2022-11-20
##  NA's   :3341              NA's   :1
```

`summary` behaves differently depending on the objects it's applied to:

Above we ran summary on the dataframe `nyc_license`. Here we can create a model `mod`. We can run the `summary` function on each and get very different results.

```r
mod <- lm(y ~ x, data = d)

summary(d)
```

```
##        id               x                   y               color
##  Min.   :   1.0   Min.   :-2.75003   Min.   : 4.356   Length:1000
##  1st Qu.: 250.8   1st Qu.:-0.60807   1st Qu.: 8.563   Class :character
##  Median : 500.5   Median : 0.02130   Median : 9.988   Mode  :character
##  Mean   : 500.5   Mean   : 0.05543   Mean   : 9.997
##  3rd Qu.: 750.2   3rd Qu.: 0.70286   3rd Qu.:11.375
##  Max.   :1000.0   Max.   : 4.00969   Max.   :16.240
```

```r
summary(mod)
```

```
##
## Call:
## lm(formula = y ~ x, data = d)
##
```

```
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.6339 -1.4383 -0.0123  1.3899  6.2329
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  9.99781    0.06441 155.216   <2e-16 ***
## x           -0.01059    0.06491  -0.163     0.87
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.034 on 998 degrees of freedom
## Multiple R-squared:  2.667e-05,  Adjusted R-squared:  -0.0009753
## F-statistic: 0.02662 on 1 and 998 DF,  p-value: 0.8704
```

2.3 print, glimpse & str

Alternative ways to see the contents of the data in more detail than `summary()` provides.

```
print(mpg)
```

```
## # A tibble: 234 x 11
##    manufacturer model    displ  year   cyl trans   drv     cty   hwy fl    class
##    <chr>        <chr>    <dbl> <int> <int> <chr>   <chr> <int> <int> <chr> <chr>
##  1 audi         a4         1.8  1999     4 auto(l~ f        18    29 p     comp~
##  2 audi         a4         1.8  1999     4 manual~ f        21    29 p     comp~
##  3 audi         a4         2    2008     4 manual~ f        20    31 p     comp~
##  4 audi         a4         2    2008     4 auto(a~ f        21    30 p     comp~
##  5 audi         a4         2.8  1999     6 auto(l~ f        16    26 p     comp~
##  6 audi         a4         2.8  1999     6 manual~ f        18    26 p     comp~
##  7 audi         a4         3.1  2008     6 auto(a~ f        18    27 p     comp~
##  8 audi         a4 quat~   1.8  1999     4 manual~ 4        18    26 p     comp~
##  9 audi         a4 quat~   1.8  1999     4 auto(l~ 4        16    25 p     comp~
## 10 audi         a4 quat~   2    2008     4 manual~ 4        20    28 p     comp~
## # ... with 224 more rows
```

Note that the data type is also included.

```
glimpse(mpg)
```

```
## Rows: 234
## Columns: 11
```

```
## $ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "audi", "~
## $ model        <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~
## $ displ        <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0, 2.~
## $ year         <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1999, 200~
## $ cyl          <int> 4, 4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 8, 8, ~
## $ trans        <chr> "auto(l5)", "manual(m5)", "manual(m6)", "auto(av)", "auto~
## $ drv          <chr> "f", "f", "f", "f", "f", "f", "f", "4", "4", "4", "4", "4~
## $ cty          <int> 18, 21, 20, 21, 16, 18, 18, 18, 16, 20, 19, 15, 17, 17, 1~
## $ hwy          <int> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 25, 25, 2~
## $ fl           <chr> "p", "p", "p", "p", "p", "p", "p", "p", "p", "p", "p", "p~
## $ class        <chr> "compact", "compact", "compact", "compact", "compact", "c~
```

```r
str(mpg)
```

```
## tibble[,11] [234 x 11] (S3: tbl_df/tbl/data.frame)
##  $ manufacturer: chr [1:234] "audi" "audi" "audi" "audi" ...
##  $ model       : chr [1:234] "a4" "a4" "a4" "a4" ...
##  $ displ       : num [1:234] 1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int [1:234] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##  $ cyl         : int [1:234] 4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr [1:234] "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##  $ drv         : chr [1:234] "f" "f" "f" "f" ...
##  $ cty         : int [1:234] 18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int [1:234] 29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr [1:234] "p" "p" "p" "p" ...
##  $ class       : chr [1:234] "compact" "compact" "compact" "compact" ...
```

(I think `str()` might have some powers beyond the above to unpack R objects.)

2.4 `dim`, `row` and `col`

Shape, rows and columns of dataframe:

```r
dim(nyc_license)
```

```
## [1] 122203     14
```

```r
nrow(nyc_license)
```

```
## [1] 122203
```

```
ncol(nyc_license)
```

```
## [1] 14
```

2.5 count()

Useful count of unique combinations. Easiest to understand with an example:

```
count(mpg,manufacturer, class)
```

```
## # A tibble: 32 x 3
##    manufacturer class          n
##    <chr>        <chr>      <int>
##  1 audi         compact       15
##  2 audi         midsize        3
##  3 chevrolet    2seater        5
##  4 chevrolet    midsize        5
##  5 chevrolet    suv            9
##  6 dodge        minivan       11
##  7 dodge        pickup        19
##  8 dodge        suv            7
##  9 ford         pickup         7
## 10 ford         subcompact     9
## # ... with 22 more rows
```

chevrolet has 5 different 2 seaters across the dataset. Using `filter`, you can see these are 5 different corvettes:

```
filter(mpg,manufacturer=="chevrolet", class=="2seater" )
```

```
## # A tibble: 5 x 11
##   manufacturer model   displ  year   cyl trans    drv     cty   hwy fl    class
##   <chr>        <chr>   <dbl> <int> <int> <chr>    <chr> <int> <int> <chr> <chr>
## 1 chevrolet    corvet~   5.7  1999     8 manual(~ r        16    26 p     2seat~
## 2 chevrolet    corvet~   5.7  1999     8 auto(l4) r        15    23 p     2seat~
## 3 chevrolet    corvet~   6.2  2008     8 manual(~ r        16    26 p     2seat~
## 4 chevrolet    corvet~   6.2  2008     8 auto(s6) r        15    25 p     2seat~
## 5 chevrolet    corvet~   7    2008     8 manual(~ r        15    24 p     2seat~
```

2.6 table()

`table()` is similar to `count()`, but provides you with the counts for all possible combinations, even if the value is 0.

```
table(mpg$manufacturer, mpg$class)
```

```
##
##              2seater compact midsize minivan pickup subcompact suv
##   audi             0      15       3       0      0          0   0
##   chevrolet        5       0       5       0      0          0   9
##   dodge            0       0       0      11     19          0   7
##   ford             0       0       0       0      7          9   9
##   honda            0       0       0       0      0          9   0
##   hyundai          0       0       7       0      0          7   0
##   jeep             0       0       0       0      0          0   8
##   land rover       0       0       0       0      0          0   4
##   lincoln          0       0       0       0      0          0   3
##   mercury          0       0       0       0      0          0   4
##   nissan           0       2       7       0      0          0   4
##   pontiac          0       0       5       0      0          0   0
##   subaru           0       4       0       0      0          4   6
##   toyota           0      12       7       0      7          0   8
##   volkswagen       0      14       7       0      0          6   0
```

2.7 head() and tail() Functions in R

```
head(d, 5) # where d is dataframe
```

```
##   id          x         y color
## 1  1 -0.3644521 11.018493   red
## 2  2  1.5566929 11.939726   red
## 3  3 -0.1567763  9.658720  blue
## 4  4 -0.5055430  7.377748  blue
## 5  5  2.8971199  7.573144   red
```

```
tail(d, 5)
```

Not that by including 'results="hide" in the code chunk, the code is displayed but the results are not.

2.8 names()

`names` isn't a particularly useful overview function, but it can be useful to pull column headings in a vector, which can then be referenced like any other vector.

```
names(nyc_license)
```

```
##  [1] "animal_name"              "animal_gender"
##  [3] "animal_birth_month"       "breed_rc"
##  [5] "borough"                  "zip_code"
##  [7] "community_district"       "census_tract2010"
##  [9] "nta"                      "city_council_district"
## [11] "congressional_district"   "state_senatorial_district"
## [13] "license_issued_date"      "license_expired_date"
```

```
names(nyc_license)[3]
```

```
## [1] "animal_birth_month"
```

Chapter 3

Vectors

Primary Source:

R for Data Science: Vectors

3.1 Vector basics

In some ways, working with vectors is harder than working with dataframes and data tables. That's slightly counterintuitive, as they're the most atomic unit one works with in R. In Manning's *Practical Data Science with R' they state:

> *"R's most basic data type is the vector, or array....R is fairly unique in having no scalar types. A single number such as the number 5 is represented in R as a vector with exactly one entry (5).*

There are two types of vectors:

1. **Atomic** vectors, of which there are six types: **logical**, **integer**, **double**, **character**, **complex**, and **raw**. Integer and double vectors are collectively known as **numeric** vectors.
2. **Lists**, which are sometimes called recursive vectors because lists can contain other lists.

The chief difference between atomic vectors and lists is that atomic vectors are homogeneous, while lists can be heterogeneous. There's one other related object: NULL. NULL is often used to represent the absence of a vector (as opposed to NA which is used to represent the absence of a value in a vector).

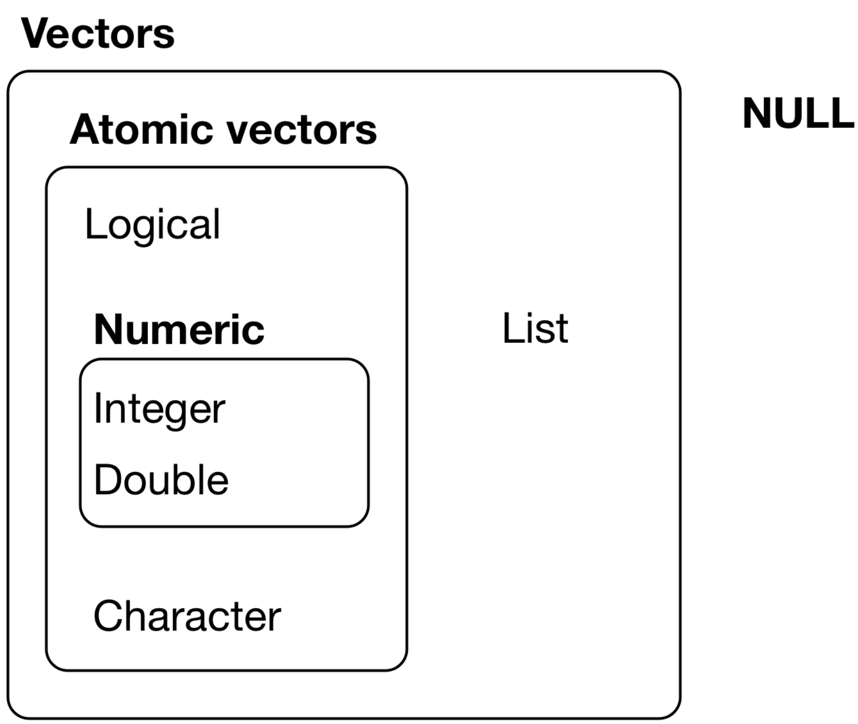


Figure 3.1: 20.1

NULL typically behaves like a vector of length 0. Figure 20.1 summarises the interrelationships.

Every vector has two key properties:

1. Its **type**, which you can determine with `typeof()`.

   ```
   typeof(letters)
   ```

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

   ```
   typeof(1:10)
   ```

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

2. Its **length**, which you can determine with `length()`.

   ```
   x <- list("a", "b", 1:10)
   length(x)
   ```

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

This chapter will introduce you to these important vectors from simplest to most complicated. You'll start with atomic vectors, then build up to lists, and finish off with augmented vectors.

3.2 Important types of atomic vector

The four most important types of atomic vector are logical, integer, double, and character.

3.3 Logical

Logical vectors are the simplest type of atomic vector because they can take only three possible values: `FALSE`, `TRUE`, and `NA`.
Logical vectors are usually constructed with comparison operators, you can also create them by hand with `c()`:

```
1:10 %% 3 == 0
```

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

```
c(TRUE, TRUE, FALSE, NA)
```

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

3.4 Numeric

Integer and double vectors are known collectively as numeric vectors. In R, numbers are doubles by default. To make an integer, place an `L` after the number:

```
typeof(1)
```

```
## [1] "double"
```

```
typeof(1L)
```

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

```
1.5L
```

```
## [1] 1.5
```

The distinction between integers and doubles is not usually important, but there are two important differences that you should be aware of:

1. Doubles are approximations.

   For example, what is square of the square root of two?

   ```
   x <- sqrt(2) ^ 2
   x
   ```

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

   ```
   x - 2
   ```

   ```
   ## [1] 4.440892e-16
   ```

This behaviour is common when working with floating point numbers: most calculations include some approximation error.

Instead of comparing floating point numbers using `==`, you should use `dplyr::near()` which allows for some numerical tolerance.

2. Integers have one special value: `NA`, while doubles have four: `NA`, `NaN`, `Inf` and `-Inf`. All three special values `NaN`, `Inf` and `-Inf` can arise during division:

```
c(-1, 0, 1) / 0
```

```
## [1] -Inf  NaN  Inf
```

Avoid using `==` to check for these other special values. Instead use the helper functions `is.finite()`, `is.infinite()`, and `is.nan()`:

	0	Inf	NA	NaN
`is.finite()`	x			
`is.infinite()`		x		
`is.na()`			x	x
`is.nan()`				x

3.5 Character

Character vectors are the most complex type of atomic vector, because each element of a character vector is a string, and a string can contain an arbitrary amount of data.

`parse_number` function. (One of several parse functions.) Could be quite handy.

```
parse_number(c("1.0", "3.5", "$1,000.00", "NA", "ABCD12234.90", "1234ABC", "A123B", "A1B2C"))
```

```
## [1]     1.0     3.5  1000.0      NA 12234.9  1234.0   123.0     1.0
```

You've already seen the most important type of implicit coercion: using a logical vector in a numeric context. In this case `TRUE` is converted to `1` and `FALSE` converted to `0`. That means the sum of a logical vector is the number of `TRUE`s, and the mean of a logical vector is the proportion of `TRUE`s:

(Also one of the first times I've seen `sample()` in this document.)

```
x <- sample(20, 100, replace = TRUE) # sample integers from 1-20, 100 times w/replacement
y <- x > 10 # create a logical vector (I think it's a vector)
sum(y)  # how many are greater than 10?
```

```
## [1] 49
```

```
mean(y) # what proportion are greater than 10?
```

```
## [1] 0.49
```

3.6 Scalars and recycling rules

As well as implicitly coercing the types of vectors to be compatible, R will also implicitly coerce the length of vectors. This is called vector **recycling**, because the shorter vector is repeated, or recycled, to the same length as the longer vector.

This is generally most useful when you are mixing vectors and "scalars".

Because there are no scalars, most built-in functions are **vectorised**, meaning that they will operate on a vector of numbers. That's why, for example, this code works:

```
sample(10) + 100
```

```
##  [1] 110 101 103 108 106 102 109 105 107 104
```

```
runif(10) > 0.5
```

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

```
5*sample(5,10,replace=TRUE)
```

```
##  [1] 10 15 20 20  5 25  5 25 25  5
```

`runif` is not a conditional `if` statement. It is a `r`andom draw from the [0,1] (uniform) interval.

3.7 Naming vectors

All types of vectors can be named. You can name them during creation with `c()`:

```
c(x = 1, y = 2, z = 4)
```

```
## x y z
## 1 2 4
```

Or after the fact with `purrr::set_names()`:

```
set_names(1:3, c("a", "b", "c"))
```

```
## a b c
## 1 2 3
```

Named vectors are most useful for subsetting, described next.

3.8 Subsetting

So far we've used `dplyr::filter()` to filter the rows in a tibble. `filter()` only works with tibble, so we'll need a new tool for vectors: `[`. `[` is the subsetting function, and is called like `x[a]`.

There are four types of things that you can subset a vector with:

1. A numeric vector containing only integers. The integers must either be all positive, all negative, or zero.

   Subsetting with positive integers keeps the elements at those positions:

   ```
   x <- c("one", "two", "three", "four", "five")
   x[c(3, 2, 5)]
   ```

   ```
   ## [1] "three" "two"   "five"
   ```

   By repeating a position, you can actually make a longer output than input:

   ```
   x[c(1, 1, 5, 5, 5, 2)]
   ```

   ```
   ## [1] "one"  "one"  "five" "five" "five" "two"
   ```

   Negative values drop the elements at the specified positions:

   ```
   x[c(-1, -3, -5)]
   ```

   ```
   ## [1] "two"  "four"
   ```

   It's an error to mix positive and negative values:

```
x[c(1, -1)]
```

```
## Error in x[c(1, -1)]: only 0's may be mixed with negative subscripts
```

2. Subsetting with a logical vector keeps all values corresponding to a `TRUE` value. This is most often useful in conjunction with the comparison functions.

```
x <- c(10, 3, NA, 5, 8, 1, NA)
# All non-missing values of x
x[!is.na(x)]
```

```
## [1] 10  3  5  8  1
```

```
# All even (or missing!) values of x
x[x %% 2 == 0]
```

```
## [1] 10 NA  8 NA
```

```
# All even AND not missing values of x
x[x %% 2 == 0 & !is.na(x)]
```

```
## [1] 10  8
```

3. If you have a named vector, you can subset it with a character vector:

```
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]
```

```
## xyz def
##   5   2
```

4. The simplest type of subsetting is nothing, `x[]`, which returns the complete `x`. This is not useful for subsetting vectors, but it is useful when subsetting matrices (and other high dimensional structures) because it lets you select all the rows or all the columns, by leaving that index blank. For example, if `x` is 2d, `x[1, ]` selects the first row and all the columns, and `x[, -1]` selects all rows and all columns except the first. *(NOTE TO SELF: This last one is quite different from python, and a little unintuitive.)*

To learn more about the applications of subsetting, reading the "Subsetting" chapter of *Advanced R*: http://adv-r.had.co.nz/Subsetting.html#applications.

There is an important variation of `[` called `[[`. `[[` only ever extracts a single element, and always drops names. It's a good idea to use it whenever you want to make it clear that you're extracting a single item, as in a for loop. The distinction between `[` and `[[` is most important for lists, as we'll see shortly.

3.9 Exercises

The expression `sum(!is.finite(x))` calculates the number of elements in the vector that are equal to missing (NA), not-a-number (NaN), or infinity (Inf).

I hadn't realized that NA = 'missing' and NaN = 'not-a-number' until now.

3.9.1 Exercise 1

Two uses for this exercise. One, it's one of my first exposures to `function()` and the first two answers contrast `[]` from `[[]]`, which seem important and easy to confuse.

Create functions that take a vector as input and returns the last value. Should you use [or [[?

```
last_value <- function(x) {
  # check for case with no length
  if (length(x)) {
    x[[length(x)]]
  } else {
    x
  }
}
last_value(numeric())
```

```
## numeric(0)
```

```
last_value(1)
```

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

```
last_value(1:10)
```

```
## [1] 10
```

3.9.2 Exercise 2

Return the elements at even numbered positions.

```
even_indices <- function(x) {
  if (length(x)) {
    x[seq_along(x) %% 2 == 0]
  } else {
    x
  }
}
even_indices(numeric())
```

```
## numeric(0)
```

```
even_indices(1)
```

```
## numeric(0)
```

```
even_indices(1:10)
```

```
## [1]  2  4  6  8 10
```

```
even_indices(letters)
```

```
##  [1] "b" "d" "f" "h" "j" "l" "n" "p" "r" "t" "v" "x" "z"
```

Great example on stackoverflow of`seq`vs.`seq_along`. It appears there's every reason to use`seq_along`instead as its behavior makes more sense.

3.9.3 Exercise 3: `seq()`

Creating a sequence in a vector:

```
x <- seq(from = 10, to = 40, by = 10)
y <- seq(from = 1, to = 4, by = 1)^2
x
```

```
## [1] 10 20 30 40
```

```
y
```

```
## [1]  1  4  9 16
```

3.9.4 Exercise 4: `rep()`

Repeating data in a vector:

```
one_to_ten_1 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
one_to_ten_2 <- 1:10
ten_to_one <- 10:1

one_to_ten_1
```

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

```
one_to_ten_2
```

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

```
ten_to_one
```

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

Using `replicate`

Two arguments: `times` or `each`:

```
letter_vector <- c('a', 'b', 'c')

rep(letter_vector, times = 3)
```

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

```
rep(letter_vector, each = 3)
```

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

```
rep(1:4, times = 1:4)
```

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

3.9.5 Exercise 5: Sampling vectors

Draws from a normal distribution:

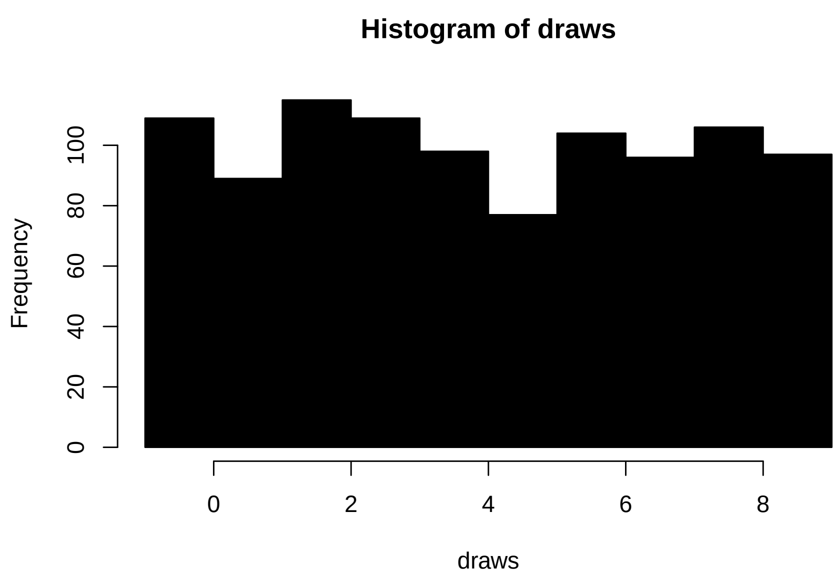
n = 8 draws from a normal distribution with mean 100 and sd = 20

```
rnorm(n = 8, mean = 100, sd = 20)
```

```
## [1]  91.66626 100.87064  94.58470 103.17574 168.69782 127.99914  68.42672
## [8]  94.25809
```

Draws from the uniform distribution

```
draws <- runif(n = 1000, min = -1, max = 9)
hist(draws, col = 'black')
```



Draw from a vector using the `sample()` function with or without replacement

```
urn <- c('red_ball', 'blue_ball', 'green_ball')
sample(x = urn, size = 4, replace = T)
```

```
## [1] "red_ball"   "green_ball" "green_ball" "red_ball"
```

To shuffle (makes use of default arguments):

```
sample(urn)
```

```
## [1] "green_ball" "blue_ball"  "red_ball"
```

3.9.6 Exercise 6: Make a `matrix()`

Make a matrix:

```
m <- matrix(data = 1:10, ncol = 2)
m
```

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

```
m[c(1,2), c(2,1)]
```

```
##      [,1] [,2]
## [1,]    6    1
## [2,]    7    2
```

3.10 Recursive vectors (lists)

Lists are a step up in complexity from atomic vectors, because lists can contain other lists. This makes them suitable for representing hierarchical or tree-like structures. You create a list with `list()`:

```
x <- list(1, 2, 3)
x
```

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

A very useful tool for working with lists is `str()` because it focuses on the **str**ucture, not the contents.

```
y <- list("a", 1L, 1.5, TRUE)
str(y)
```

```
## List of 4
##  $ : chr "a"
##  $ : int 1
##  $ : num 1.5
##  $ : logi TRUE
```

Lists can even contain other lists!

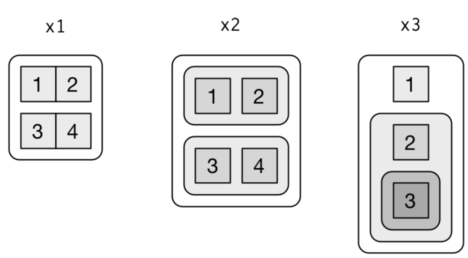
```
z <- list(list(1, 2), list(3, 4))
str(z)
```

```
## List of 2
##  $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
##  $ :List of 2
##   ..$ : num 3
##   ..$ : num 4
```

To explain more complicated list manipulation functions, it's helpful to have a visual representation of lists. For example, take these three lists:

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```

I'll draw them as follows:



There are three principles:

1. Lists have rounded corners. Atomic vectors have square corners.

2. Children are drawn inside their parent, and have a slightly darker background to make it easier to see the hierarchy.

3. The orientation of the children (i.e. rows or columns) isn't important, so I'll pick a row or column orientation to either save space or illustrate an important property in the example.

3.10.1 Subsetting

There are three ways to subset a list, which I'll illustrate with a list named `a`:

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
```

`[ ]` extracts a sub-list. The result will always be a list.

```
    str(a[1:2])
```

```
## List of 2
##  $ a: int [1:3] 1 2 3
##  $ b: chr "a string"
```

```
    str(a[4])
```

```
## List of 1
##  $ d:List of 2
##   ..$ : num -1
##   ..$ : num -5
```

As with vectors, you can subset with a logical, integer, or character vector. **`[[ ]]`extracts a single component from a list. It removes a level of hierarchy from the list.**

```
    str(a[[1]])
```

```
##  int [1:3] 1 2 3
```

```
    str(a[[4]])
```

```
## List of 2
##  $ : num -1
##  $ : num -5
```

$is a shorthand for extracting named elements of a list. It works similarly to`[[ ]]`except that you don't need to use quotes.

```
a$a
```

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

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

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

3.10.2 More vector stuff

```
v <- c(1,2,3,4,6)
```

If a vector is one-dimensional, then we can either:

- Reference a location in that vector:
  - `v[2]` Will print the value in the second position
  - `v[5]` Will print the value in the fifth position
  - `v[c(2,5)]` Will print the value in the second and fifth positions
  - `v[-1]` Will print everything but the first *note the difference between this and python*
- Pass a logical test that will print values
  - `v == 2` Tests for each value in that vector taking a particular tested, in this case, 2.

And so,

- `v[v == 2]` Will print only the values that meet the test.
- `v[v == 6]` Will not print anything
- `v[v %in% 1:3]` uses the set-based `%in%` operator which looks for existence in a range.

```
# returns by position
v[2]
```

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

```r
v[5]
```

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

```r
v[c(2,5)]
```

```
## [1] 2 6
```

```r
# everything but the first position
v[-1]
```

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

```r
v == 2
```

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

```r
v[v == 2]
```

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

```r
v %in% 1:3
```

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

```r
v[ v %in% 1:3 ]
```

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

The distinction between `[ ]` and `[[ ]]` is really important for lists, because `[[ ]]` drills down into the list while `[ ]` returns a new, smaller list.

3.10.3 Lists of condiments

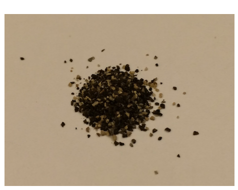
The difference between `[` and `[[` is very important, but it's easy to get confused. To help you remember, let me show you an unusual pepper shaker.

If this pepper shaker is your list `x`, then, `x[1]` is a pepper shaker containing a single pepper packet:

`x[2]` would look the same, but would contain the second packet. `x[1:2]` would be a pepper shaker containing two pepper packets.

`x[[1]]` is:

If you wanted to get the content of the pepper package, you'd need `x[[1]][[1]]`:



3.11 Augmented vectors

Atomic vectors and lists are the building blocks for other important vector types like factors and dates. I call these **augmented vectors**, because they are vectors with additional **attributes**, including class. Because augmented vectors have a class, they behave differently to the atomic vector on which they are built. In this book, we make use of four important augmented vectors:

- Factors
- Dates
- Date-times
- Tibbles

These are described below.

3.11.1 Dates and date-times

Dates in R are numeric vectors that represent the number of days since 1 January 1970.

```
x <- as.Date("1971-01-01")
unclass(x)
```

```
## [1] 365
```

```
typeof(x)
```

```
## [1] "double"
```

```
attributes(x)
```

```
## $class
## [1] "Date"
```

Chapter 4

Methods

We describe our methods in this chapter.

Chapter 5

Applications

Some *significant* applications are demonstrated in this chapter.

5.1 Example one

5.2 Example two

Chapter 6

Final Words

We have finished a nice book.

Chapter 7

Dates

Bibliography

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2021). *bookdown: Authoring Books and Technical Documents with R Markdown*. https://github.com/rstudio/bookdown, https://pkgs.rstudio.com/bookdown/.