# Data Carpentry

Brendan Knapp and Christopher Callaghan

2020-10-04

COMMON OPERATIONAL RESEARCH ENVIRONMENT
https://nps.edu/web/core

# Contents

# Welcome

# Preface

init

# Part I

# Day 1

# Chapter 1

# R and RStudio

## 1.1 R

### 1.1.1 Installation

https://cran.r-project.org/

## 1.2 RStudio

### 1.2.1 Installation

https://rstudio.com/products/rstudio/download/

# Chapter 2

# The Basics

```
"Hello, World!"
#> [1] "Hello, World!"
```

## 2.1 R as a Calculator

```
1 + 1                      # addition
#> [1] 2

1 - 1                      # subtraction
#> [1] 0

2 * 3                      # multiplication
#> [1] 6

1 + 1 * 3                  # combining operations
#> [1] 4

(1 + 1) * 3                # operator precedence
#> [1] 6

3 / 2                      # division
#> [1] 1.5

# ↓↓ pronounced "modulo"
3 %% 2                     # division remainder
#> [1] 1

4 %/% 2                    # integer division
#> [1] 2

3^2                        # exponents
#> [1] 9
```

```
4**2                      # also exponents!
#> [1] 16

Inf + 1         #
#> [1] Inf
```

If your code doesn't form a complete *expression*, then R will look for more on the next line.
Here's an example:

```
1 +
```

`1 +` isn't a complete expression, so R prompt for more code on subsequent lines. You'll see something like the following:

```
> 1 +
+
+
+
```

If this happens, press the **Esc**(scape) key (you may have to click on the Console pane first) and fix your code.

## 2.2 Fundamental Types

R has several basic data types that serve as the foundation upon which everything is built.

```
1             # double (short for double-precision floating-point number)
#> [1] 1
3.14          # also double (we can just think of them as decimals)
#> [1] 3.14

1L            # integer (`L` for "Literal" or `long` integers)
#> [1] 1

"1"           # character or string (kinda... we'll discuss later)
#> [1] "1"

TRUE          # logical (similar to `bool`s in other languages)
#> [1] TRUE
FALSE         # also logical
#> [1] FALSE

4i            # complex (we're never going to use these)
#> [1] 0+4i
```

Like most programming languages, R lets us mix *comments* into our code. Anything that follows `#` on the same line is ignored by R.
Comments enable us to annotate our work or temporarily (hopefully) disable lines of code.

```
-1 * -1000 # a negative number times a negative is positive
#> [1] 1000

# TRUE + FALSE # felt cute, might un-comment later
```

Leverage comments to communicate with humans! They're an opportunity for explaining *what* something does and (often more-importantly) *why* something works or is necessary.
Since comments are ubiquitous, it's worth pointing out two common conventions:

```
# TODO(CC): CC (initials) is going to implement some behavior
# FIXME(BK): BK is going to fix some problem
```

## 2.3 Variables

R's assignment operator is `<-`.

```
my_first_var <- "referring to data w/ names is handy!"
my_first_var
#> [1] "referring to data w/ names is handy!"
```

We can also use `=` like many other languages, but we *highly* discourage this (especially starting out) because we use `=` elsewhere. If you stick to `<-`, you'll never have to guess where you've assigned variables or rely on context clues to predict `=`'s intended purpose or behavior.

You should always prefer descriptive variable names so that others can more easily understand your code. Most of the time, the other person will just be you in the future.

```
length <- 2
width <- 4

area <- length * width
area
#> [1] 8
```

## 2.4 Multiple Values

We'll almost always need to deal with more than one value, so R let's us `c()`ombine values.

```
c(1, 2, 3, 4, 5)
#> [1] 1 2 3 4 5
```

We won't get into the nitty-gritty details just yet, but we typically call a collection of values of the same type (*homogeneous*) a `vector`.

R is special for a few reasons and having native `vectors` is definitely one of them. Understanding how they work is fundamental to writing good (and fast!) code.

```
c(1, 2, 3, 4, 5, 6)        # `c()` is short for "combine"
#> [1] 1 2 3 4 5 6

1:6                        # `:` lets us create sequences
#> [1] 1 2 3 4 5 6

my_first_vector <- -5:5 # we'll explain `vector`s later,
my_first_vector
#>  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

## 2.5 Functions

```
# ↓ name of function
sqrt(x = 16)
#> [1] 4
```

R comes with some handy variables built-in , such as `letters` and `LETTERS`.

```
letters
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v"
#> [23] "w" "x" "y" "z"

LETTERS
#>  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V"
#> [23] "W" "X" "Y" "Z"
```

```
#       ↓ parameter, formal, or argument
toupper(x = letters)
#>  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V"
#> [23] "W" "X" "Y" "Z"

#           ↓↓↓↓↓↓↓ argument (always)
tolower(x = LETTERS)
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v"
#> [23] "w" "x" "y" "z"
```

We refer to `x = letters` as a *named* argument because we specify the parameter (`x`) to which we're passing our argument (`letters`), but we often don't specify the name of a parameter.

```
tolower(letters)
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v"
#> [23] "w" "x" "y" "z"
```

We can't screw up too easily since `tolower()` and `toupper()` only have one paramter (`x`), but many functions can take multiple arguments.

Let's say we have a `vector` of `unsorted_numbers`:

```
unsorted_numbers <- c(3, 2, 10, 8, 1, 4, 9, 6, 5, 7)
unsorted_numbers
#>  [1]  3  2 10  8  1  4  9  6  5  7
```

Like most languages, R has a built-in `sort()` function we can use, which works like so:

```
sort(x = unsorted_numbers)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

By default, `sort()` sorts in *ascending* order, but we oftentimes will want to sort in *descending* (or `decreasing`) order.

Rather than having a separate function called `sort_decreasing()`, we pass an argument to `sort()`'s `decreasing` parameter.

```
sort(x = unsorted_numbers, decreasing = TRUE)
#>  [1] 10  9  8  7  6  5  4  3  2  1
```

Even though `sort()` has multiple parameters, we can still skip the names if we pass our arguments *by position*.

```
sort(unsorted_numbers, TRUE)
#>  [1] 10  9  8  7  6  5  4  3  2  1
```

Considering that `x` is `sort()`'s first parameter, and `decreaing` is `sort()`'s second parameter, we can pass our arguments (`unsorted_numbers` and `TRUE`) in the same order and R will know what we meant.

We can also mix *positional* and *named* argument (and often do), but we should always prioritize *readable* code.

```
sort(unsorted_numbers, decreasing = TRUE) # good
#>  [1] 10  9  8  7  6  5  4  3  2  1

sort(x = unsorted_numbers, TRUE)          # avoid this
#>  [1] 10  9  8  7  6  5  4  3  2  1

sort(decreasing = TRUE, unsorted_numbers) # just... no
#>  [1] 10  9  8  7  6  5  4  3  2  1
```

## 2.6 Documentation

You're hopefully wondering "How could we know the order of `sort()`'s parameters?" which leads us to documentation.

If you want more information on a specific function, you should check out the documentation, which you can do with `?` or `help()`.

Here's what that looks like for `sort()`

```
?sort
```

```
help(sort) # does the same thing as `?sort`
```

sort {base} R Documentation

**Sorting or Ordering Vectors**

**Description**

Sort (or *order*) a vector or factor (partially) into ascending or descending order. For ordering along more than one variable, e.g., for sorting data frames, see `order`.

**Usage**

```
sort(x, decreasing = FALSE, ...)

## Default S3 method:
sort(x, decreasing = FALSE, na.last = NA, ...)

sort.int(x, partial = NULL, na.last = NA, decreasing = FALSE,
         method = c("auto", "shell", "quick", "radix"), index.return = FALSE)
```

**Arguments**

`x`
: for `sort` an **R** object with a class or a numeric, complex, character or logical vector. For `sort.int`, a numeric, complex, character or logical vector, or a factor.

`decreasing`
: logical. Should the sort be increasing or decreasing? For the `"radix"` method, this can be a vector of length equal to the number of arguments in `...`. For the other methods, it must be length one. Not available for partial sorting.

`...`
: arguments to be passed to or from methods or (for the default methods and objects without a class) to `sort.int`.

`na.last`
: for controlling the treatment of `NA`s. If `TRUE`, missing values in the data are put last; if `FALSE`, they are put first; if `NA`, they are removed.

`partial`
: `NULL` or a vector of indices for partial sorting.

`method`
: character string specifying the algorithm used. Not available for partial sorting. Can be abbreviated.

`index.return`
: logical indicating if the ordering index vector should be returned as well. Supported by `method == "radix"` for any `na.last` mode and data type, and the other methods when `na.last = NA` (the default) and fully sorting non-factors.

There's *a ton* of information here, but all we're interested in at the moment is the order in which we need to pass arguments to `sort()`, which we can find in the **Arguments** section.

## 2.7 Missingness and Nothingness

### 2.7.1 NA

You may have noticed the `na.last` argument in `sort()`'s documentation. R can represent nothingness with `NULL` (same as `null` or `None` in other languages), but it can also represent *unknown* or *missing* values with `NA`.

```r
unsorted_numbers_with_nas <- c(3, 2, 10, 8, NA, 1, 4, 9, NA, 6, 5, 7)
unsorted_numbers_with_nas
#>  [1]  3  2 10  8 NA  1  4  9 NA  6  5  7
```

sort()'s default behavior is na.last = NA, which simply removes any NAs.

```r
sort(unsorted_numbers_with_nas)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

If we want to keep NAs, we must specify whether sort() places them first or last.

```r
sort(unsorted_numbers_with_nas, na.last = TRUE)
#>  [1]  1  2  3  4  5  6  7  8  9 10 NA NA
sort(unsorted_numbers_with_nas, na.last = FALSE)
#>  [1] NA NA  1  2  3  4  5  6  7  8  9 10
```

### 2.7.2 NULL

For the moment, think of the difference between NA and NULL as being that vectors (like unsorted_numbers_with_nas) can have NA values but they cannot have NULL values.

If we try to put NULL in a vector, it simply disappears.

```r
c(3, 2, 10, 8, NULL, 1, 4, 9, NULL, 6, 5, 7)
#>  [1]  3  2 10  8  1  4  9  6  5  7
```

But, how do we check if something is NA or NULL?

## 2.8 Predicate Functions

A *predicate function* is a function that returns either TRUE or FALSE based on some condition the function is checking.

Predicate functions *should* use a name that expresses this intent, such as is<some condition>, any<some condition>(), or all<some condition>().

If we want check if something is NULL, we use is.null().

```r
is.null("this string isn't NULL!")
#> [1] FALSE
is.null(NULL)
#> [1] TRUE
```

R has *many* built-in predicate functions, including ones to check the basic data types that we've already seen.

```r
is.double(1)
#> [1] TRUE
is.double(1L)
#> [1] FALSE

vec_dbl <- c(8, 6, 7, 5, 3, 0, 9)
is.double(vec_dbl)
#> [1] TRUE

is.integer(1)
#> [1] FALSE
is.integer(1L)
#> [1] TRUE
vec_int <- 1:10
is.integer(vec_int)
#> [1] TRUE

is.character(3.14)
#> [1] FALSE
is.character("is it though?")
#> [1] TRUE
is.character(letters)
#> [1] TRUE

is.logical("the year 2020")
#> [1] FALSE
is.logical(TRUE)
#> [1] TRUE
is.logical(FALSE)
#> [1] TRUE
vec_lgl <- c(TRUE, FALSE, TRUE)
is.logical(vec_lgl)
#> [1] TRUE
```

Similar to `is.null()`, there's `is.na()`.

```r
is.na("not NA!")
#> [1] FALSE
is.na(NA)
#> [1] TRUE
```

Recall our variable `unsorted_numbers_with_nas`.

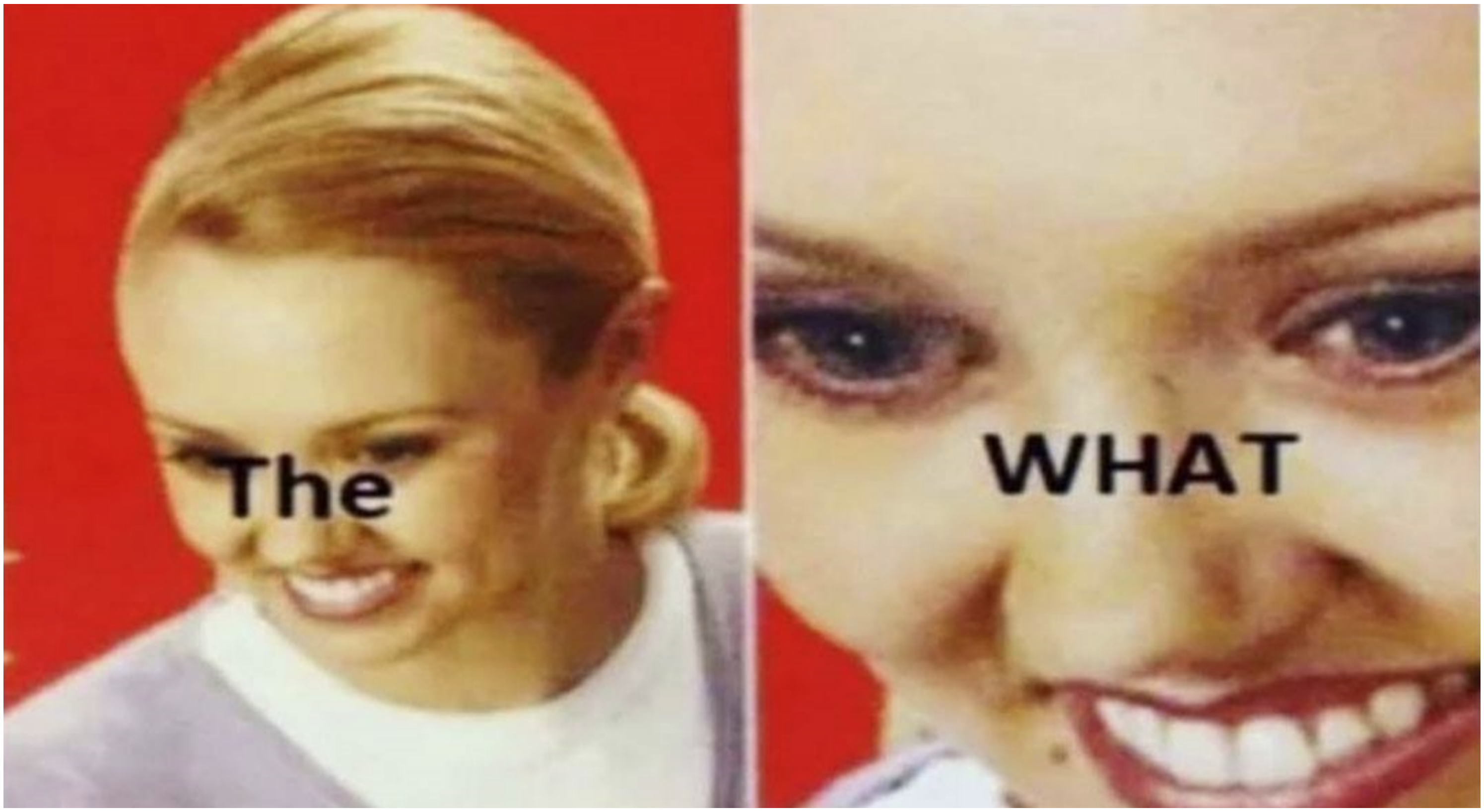
```r
unsorted_numbers_with_nas
#>  [1]  3  2 10  8 NA  1  4  9 NA  6  5  7
```

Consider the following:

- The predicate functions we've seen so far `return` either `TRUE` or `FALSE`.
- `vectors` can contain both `NA` *and* non-`NA` values.

Can you guess what `is.na() returns`?

```
is.na(unsorted_numbers_with_nas)
#>  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
```




We'll discuss accessing a `vector`'s individual elements later, but `is.na()` is what we call a *vectorized* function: a function that takes `vector` argument and operates on every element simultaneously.

## 2.9 Vectorized Functions

As high-speed R coders, we should prefer `vector`ized solutions whenever possible as they're not only idiomatic (and thus easy for other R users to understand), but they're typically several orders of magnitude faster than other solutions.

While R isn't the fastest language out there, complaints about its speed usually come from poor code, including code that "speaks" R with a C or Python accent.

The simplest way to wrap our heads around `vector`ized operations is with math. let's first make a `vector` with five `0`s in it.

We could do that like the following:

```
c(0, 0, 0, 0, 0)
#> [1] 0 0 0 0 0
```

But, good coders are *lazy* and want to (correctly) automate everything they can. With that in mind, let's `rep()`eat `0` `5` times.

```
zeros <- rep(0, length = 5)
zeros
#> [1] 0 0 0 0 0
```

For our purposes, the term *scalar* refers to an object that is a single value.

If we want to add `1` (a scalar) to every element of `zeros`, we can run `zeros + 1` or `1 + zeros`:

```
zeros + 1
#> [1] 1 1 1 1 1
```

R knows that `1` is a single value (and assumes we know what we're doing) and performs the operation (`+`) between it and every element of `zeros`. In R-speak, we refer to this behavior as *recycling*.

Let's see what happens when we add `zeros` and a `vector` containing two elements.

```
two_threes <- c(3, 3)

zeros + two_threes
#> Warning in zeros + two_threes: longer object length is not a multiple of shorter object length
#> [1] 3 3 3 3 3
```

That's probably not what you expected and R gives us a `warning()` to tell us something seems wrong.

R let's us get away with a lot of things it shouldn't, which includes

# Part II

# Day 2

# Chapter 3

# Tabular Data

- Aliases:
  - Tabular files
  - Flat
  - Delimited
- Includes:
  - Comma-Separated Value (.csv)
  - Tab-Separated Value (.tsv)

## 3.1 Basics

```
library(readr)
```

Here's some example data, modified from http://www.gapminder.org/data/

```
country,continent,year,lifeExp,pop,gdpPercap       # header/column names, separated by commas
Afghanistan,Asia,1952,28.801,8425333,779.4453145
Afghanistan,Asia,1957,30.332,9240934,820.8530296   # comma-separated values
Afghanistan,Asia,1962,31.997,10267083,853.10071
Afghanistan,Asia,1967,34.02,11537966,836.1971382
Afghanistan,Asia,1972,36.088,13079460,739.9811058
Afghanistan,Asia,1977,38.438,14880372,786.11336
Afghanistan,Asia,1982,39.854,12881816,978.0114388
Afghanistan,Asia,1987,40.822,13867957,852.3959448
```

```
csv_text <-
'country,continent,year,lifeExp,pop,gdpPercap
Afghanistan,Asia,1952,28.801,8425333,779.4453145
Afghanistan,Asia,1957,30.332,9240934,820.8530296
Afghanistan,Asia,1962,31.997,10267083,853.10071
Afghanistan,Asia,1967,34.02,11537966,836.1971382
Afghanistan,Asia,1972,36.088,13079460,739.9811058
```

```
Afghanistan,Asia,1977,38.438,14880372,786.11336
Afghanistan,Asia,1982,39.854,12881816,978.0114388
Afghanistan,Asia,1987,40.822,13867957,852.3959448'

csv_file <- tempfile(fileext = ".csv")
csv_file # a temporary file path
#> [1] "/tmp/RtmpkiXbVj/filec5e2405303c.csv"
writeLines(text = csv_text, con = csv_file) # write `csv_text` to `csv_file`
```

```
read_csv(file = csv_file)
#> Parsed with column specification:
#> cols(
#>   country = col_character(),
#>   continent = col_character(),
#>   year = col_double(),
#>   lifeExp = col_double(),
#>   pop = col_double(),
#>   gdpPercap = col_double()
#> )
#> # A tibble: 8 x 6
#>   country     continent  year lifeExp      pop gdpPercap
#>   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.
#> 2 Afghanistan Asia       1957    30.3  9240934      821.
#> 3 Afghanistan Asia       1962    32.0 10267083      853.
#> 4 Afghanistan Asia       1967    34.0 11537966      836.
#> 5 Afghanistan Asia       1972    36.1 13079460      740.
#> 6 Afghanistan Asia       1977    38.4 14880372      786.
#> 7 Afghanistan Asia       1982    39.9 12881816      978.
#> 8 Afghanistan Asia       1987    40.8 13867957      852.
```

You may encounter Tab-Delimited data where values are separated by `\t` instead of `,`. Instead of `readr::read_csv()`, we can use `readr::read_tsv()`.

```
tsv_text <-
'country\tcontinent\tyear\tlifeExp\tpop\tgdpPercap
Afghanistan\tAsia\t1952\t28.801\t8425333\t779.4453145
Afghanistan\tAsia\t1957\t30.332\t9240934\t820.8530296
Afghanistan\tAsia\t1962\t31.997\t10267083\t853.10071
Afghanistan\tAsia\t1967\t34.02\t11537966\t836.1971382
Afghanistan\tAsia\t1972\t36.088\t13079460\t739.9811058
Afghanistan\tAsia\t1977\t38.438\t14880372\t786.11336
Afghanistan\tAsia\t1982\t39.854\t12881816\t978.0114388
Afghanistan\tAsia\t1987\t40.822\t13867957\t852.3959448'

tsv_file <- tempfile(fileext = ".tsv")
writeLines(text = tsv_text, con = tsv_file)
```

```
read_tsv(file = tsv_file)
#> Parsed with column specification:
```

```
#> cols(
#>   country = col_character(),
#>   continent = col_character(),
#>   year = col_double(),
#>   lifeExp = col_double(),
#>   pop = col_double(),
#>   gdpPercap = col_double()
#> )
#> # A tibble: 8 x 6
#>   country     continent  year lifeExp      pop gdpPercap
#>   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.
#> 2 Afghanistan Asia       1957    30.3  9240934      821.
#> 3 Afghanistan Asia       1962    32.0 10267083      853.
#> 4 Afghanistan Asia       1967    34.0 11537966      836.
#> 5 Afghanistan Asia       1972    36.1 13079460      740.
#> 6 Afghanistan Asia       1977    38.4 14880372      786.
#> 7 Afghanistan Asia       1982    39.9 12881816      978.
#> 8 Afghanistan Asia       1987    40.8 13867957      852.
```

If we find ourselves reading delmited data that uses something other than `\t` or `,` to separate values, we can use `readr::read_delim()`.

```
pipe_separated_values_text <-
'country|continent|year|lifeExp|pop|gdpPercap
Afghanistan|Asia|1952|28.801|8425333|779.4453145
Afghanistan|Asia|1957|30.332|9240934|820.8530296
Afghanistan|Asia|1962|31.997|10267083|853.10071
Afghanistan|Asia|1967|34.02|11537966|836.1971382
Afghanistan|Asia|1972|36.088|13079460|739.9811058
Afghanistan|Asia|1977|38.438|14880372|786.11336
Afghanistan|Asia|1982|39.854|12881816|978.0114388
Afghanistan|Asia|1987|40.822|13867957|852.3959448'

psv_file <- tempfile(fileext = ".tsv")
writeLines(text = pipe_separated_values_text, con = psv_file)
```

```
read_delim(file = psv_file, delim = "|")
#> Parsed with column specification:
#> cols(
#>   country = col_character(),
#>   continent = col_character(),
#>   year = col_double(),
#>   lifeExp = col_double(),
#>   pop = col_double(),
#>   `gdpPercap     ` = col_double()
#> )
#> # A tibble: 8 x 6
#>   country     continent  year lifeExp      pop `gdpPercap     `
#>   <chr>       <chr>     <dbl>   <dbl>    <dbl>            <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333             779.
```

```
#> 2 Afghanistan Asia       1957    30.3  9240934            821.
#> 3 Afghanistan Asia       1962    32.0 10267083            853.
#> 4 Afghanistan Asia       1967    34.0 11537966            836.
#> 5 Afghanistan Asia       1972    36.1 13079460            740.
#> 6 Afghanistan Asia       1977    38.4 14880372            786.
#> 7 Afghanistan Asia       1982    39.9 12881816            978.
#> 8 Afghanistan Asia       1987    40.8 13867957            852.
```

```
country,continent,year,lifeExp,pop,gdpPercap       # header/column names
Afghanistan,Asia,1952,28.801,8425333,779.4453145
Afghanistan,Asia,1957,30.332,9240934,820.8530296
Afghanistan,Asia,1962,31.997,10267083,853.10071
Afghanistan,Asia,1967,34.02,11537966,836.1971382
Afghanistan,Asia,1972,36.088,13079460,739.9811058
Afghanistan,Asia,1977,38.438,14880372,786.11336
Afghanistan,Asia,1982,39.854,12881816,978.0114388
Afghanistan,Asia,1987,40.822,13867957,852.3959448
Afghanistan,,,N/A,,                                 # notice that we're missing values
```

```
csv_text <-
'country,continent,year,lifeExp,pop,gdpPercap
Afghanistan,Asia,1952,28.801,8425333,779.4453145
Afghanistan,Asia,1957,30.332,9240934,820.8530296
Afghanistan,Asia,1962,31.997,10267083,853.10071
Afghanistan,Asia,1967,34.02,11537966,836.1971382
Afghanistan,Asia,1972,36.088,13079460,739.9811058
Afghanistan,Asia,1977,38.438,14880372,786.11336
Afghanistan,Asia,1982,39.854,12881816,978.0114388
Afghanistan,Asia,1987,40.822,13867957,852.3959448
Afghanistan,,,N/A,,'

csv_file <- tempfile(fileext = ".csv")
writeLines(text = csv_text, con = csv_file)
```

## 3.2 Common Pitfalls

### 3.2.1 Incorrect Column Types

```
data_frame_from_csv <- read_csv(file = csv_file)
#> Parsed with column specification:
#> cols(
#>   country = col_character(),
#>   continent = col_character(),
#>   year = col_double(),
#>   lifeExp = col_character(),
#>   pop = col_double(),
#>   gdpPercap = col_double()
```

```
#> )
data_frame_from_csv
#> # A tibble: 9 x 6
#>   country     continent  year lifeExp      pop gdpPercap
#>   <chr>       <chr>     <dbl> <chr>      <dbl>     <dbl>
#> 1 Afghanistan Asia       1952 28.801   8425333      779.
#> 2 Afghanistan Asia       1957 30.332   9240934      821.
#> 3 Afghanistan Asia       1962 31.997  10267083      853.
#> 4 Afghanistan Asia       1967 34.02   11537966      836.
#> 5 Afghanistan Asia       1972 36.088  13079460      740.
#> 6 Afghanistan Asia       1977 38.438  14880372      786.
#> 7 Afghanistan Asia       1982 39.854  12881816      978.
#> 8 Afghanistan Asia       1987 40.822  13867957      852.
#> 9 Afghanistan <NA>         NA N/A           NA        NA
```

Notice that our `year` column says `<dbl>`, referring to it being of type `double`, yet all of our `year` values are whole numbers.

```
typeof(data_frame_from_csv$year)
#> [1] "double"
data_frame_from_csv$year
#> [1] 1952 1957 1962 1967 1972 1977 1982 1987   NA
```

We also have `"N/A"` in our `lifeExp` column, forcing R to interpret all `lifeExp` values as `characters` (`<chr>`).

```
typeof(data_frame_from_csv$lifeExp)
#> [1] "character"
data_frame_from_csv$lifeExp
#> [1] "28.801" "30.332" "31.997" "34.02"  "36.088" "38.438" "39.854" "40.822" "N/A"
```

#### 3.2.1.1 Solution

```
read_csv(
  file = csv_file,
  col_types = cols(
    country = col_character(),
    continent = col_character(),
    year = col_integer(),        # read `year` as `integer`
    lifeExp = col_double(),      # read `lifeExp` as `double`
    pop = col_double(),
    gdpPercap = col_double()
  ),
  na = c("", "N/A")              # be explicit about how `csv_file` represents missing values
)
#> # A tibble: 9 x 6
#>   country     continent  year lifeExp      pop gdpPercap
#>   <chr>       <chr>     <int>   <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.
```

```
#> 2 Afghanistan Asia       1957    30.3  9240934      821.
#> 3 Afghanistan Asia       1962    32.0 10267083      853.
#> 4 Afghanistan Asia       1967    34.0 11537966      836.
#> 5 Afghanistan Asia       1972    36.1 13079460      740.
#> 6 Afghanistan Asia       1977    38.4 14880372      786.
#> 7 Afghanistan Asia       1982    39.9 12881816      978.
#> 8 Afghanistan Asia       1987    40.8 13867957      852.
#> 9 Afghanistan <NA>         NA    NA          NA       NA
```

# Chapter 4

# Manipulating Data Frames

```
library(tidyverse, warn.conflicts = FALSE)
#> -- Attaching packages --------------------------------------------------------------- tidyverse 1.3.0 --
#> v ggplot2 3.3.2     v purrr   0.3.4
#> v tibble  3.0.3     v dplyr   1.0.2
#> v tidyr   1.1.2     v stringr 1.4.0
#> v readr   1.3.1     v forcats 0.5.0
#> -- Conflicts ------------------------------------------------------------------ tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()

df <- tibble(
  group = c("a", "a", "b", "b", "b"),
  a = c(1, 4, NA, 3, 5),
  b = c(9, NA, 8, 10, 7),
  c = c(TRUE, FALSE, NA, FALSE, TRUE),
  d = c(LETTERS[1:3], NA, LETTERS[[5]]),
  e = factor(1:5, labels = c("tiny", "small", "medium", "big", "huge")),
  f_col = c(as.Date(NA), as.Date("2020-09-23") + c(3, 2, 1, 4)),
  g_col = c(as.POSIXct("2020-09-23 00:00:00") + 1:4 * 60 * 60 * 24 * 1.1, NA),
  col_h = list(c(1, 10), c(2, NA), c(3, 8), c(4, 7), c(5, 6)),
  col_i = list(NULL, pi, month.abb[6:10], iris, as.matrix(mtcars))
)

df
#> # A tibble: 5 x 10
#>   group     a     b c     d     e      f_col      g_col               col_h    col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct>  <date>     <dttm>              <list>   <list>
#> 1 a         1     9 TRUE  A     tiny   NA         2020-09-24 02:24:00 <dbl [2~ <NULL>
#> 2 a         4    NA FALSE B     small  2020-09-26 2020-09-25 04:48:00 <dbl [2~ <dbl [1]>
#> 3 b        NA     8 NA    C     medium 2020-09-25 2020-09-26 07:12:00 <dbl [2~ <chr [5]>
#> 4 b         3    10 FALSE <NA>  big    2020-09-24 2020-09-27 09:36:00 <dbl [2~ <df[,5] [150 x ~
#> 5 b         5     7 TRUE  E     huge   2020-09-27 NA                  <dbl [2~ <dbl[,11] [32 x~
glimpse(df)
#> Rows: 5
#> Columns: 10
#> $ group <chr> "a", "a", "b", "b", "b"
```

```
#> $ a     <dbl> 1, 4, NA, 3, 5
#> $ b     <dbl> 9, NA, 8, 10, 7
#> $ c     <lgl> TRUE, FALSE, NA, FALSE, TRUE
#> $ d     <chr> "A", "B", "C", NA, "E"
#> $ e     <fct> tiny, small, medium, big, huge
#> $ f_col <date> NA, 2020-09-26, 2020-09-25, 2020-09-24, 2020-09-27
#> $ g_col <dttm> 2020-09-24 02:24:00, 2020-09-25 04:48:00, 2020-09-26 07:12:00, 2020-09-27 0...
#> $ col_h <list> [<1, 10>, <2, NA>, <3, 8>, <4, 7>, <5, 6>]
#> $ col_i <list> [NULL, 3.14, <"Jun", "Jul", "Aug", "Sep", "Oct">, <data.frame[150 x 5]>, <m...
```

## 4.1 select() Columns

### 4.1.1 by Name

```
df %>%
  select(a)
#> # A tibble: 5 x 1
#>       a
#>   <dbl>
#> 1     1
#> 2     4
#> 3    NA
#> 4     3
#> 5     5
```

```
df %>%
  select(a, c, e)
#> # A tibble: 5 x 3
#>       a c     e
#>   <dbl> <lgl> <fct>
#> 1     1 TRUE  tiny
#> 2     4 FALSE small
#> 3    NA NA    medium
#> 4     3 FALSE big
#> 5     5 TRUE  huge
```

```
df %>%
  select(b, d, f_col)
#> # A tibble: 5 x 3
#>       b d     f_col
#>   <dbl> <chr> <date>
#> 1     9 A     NA
#> 2    NA B     2020-09-26
#> 3     8 C     2020-09-25
#> 4    10 <NA>  2020-09-24
#> 5     7 E     2020-09-27
```

```
df %>%
  select(b, c, everything())
#> # A tibble: 5 x 10
#>       b c     group     a d     e      f_col      g_col               col_h    col_i
#>   <dbl> <lgl> <chr> <dbl> <chr> <fct>  <date>     <dttm>              <list>   <list>
#> 1     9 TRUE  a         1 A     tiny   NA         2020-09-24 02:24:00 <dbl [2~ <NULL>
#> 2    NA FALSE a         4 B     small  2020-09-26 2020-09-25 04:48:00 <dbl [2~ <dbl [1]>
#> 3     8 NA    b        NA C     medium 2020-09-25 2020-09-26 07:12:00 <dbl [2~ <chr [5]>
#> 4    10 FALSE b         3 <NA>  big    2020-09-24 2020-09-27 09:36:00 <dbl [2~ <df[,5] [150 x ~
#> 5     7 TRUE  b         5 E     huge   2020-09-27 NA                  <dbl [2~ <dbl[,11] [32 x~
```

```
df %>%
  select(b, c, everything(), -a)
#> # A tibble: 5 x 9
#>       b c     group d     e      f_col      g_col               col_h     col_i
#>   <dbl> <lgl> <chr> <chr> <fct>  <date>     <dttm>              <list>    <list>
#> 1     9 TRUE  a     A     tiny   NA         2020-09-24 02:24:00 <dbl [2]> <NULL>
#> 2    NA FALSE a     B     small  2020-09-26 2020-09-25 04:48:00 <dbl [2]> <dbl [1]>
#> 3     8 NA    b     C     medium 2020-09-25 2020-09-26 07:12:00 <dbl [2]> <chr [5]>
#> 4    10 FALSE b     <NA>  big    2020-09-24 2020-09-27 09:36:00 <dbl [2]> <df[,5] [150 x 5]>
#> 5     7 TRUE  b     E     huge   2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x 11]>
```

```
cols_to_select <- c("a", "c", "e")
df %>%
  select(all_of(cols_to_select))
#> # A tibble: 5 x 3
#>       a c     e
#>   <dbl> <lgl> <fct>
#> 1     1 TRUE  tiny
#> 2     4 FALSE small
#> 3    NA NA    medium
#> 4     3 FALSE big
#> 5     5 TRUE  huge
```

### 4.1.2 by Index

```
df %>%
  select(1L)
#> # A tibble: 5 x 1
#>   group
#>   <chr>
#> 1 a
#> 2 a
#> 3 b
#> 4 b
#> 5 b
```

```
df %>%
  select(1, 3, 5)
#> # A tibble: 5 x 3
#>   group     b d    
#>   <chr> <dbl> <chr>
#> 1 a         9 A    
#> 2 a        NA B    
#> 3 b         8 C    
#> 4 b        10 <NA> 
#> 5 b         7 E
```

```
df %>%
  select(2, 4, 6)
#> # A tibble: 5 x 3
#>       a c     e     
#>   <dbl> <lgl> <fct> 
#> 1     1 TRUE  tiny  
#> 2     4 FALSE small 
#> 3    NA NA    medium
#> 4     3 FALSE big   
#> 5     5 TRUE  huge
```

```
df %>%
  select(2:3, everything())
#> # A tibble: 5 x 10
#>       a     b group c     d     e      f_col      g_col               col_h    col_i
#>   <dbl> <dbl> <chr> <lgl> <chr> <fct>  <date>     <dttm>              <list>   <list>
#> 1     1     9 a     TRUE  A     tiny   NA         2020-09-24 02:24:00 <dbl [2~ <NULL>
#> 2     4    NA a     FALSE B     small  2020-09-26 2020-09-25 04:48:00 <dbl [2~ <dbl [1]>
#> 3    NA     8 b     NA    C     medium 2020-09-25 2020-09-26 07:12:00 <dbl [2~ <chr [5]>
#> 4     3    10 b     FALSE <NA>  big    2020-09-24 2020-09-27 09:36:00 <dbl [2~ <df[,5] [150 x ~
#> 5     5     7 b     TRUE  E     huge   2020-09-27 NA                  <dbl [2~ <dbl[,11] [32 x~
```

```
df %>%
  select(2:3, everything(), -1)
#> # A tibble: 5 x 9
#>       a     b c     d     e      f_col      g_col               col_h     col_i
#>   <dbl> <dbl> <lgl> <chr> <fct>  <date>     <dttm>              <list>    <list>
#> 1     1     9 TRUE  A     tiny   NA         2020-09-24 02:24:00 <dbl [2]> <NULL>
#> 2     4    NA FALSE B     small  2020-09-26 2020-09-25 04:48:00 <dbl [2]> <dbl [1]>
#> 3    NA     8 NA    C     medium 2020-09-25 2020-09-26 07:12:00 <dbl [2]> <chr [5]>
#> 4     3    10 FALSE <NA>  big    2020-09-24 2020-09-27 09:36:00 <dbl [2]> <df[,5] [150 x 5]>
#> 5     5     7 TRUE  E     huge   2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x 11]>
```

```
cols_to_select <- c(1, 3, 5)
df %>%
  select(all_of(cols_to_select))
#> # A tibble: 5 x 3
```

```
#>   group     b d
#>   <chr> <dbl> <chr>
#> 1 a         9 A
#> 2 a        NA B
#> 3 b         8 C
#> 4 b        10 <NA>
#> 5 b         7 E
```

```
cols_to_select <- c(1, 3, 5, 1000)
df %>%
  select(any_of(cols_to_select))
#> # A tibble: 5 x 3
#>   group     b d
#>   <chr> <dbl> <chr>
#> 1 a         9 A
#> 2 a        NA B
#> 3 b         8 C
#> 4 b        10 <NA>
#> 5 b         7 E
```

### 4.1.3 by Name Pattern

`contains()` selects a column if *any* part of its name contains `match=`.

```
df %>%
  select(contains(match = "col"))
#> # A tibble: 5 x 4
#>   f_col      g_col               col_h     col_i
#>   <date>     <dttm>              <list>    <list>
#> 1 NA         2020-09-24 02:24:00 <dbl [2]> <NULL>
#> 2 2020-09-26 2020-09-25 04:48:00 <dbl [2]> <dbl [1]>
#> 3 2020-09-25 2020-09-26 07:12:00 <dbl [2]> <chr [5]>
#> 4 2020-09-24 2020-09-27 09:36:00 <dbl [2]> <df[,5] [150 x 5]>
#> 5 2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x 11]>
```

`starts_with()` selects a column if its name starts with `match=`.

```
df %>%
  select(starts_with("col_"))
#> # A tibble: 5 x 2
#>   col_h     col_i
#>   <list>    <list>
#> 1 <dbl [2]> <NULL>
#> 2 <dbl [2]> <dbl [1]>
#> 3 <dbl [2]> <chr [5]>
#> 4 <dbl [2]> <df[,5] [150 x 5]>
#> 5 <dbl [2]> <dbl[,11] [32 x 11]>
```

`starts_with()` selects a column if its name ends with `match=`.

```
df %>%
  select(ends_with("_col"))
#> # A tibble: 5 x 2
#>   f_col      g_col
#>   <date>     <dttm>
#> 1 NA         2020-09-24 02:24:00
#> 2 2020-09-26 2020-09-25 04:48:00
#> 3 2020-09-25 2020-09-26 07:12:00
#> 4 2020-09-24 2020-09-27 09:36:00
#> 5 2020-09-27 NA
```

`matches()`s Selects a column if its name matches a regular expression pattern.

```
df %>%
  select(matches("(^\\w_)?col(_\\w)?"))
#> # A tibble: 5 x 4
#>   f_col      g_col               col_h     col_i
#>   <date>     <dttm>              <list>    <list>
#> 1 NA         2020-09-24 02:24:00 <dbl [2]> <NULL>
#> 2 2020-09-26 2020-09-25 04:48:00 <dbl [2]> <dbl [1]>
#> 3 2020-09-25 2020-09-26 07:12:00 <dbl [2]> <chr [5]>
#> 4 2020-09-24 2020-09-27 09:36:00 <dbl [2]> <df[,5] [150 x 5]>
#> 5 2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x 11]>
```

### 4.1.4 by Data Type

```
df %>%
  select(where(is.factor))
#> # A tibble: 5 x 1
#>   e
#>   <fct>
#> 1 tiny
#> 2 small
#> 3 medium
#> 4 big
#> 5 huge
```

```
df %>%
  select_if(is.factor)
#> # A tibble: 5 x 1
#>   e
#>   <fct>
#> 1 tiny
#> 2 small
#> 3 medium
#> 4 big
#> 5 huge
```

```
df %>%
  select(where(is.factor), f_col)
#> # A tibble: 5 x 2
#>   e      f_col
#>   <fct>  <date>
#> 1 tiny   NA
#> 2 small  2020-09-26
#> 3 medium 2020-09-25
#> 4 big    2020-09-24
#> 5 huge   2020-09-27
```

```
df %>%
  select(a, !where(is.integer))
#> # A tibble: 5 x 10
#>       a group     b c     d     e      f_col      g_col               col_h    col_i
#>   <dbl> <chr> <dbl> <lgl> <chr> <fct>  <date>     <dttm>              <list>   <list>
#> 1     1 a         9 TRUE  A     tiny   NA         2020-09-24 02:24:00 <dbl [2~ <NULL>
#> 2     4 a        NA FALSE B     small  2020-09-26 2020-09-25 04:48:00 <dbl [2~ <dbl [1]>
#> 3    NA b         8 NA    C     medium 2020-09-25 2020-09-26 07:12:00 <dbl [2~ <chr [5]>
#> 4     3 b        10 FALSE <NA>  big    2020-09-24 2020-09-27 09:36:00 <dbl [2~ <df[,5] [150 x ~
#> 5     5 b         7 TRUE  E     huge   2020-09-27 NA                  <dbl [2~ <dbl[,11] [32 x~
```

```
df %>%
  select(where(is.character) | where(is.factor))
#> # A tibble: 5 x 3
#>   group d     e
#>   <chr> <chr> <fct>
#> 1 a     A     tiny
#> 2 a     B     small
#> 3 b     C     medium
#> 4 b     <NA>  big
#> 5 b     E     huge
```

```
df %>%
  select(where(~ is.double(.) | is.list(.)))
#> # A tibble: 5 x 6
#>       a     b f_col      g_col               col_h     col_i
#>   <dbl> <dbl> <date>     <dttm>              <list>    <list>
#> 1     1     9 NA         2020-09-24 02:24:00 <dbl [2]> <NULL>
#> 2     4    NA 2020-09-26 2020-09-25 04:48:00 <dbl [2]> <dbl [1]>
#> 3    NA     8 2020-09-25 2020-09-26 07:12:00 <dbl [2]> <chr [5]>
#> 4     3    10 2020-09-24 2020-09-27 09:36:00 <dbl [2]> <df[,5] [150 x 5]>
#> 5     5     7 2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x 11]>
```

```
df %>%
  select_if(~ is.character(.x) | is.factor(.x))
#> # A tibble: 5 x 3
#>   group d     e
```

```
#>   <chr> <chr> <fct>
#> 1 a     A     tiny
#> 2 a     B     small
#> 3 b     C     medium
#> 4 b     <NA>  big
#> 5 b     E     huge
```

## 4.2 filter() Rows

### 4.2.1 by row_number()

```
df %>%
  filter(row_number() == 1)
#> # A tibble: 1 x 10
#>   group     a     b c     d     e     f_col      g_col               col_h     col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct> <date>     <dttm>              <list>    <list>
#> 1 a         1     9 TRUE  A     tiny  NA         2020-09-24 02:24:00 <dbl [2]> <NULL>
```

```
df %>%
  filter(row_number() > 1)
#> # A tibble: 4 x 10
#>   group     a     b c     d     e      f_col      g_col               col_h    col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct>  <date>     <dttm>              <list>   <list>
#> 1 a         4    NA FALSE B     small  2020-09-26 2020-09-25 04:48:00 <dbl [2~ <dbl [1]>
#> 2 b        NA     8 NA    C     medium 2020-09-25 2020-09-26 07:12:00 <dbl [2~ <chr [5]>
#> 3 b         3    10 FALSE <NA>  big    2020-09-24 2020-09-27 09:36:00 <dbl [2~ <df[,5] [150 x ~
#> 4 b         5     7 TRUE  E     huge   2020-09-27 NA                  <dbl [2~ <dbl[,11] [32 x~
```

### 4.2.2 by Name

```
df %>%
  filter(a == 2)
#> # A tibble: 0 x 10
#> # ... with 10 variables: group <chr>, a <dbl>, b <dbl>, c <lgl>, d <chr>, e <fct>,
#> #   f_col <date>, g_col <dttm>, col_h <list>, col_i <list>
```

```
df %>%
  filter(a != 2)
#> # A tibble: 4 x 10
#>   group     a     b c     d     e     f_col      g_col               col_h     col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct> <date>     <dttm>              <list>    <list>
#> 1 a         1     9 TRUE  A     tiny  NA         2020-09-24 02:24:00 <dbl [2]> <NULL>
#> 2 a         4    NA FALSE B     small 2020-09-26 2020-09-25 04:48:00 <dbl [2]> <dbl [1]>
```

```
#> 3 b          3    10 FALSE <NA>   big   2020-09-24 2020-09-27 09:36:00 <dbl [2]> <df[,5] [150 x ~
#> 4 b          5     7 TRUE  E      huge  2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x~
```

```
df %>%
  filter(c)
#> # A tibble: 2 x 10
#>   group     a     b c     d     e     f_col      g_col               col_h     col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct> <date>     <dttm>              <list>    <list>
#> 1 a         1     9 TRUE  A     tiny  NA         2020-09-24 02:24:00 <dbl [2]> <NULL>
#> 2 b         5     7 TRUE  E     huge  2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x~
```

```
df %>%
  filter(!c)
#> # A tibble: 2 x 10
#>   group     a     b c     d     e     f_col      g_col               col_h     col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct> <date>     <dttm>              <list>    <list>
#> 1 a         4    NA FALSE B     small 2020-09-26 2020-09-25 04:48:00 <dbl [2]> <dbl [1]>
#> 2 b         3    10 FALSE <NA>  big   2020-09-24 2020-09-27 09:36:00 <dbl [2]> <df[,5] [150 x ~
```

```
df %>%
  filter(a == 5, d == "E")
#> # A tibble: 1 x 10
#>   group     a     b c     d     e     f_col      g_col               col_h     col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct> <date>     <dttm>              <list>    <list>
#> 1 b         5     7 TRUE  E     huge  2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x~
```

```
df %>%
  filter(a >= 3 | f_col == "2020-09-24")
#> # A tibble: 3 x 10
#>   group     a     b c     d     e     f_col      g_col               col_h     col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct> <date>     <dttm>              <list>    <list>
#> 1 a         4    NA FALSE B     small 2020-09-26 2020-09-25 04:48:00 <dbl [2]> <dbl [1]>
#> 2 b         3    10 FALSE <NA>  big   2020-09-24 2020-09-27 09:36:00 <dbl [2]> <df[,5] [150 x ~
#> 3 b         5     7 TRUE  E     huge  2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x~
```

```
df %>%
  filter(a < 2 | c)
#> # A tibble: 2 x 10
#>   group     a     b c     d     e     f_col      g_col               col_h     col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct> <date>     <dttm>              <list>    <list>
#> 1 a         1     9 TRUE  A     tiny  NA         2020-09-24 02:24:00 <dbl [2]> <NULL>
#> 2 b         5     7 TRUE  E     huge  2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x~
```

```
df %>%
  filter(!is.na(a), !is.na(b), !is.na(d))
#> # A tibble: 2 x 10
```

```
#>   group     a     b c     d     e     f_col      g_col               col_h     col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct> <date>     <dttm>              <list>    <list>
#> 1 a         1     9 TRUE  A     tiny  NA         2020-09-24 02:24:00 <dbl [2]> <NULL>
#> 2 b         5     7 TRUE  E     huge  2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x~
```

### 4.2.3 by Type

```
df %>%
  filter(across(where(is.numeric), ~ .x >= 5))
#> # A tibble: 1 x 10
#>   group     a     b c     d     e     f_col      g_col               col_h     col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct> <date>     <dttm>              <list>    <list>
#> 1 b         5     7 TRUE  E     huge  2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x~
```

```
df %>%
  filter_if(is.numeric, ~ .x >= 5)
#> # A tibble: 1 x 10
#>   group     a     b c     d     e     f_col      g_col               col_h     col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct> <date>     <dttm>              <list>    <list>
#> 1 b         5     7 TRUE  E     huge  2020-09-27 NA                  <dbl [2]> <dbl[,11] [32 x~
```

```
df %>%
  filter_if(is.list, ~ map_lgl(.x, ~ !is.null(.x)))
#> # A tibble: 4 x 10
#>   group     a     b c     d     e      f_col      g_col               col_h    col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct>  <date>     <dttm>              <list>   <list>
#> 1 a         4    NA FALSE B     small  2020-09-26 2020-09-25 04:48:00 <dbl [2~ <dbl [1]>
#> 2 b        NA     8 NA    C     medium 2020-09-25 2020-09-26 07:12:00 <dbl [2~ <chr [5]>
#> 3 b         3    10 FALSE <NA>  big    2020-09-24 2020-09-27 09:36:00 <dbl [2~ <df[,5] [150 x ~
#> 4 b         5     7 TRUE  E     huge   2020-09-27 NA                  <dbl [2~ <dbl[,11] [32 x~
```

## 4.3 arrange() Rows

```
df %>%
  arrange(a)
#> # A tibble: 5 x 10
#>   group     a     b c     d     e      f_col      g_col               col_h    col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct>  <date>     <dttm>              <list>   <list>
#> 1 a         1     9 TRUE  A     tiny   NA         2020-09-24 02:24:00 <dbl [2~ <NULL>
#> 2 b         3    10 FALSE <NA>  big    2020-09-24 2020-09-27 09:36:00 <dbl [2~ <df[,5] [150 x ~
#> 3 a         4    NA FALSE B     small  2020-09-26 2020-09-25 04:48:00 <dbl [2~ <dbl [1]>
#> 4 b         5     7 TRUE  E     huge   2020-09-27 NA                  <dbl [2~ <dbl[,11] [32 x~
#> 5 b        NA     8 NA    C     medium 2020-09-25 2020-09-26 07:12:00 <dbl [2~ <chr [5]>
```

```
df %>%
  arrange(desc(a))
#> # A tibble: 5 x 10
#>   group     a     b c     d     e      f_col      g_col               col_h    col_i
#>   <chr> <dbl> <dbl> <lgl> <chr> <fct>  <date>     <dttm>              <list>   <list>
#> 1 b         5     7 TRUE  E     huge   2020-09-27 NA                  <dbl [2~ <dbl[,11] [32 x~
#> 2 a         4    NA FALSE B     small  2020-09-26 2020-09-25 04:48:00 <dbl [2~ <dbl [1]>
#> 3 b         3    10 FALSE <NA>  big    2020-09-24 2020-09-27 09:36:00 <dbl [2~ <df[,5] [150 x ~
#> 4 a         1     9 TRUE  A     tiny   NA         2020-09-24 02:24:00 <dbl [2~ <NULL>
#> 5 b        NA     8 NA    C     medium 2020-09-25 2020-09-26 07:12:00 <dbl [2~ <chr [5]>
```

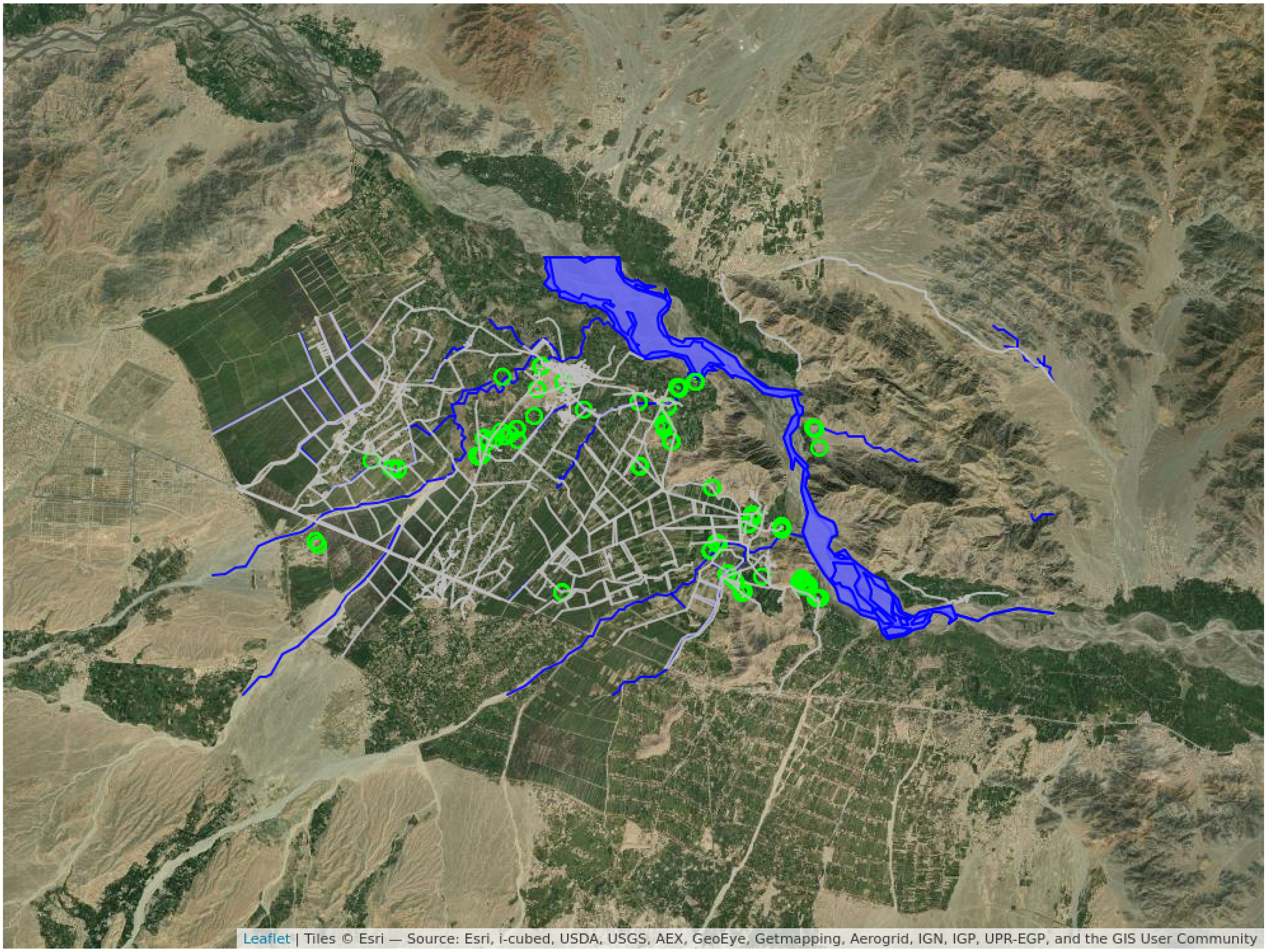
# Part III

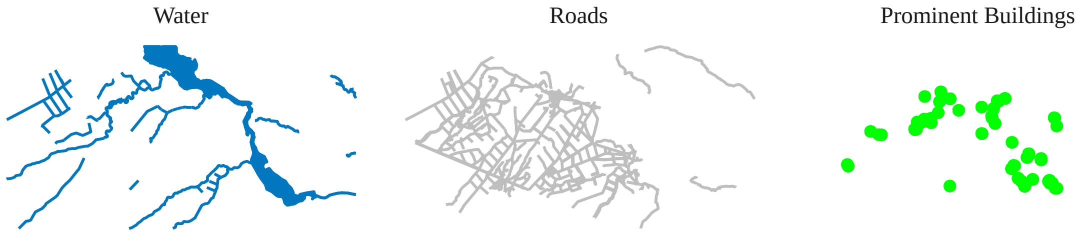
# Spatial Data

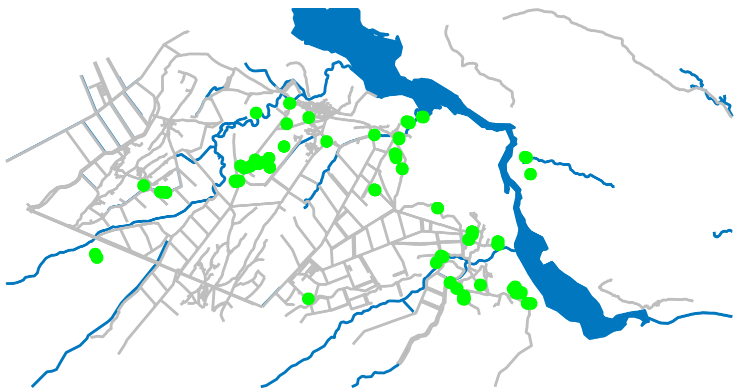
# Chapter 5

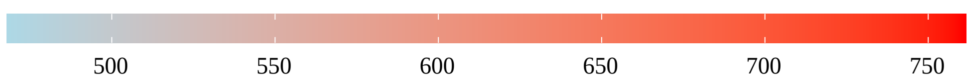
# Types of Spatial Data

```
packages <- c(
  "leaflet",  # interactive web mapping
  "osmdata",  # Open Street Maps API data
  "raster",   # obtaining administrative boundary data and spatial raster data handling
  "sf",       # spatial vector data handling
  "stars",    # {sf}'s spatio-temporal raster counterpart
  "tidyverse" # data manipulation
)

install.packages(
  packages[!sapply(packages, requireNamespace, quietly = TRUE)]
)

library(leaflet)
library(sf)
#> Linking to GEOS 3.7.1, GDAL 2.4.2, PROJ 5.2.0
library(stars)
#> Loading required package: abind
library(tidyverse)
#> -- Attaching packages --------------------------------------------------------------- tidyverse 1.3.0 --
#> v ggplot2 3.3.2     v purrr   0.3.4
#> v tibble  3.0.3     v dplyr   1.0.2
#> v tidyr   1.1.2     v stringr 1.4.0
#> v readr   1.3.1     v forcats 0.5.0
#> -- Conflicts ------------------------------------------------------------------ tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

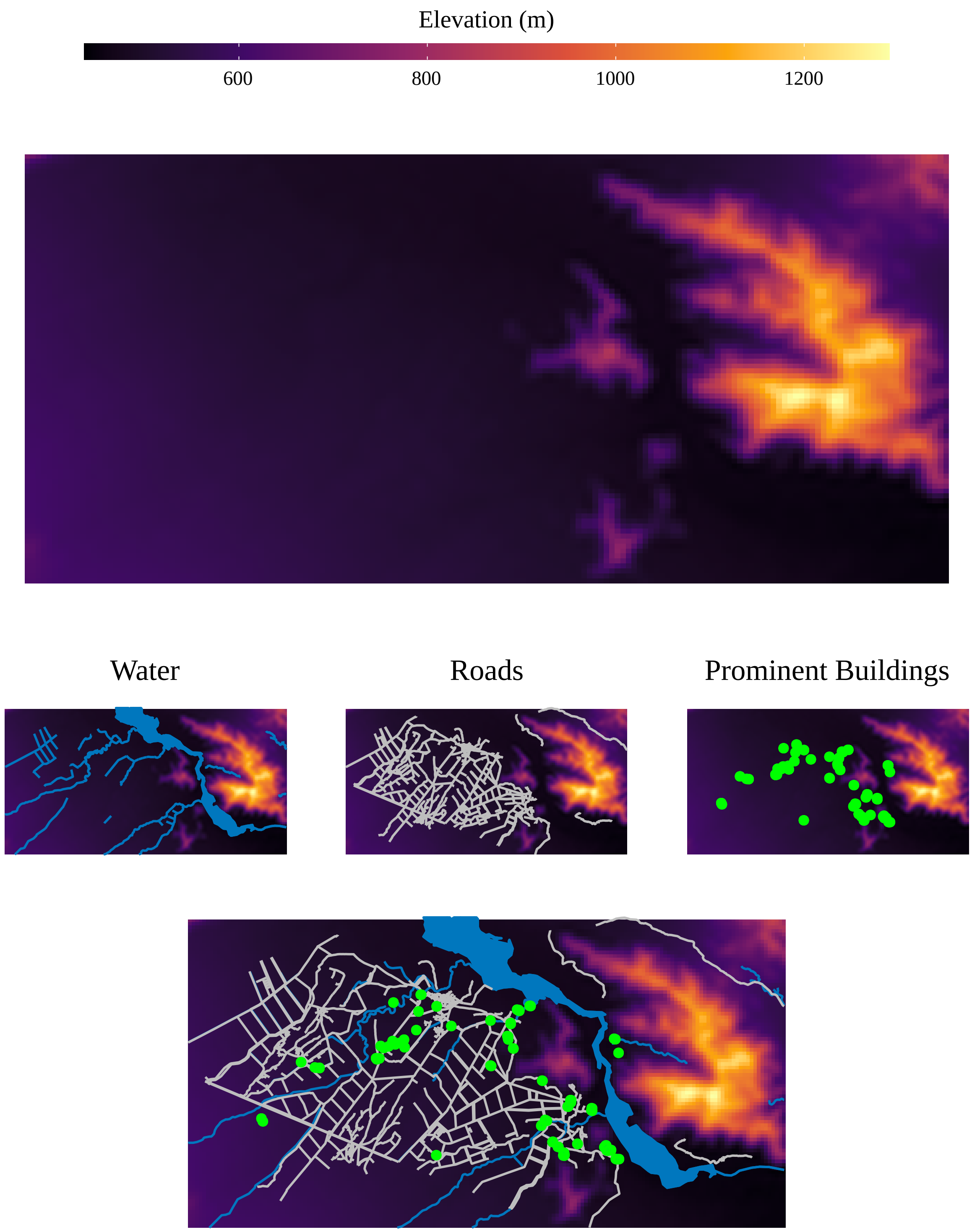

Leaflet | Tiles © Esri — Source: Esri, i-cubed, USDA, USGS, AEX, GeoEye, Getmapping, Aerogrid, IGN, IGP, UPR-EGP, and the GIS User Community






| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 489 | 487 | 485 | 499 | 543 | 584 | 585 | 551 | 518 | 501 | 499 | 503 | 502 | 508 | 532 | 549 | 535 |
| 527 | 525 | 521 | 530 | 565 | 598 | 595 | 564 | 535 | 519 | 515 | 518 | 516 | 527 | 558 | 575 | 550 |
| 569 | 582 | 588 | 597 | 618 | 639 | 639 | 618 | 591 | 570 | 557 | 556 | 561 | 574 | 593 | 590 | 549 |
| 576 | 619 | 641 | 649 | 661 | 678 | 691 | 690 | 673 | 648 | 627 | 622 | 633 | 643 | 629 | 589 | 535 |
| 553 | 608 | 632 | 630 | 635 | 656 | 685 | 709 | 716 | 710 | 705 | 707 | 711 | 694 | 638 | 567 | 513 |
| 520 | 559 | 569 | 558 | 559 | 583 | 621 | 665 | 705 | 736 | 757 | 761 | 747 | 699 | 617 | 539 | 498 |
| 492 | 509 | 511 | 504 | 507 | 530 | 565 | 617 | 682 | 735 | 755 | 741 | 711 | 662 | 591 | 529 | 495 |
| 478 | 482 | 483 | 484 | 492 | 514 | 552 | 608 | 677 | 720 | 709 | 664 | 626 | 598 | 562 | 527 | 500 |
| 475 | 476 | 477 | 481 | 489 | 516 | 564 | 625 | 677 | 688 | 645 | 582 | 544 | 532 | 524 | 514 | 501 |
| 471 | 474 | 478 | 482 | 487 | 510 | 561 | 615 | 641 | 622 | 573 | 522 | 495 | 492 | 495 | 497 | 497 |
| 471 | 473 | 478 | 486 | 491 | 508 | 547 | 581 | 579 | 548 | 514 | 492 | 482 | 482 | 483 | 486 | 492 |
| 485 | 480 | 483 | 495 | 509 | 526 | 549 | 556 | 534 | 500 | 482 | 480 | 480 | 480 | 481 | 483 | 488 |
| 525 | 521 | 515 | 516 | 525 | 536 | 543 | 533 | 508 | 482 | 474 | 478 | 481 | 481 | 482 | 484 | 485 |
| 557 | 565 | 551 | 530 | 521 | 522 | 522 | 513 | 494 | 478 | 473 | 477 | 480 | 481 | 482 | 483 | 485 |
| 534 | 549 | 537 | 512 | 502 | 509 | 517 | 515 | 497 | 477 | 471 | 476 | 479 | 480 | 482 | 482 | 483 |
| 486 | 498 | 492 | 479 | 481 | 501 | 523 | 529 | 507 | 478 | 468 | 475 | 480 | 481 | 482 | 482 | 482 |


Elevation (m)
600
800
1000
1200
Water
Roads
Prominent Buildings

# Chapter 6

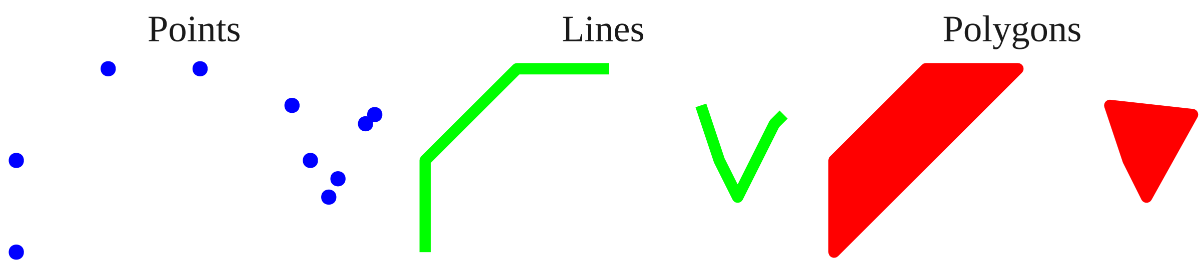
# Vector Geometries

Spatial vector data represent the world as a collection of points which, for two-dimensional data, are stored as $x$ and $y$ coordinates.

```
suppressPackageStartupMessages(library(tidyverse))
library(sf)
#> Linking to GEOS 3.7.1, GDAL 2.4.2, PROJ 5.2.0
```

```
theme_spatial <- theme_minimal(base_family = "serif") +
  theme(axis.text = element_blank(), axis.title = element_blank(),
        axis.ticks = element_blank(), panel.grid = element_blank(),
        plot.title = element_text(hjust = 0.5))

theme_set(theme_spatial)
```

Points can be joined in order to make lines, which themselves can be joined to make polygons.




```
practice_coords <- tibble(
  lng = c(-20, -20, -10, -10, 20, 20,  10,  10),
  lat = c(-20, 10, 20, -10, -10,  10,  20, -20),
  lab = c("A", "B", "C", "D", "E", "F", "G", "H"),
  grp = c("a", "a", "a", "a", "b", "b", "b", "b")
)

practice_coords
#> # A tibble: 8 x 4
#>     lng   lat lab   grp
#>   <dbl> <dbl> <chr> <chr>
#> 1   -20   -20 A     a
#> 2   -20    10 B     a
#> 3   -10    20 C     a
#> 4   -10   -10 D     a
#> 5    20   -10 E     b
#> 6    20    10 F     b
#> 7    10    20 G     b
#> 8    10   -20 H     b
```

## 6.1 POINT

`POINT` refers to the location of a single point in space.

Here, we use `st_as_sf()` to convert a regular `data.frame` into an `sf` object.

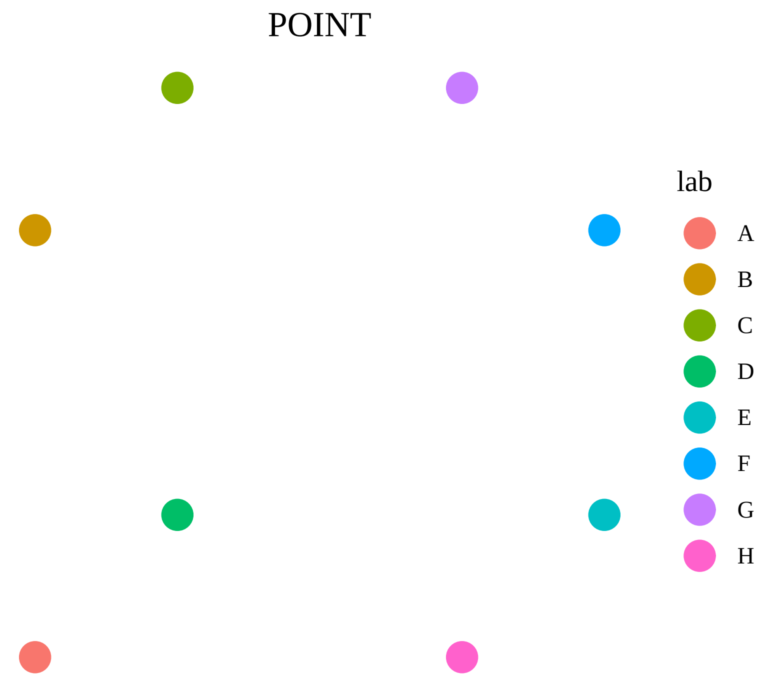
- Steps:

1. take `practice_coords`
2. convert to `sf` object with `st_as_sf()`, providing a `character vector` indicating the `names` of `practice_coords` in $(x, y)$ / $(longitude, latitude)$ order
3. `mutate()` to a add a column named `shape`, which we obtain using `st_geometry_type()`.

```
point_sf <- practice_coords %>%                      # Step 1.
  st_as_sf(coords = c(x = "lng", y ="lat")) %>%     # 2.
  mutate(shape = st_geometry_type(geometry))        # 3.

point_sf
#> Simple feature collection with 8 features and 3 fields
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: -20 ymin: -20 xmax: 20 ymax: 20
#> CRS:            NA
#> # A tibble: 8 x 4
#>   lab   grp    geometry shape
#> * <chr> <chr>   <POINT> <fct>
#> 1 A     a     (-20 -20) POINT
#> 2 B     a      (-20 10) POINT
#> 3 C     a      (-10 20) POINT
#> 4 D     a     (-10 -10) POINT
#> 5 E     b      (20 -10) POINT
#> 6 F     b       (20 10) POINT
#> 7 G     b       (10 20) POINT
#> 8 H     b      (10 -20) POINT
```

The data in our `lng` and `lat` columns are moved to a new `geometry` column.

```
ggplot(data = point_sf) +
  geom_sf(aes(color = lab), size = 5, show.legend = "point") +
  labs(title = "POINT")
```




## 6.2 MULTIPOINT

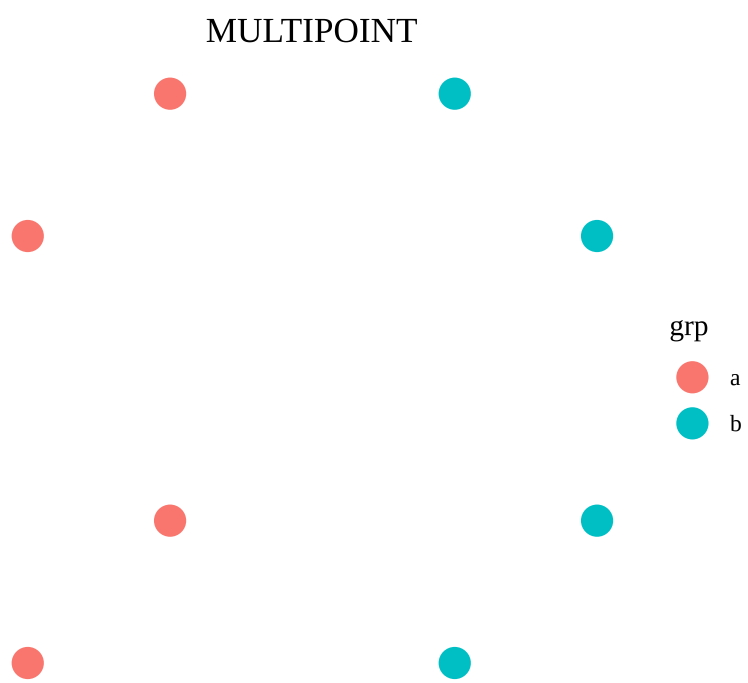
`MULTIPOINT` refers to a collection of `POINT`s.

- Steps:
  1. take `point_sf`
  2. using `group_by()`, group the rows together based on the values in their `grp` column
  3. `summarise()` each group, which combines the points of each group into a `MULTIPOINT`
  4. `mutate()` the `shape` column to change it to the new `st_geometry_type()`

```
multi_point_sf <- point_sf %>%                 # Step 1.
  group_by(grp) %>%                            # 2.
  summarise() %>%                              # 3.
  mutate(shape = st_geometry_type(geometry)) # 4.
#> `summarise()` ungrouping output (override with `.groups` argument)

multi_point_sf
#> Simple feature collection with 2 features and 2 fields
#> geometry type:  MULTIPOINT
#> dimension:      XY
#> bbox:           xmin: -20 ymin: -20 xmax: 20 ymax: 20
#> CRS:            NA
#> # A tibble: 2 x 3
#>   grp                                    geometry shape
#> * <chr>                              <MULTIPOINT> <fct>
#> 1 a     ((-20 -20), (-20 10), (-10 -10), (-10 20)) MULTIPOINT
#> 2 b         ((10 -20), (10 20), (20 -10), (20 10)) MULTIPOINT
```

Instead of the 8 separate `POINT`s with which we started, we now have 2 rows of `MULTIPOINT`s, each of which contain 4 points.

```
ggplot(data = multi_point_sf) +
  geom_sf(aes(color = grp), size = 5, show.legend = "point") +
  labs(title = "MULTIPOINT")
```




## 6.3 LINESTRING

`LINESTRING` is how we represent individual lines.

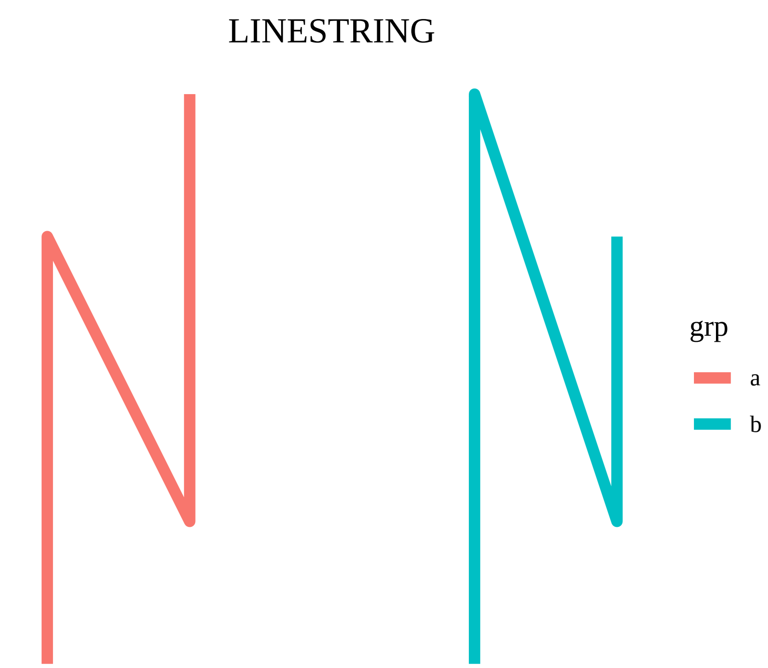
- Steps:
  1. take `multi_point_sf`
  2. cast the `geometry to= LINESTRING` using `st_cast()` 3 `mutate()` the `shape` column to change it to the new `st_geometry_type()`

```
linestring_sf <- multi_point_sf %>%          # Step 1.
  st_cast(to = "LINESTRING") %>%             # 2.
  mutate(shape = st_geometry_type(geometry)) # 3.

linestring_sf
#> Simple feature collection with 2 features and 2 fields
#> geometry type:  LINESTRING
#> dimension:      XY
#> bbox:           xmin: -20 ymin: -20 xmax: 20 ymax: 20
#> CRS:            NA
#> # A tibble: 2 x 3
```

```
#>   grp   shape                              geometry
#> * <chr> <fct>                          <LINESTRING>
#> 1 a     LINESTRING (-20 -20, -20 10, -10 -10, -10 20)
#> 2 b     LINESTRING     (10 -20, 10 20, 20 -10, 20 10)
```

Now we have 2 rows that each contain a `LINESTRING`, which was built by connecting each point to the next.

```
ggplot(data = linestring_sf) +
  geom_sf(aes(color = grp), size = 2, show.legend = "line") +
  labs(title = "LINESTRING")
```




## 6.4 MULTILINESTRING

Similar to `MULTIPOINT`s that contain multiple points, we also have `MULTILINESTRING`s.

- Steps:
    1. take `linestring_sf`
    2. `summarise()` the rows, combining them all into a single `MULTILINESTRING`
    3. `mutate()` the `shape` column to change it to the new `st_geometry_type()` and replace the `grp` column that is dropped when we `summarise()` without using `group_by()`

```
multi_linestring_sf <- linestring_sf %>%     # Step 1.
  summarise() %>%                            # 2.
  mutate(shape = st_geometry_type(geometry), # 3.
```

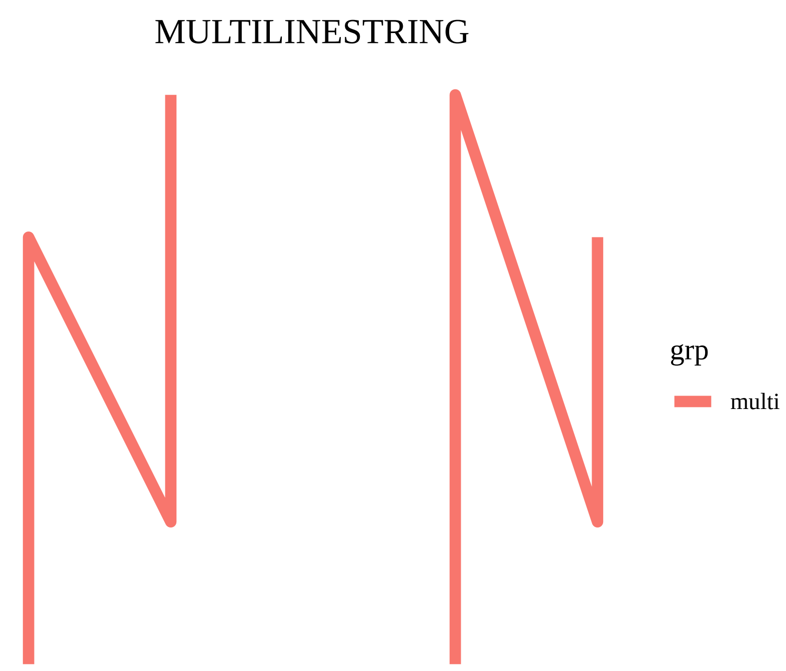


COMMON OPERATIONAL RESEARCH ENVIRONMENT
https://nps.edu/web/core

```
        grp = "multi")                      # 4.

multi_linestring_sf
#> Simple feature collection with 1 feature and 2 fields
#> geometry type:  MULTILINESTRING
#> dimension:      XY
#> bbox:           xmin: -20 ymin: -20 xmax: 20 ymax: 20
#> CRS:            NA
#> # A tibble: 1 x 3
#>                                                               geometry shape           grp
#> *                                                      <MULTILINESTRING> <fct>           <chr>
#> 1 ((-20 -20, -20 10, -10 -10, -10 20), (10 -20, 10 20, 20 -10, 20 10)) MULTILINESTRING multi
```

Now we have 2 lines embedded inside a single `MULTILINESTRING` row.

```
ggplot(data = multi_linestring_sf) +
  geom_sf(aes(color = grp), size = 2, show.legend = "line") +
  labs(title = "MULTILINESTRING")
```




## 6.5 POLYGON

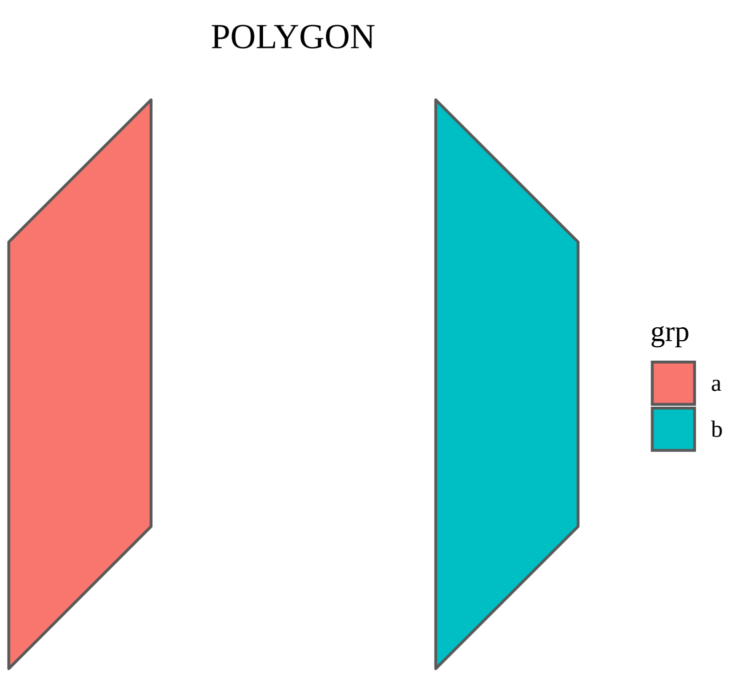
`POLYGON`s are essentially sets of lines that close to form a ring, although `POLGYON`s can also contain holes. We can easily wrap a shape around any `geometry` using `st_convex_hull()` to form a convex hull polygon.

- Steps:
  1. take `point_sf`

2. using `group_by()`, group the rows together based on the values in their `grp` column
3. `summarise()` each group, combining them into `MULTIPOINTs`
4. wrap the `MULTIPOINTs` in a polygon using `st_convex_hull()`
5. `mutate()` the `shape` column to change it to the new `st_geometry_type()`

```
polygon_sf <- point_sf %>%                   # Step 1.
  group_by(grp) %>%                          # 2.
  summarise() %>%                            # 3.
  st_convex_hull() %>%                       # 4.
  mutate(shape = st_geometry_type(geometry)) # 5.
#> `summarise()` ungrouping output (override with `.groups` argument)

polygon_sf
#> Simple feature collection with 2 features and 2 fields
#> geometry type:  POLYGON
#> dimension:      XY
#> bbox:           xmin: -20 ymin: -20 xmax: 20 ymax: 20
#> CRS:            NA
#> # A tibble: 2 x 3
#>   grp                                       geometry shape
#> * <chr>                                    <POLYGON> <fct>
#> 1 a     ((-20 -20, -20 10, -10 20, -10 -10, -20 -20)) POLYGON
#> 2 b          ((10 -20, 10 20, 20 10, 20 -10, 10 -20)) POLYGON
```

```
ggplot(data = polygon_sf) +
  geom_sf(aes(fill = grp), show.legend = "polygon") +
  labs(title = "POLYGON")
```



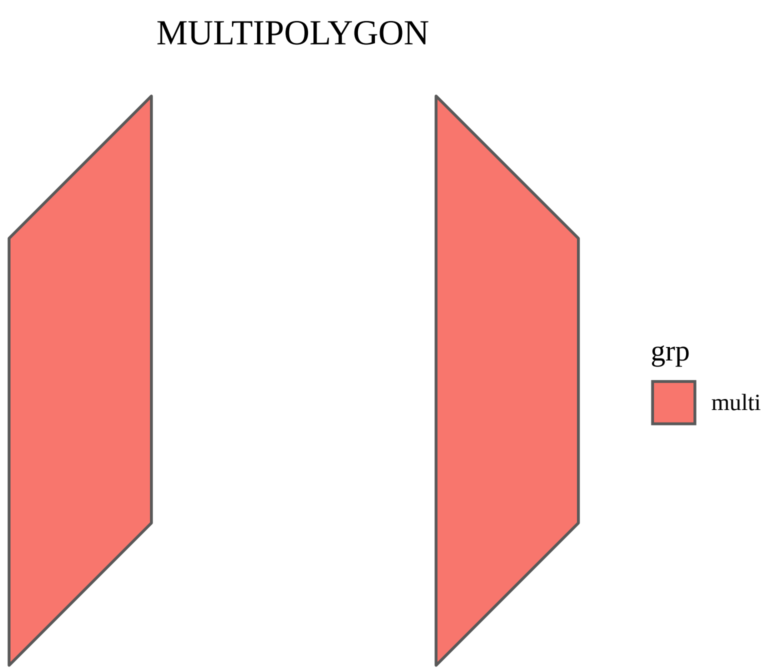
## 6.6 MULTIPOLYGON

`POLYGON`s can also be grouped together to form `MULTIPOLYGON`s.

- Steps:
  1. take `polygon_sf`
  2. `summarise()` the rows, combining them all into a single `MULTILIPOLYGON`
  3. `mutate()` the `shape` column to change it to the new `st_geometry_type()` and replace the `grp` column that is dropped when we `summarise()` without using `group_by()`

```
multi_polygon_sf <- polygon_sf %>%           # Step 1.
  summarise() %>%                            # 2.
  mutate(shape = st_geometry_type(geometry), # 3.
         grp = "multi")

multi_polygon_sf
#> Simple feature collection with 1 feature and 2 fields
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: -20 ymin: -20 xmax: 20 ymax: 20
#> CRS:            NA
#> # A tibble: 1 x 3
#>                                                                       geometry shape      grp
#> *                                                               <MULTIPOLYGON> <fct>      <chr>
#> 1 (((-20 -20, -20 10, -10 20, -10 -10, -20 -20)), ((10 -20, 10 20, 20 10, 20 ~ MULTIPOLY~ multi
```

```
ggplot(data = multi_polygon_sf) +
  geom_sf(aes(fill = grp), show.legend = "polygon") +
  labs(title = "MULTIPOLYGON")
```

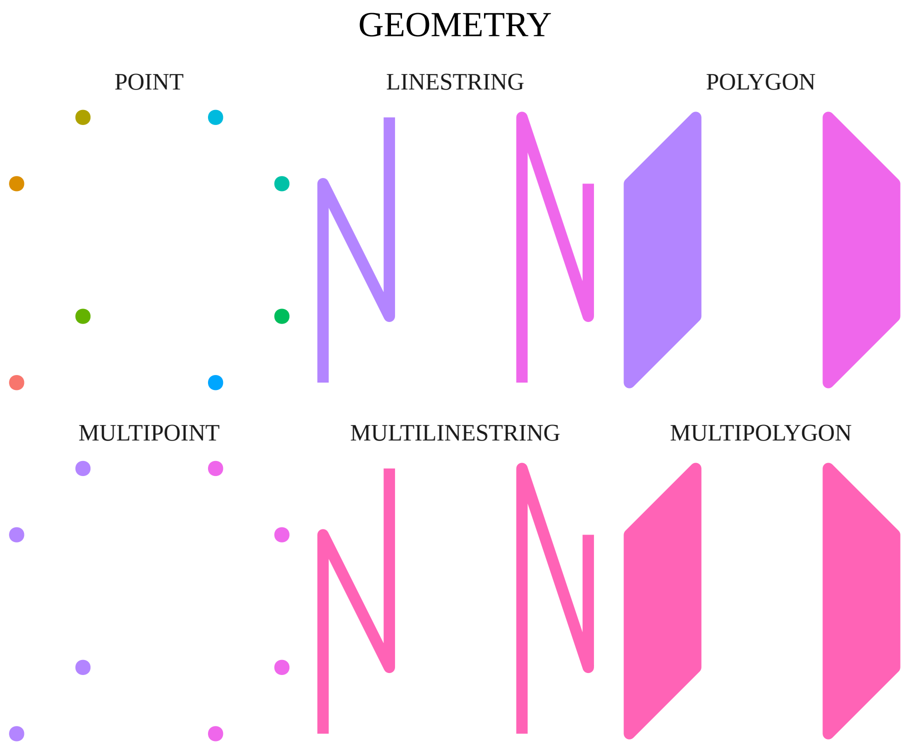


## 6.7 GEOMETRY

`GEOMETRY` is a special `geometry` type. It refers to a column of mixed geometries, i.e. we have multiple geometry types in our `geometry` column.

```
geometry_sf <- list(point_sf, multi_point_sf, linestring_sf,
                    multi_linestring_sf, polygon_sf, multi_polygon_sf) %>%
  map_if(~ "lab" %in% names(.x), select, -lab) %>%
  do.call(what = rbind) %>%
  mutate(grp = if_else(shape == "POINT", as.character(row_number()), grp))

geometry_sf
#> Simple feature collection with 16 features and 2 fields
#> geometry type:  GEOMETRY
#> dimension:      XY
#> bbox:           xmin: -20 ymin: -20 xmax: 20 ymax: 20
#> CRS:            NA
#> # A tibble: 16 x 3
#>    grp                                                          geometry shape
#>  * <chr>                                                      <GEOMETRY> <fct>
#>  1 1                                                      POINT (-20 -20) POINT
#>  2 2                                                       POINT (-20 10) POINT
#>  3 3                                                       POINT (-10 20) POINT
#>  4 4                                                      POINT (-10 -10) POINT
#>  5 5                                                       POINT (20 -10) POINT
#>  6 6                                                        POINT (20 10) POINT
#>  7 7                                                        POINT (10 20) POINT
#>  8 8                                                       POINT (10 -20) POINT
#>  9 a                         MULTIPOINT ((-20 -20), (-20 10), (-10 -10), (-10 20)) MULTIPOINT
#> 10 b                             MULTIPOINT ((10 -20), (10 20), (20 -10), (20 10)) MULTIPOINT
#> 11 a                                 LINESTRING (-20 -20, -20 10, -10 -10, -10 20) LINESTRING
#> 12 b                                     LINESTRING (10 -20, 10 20, 20 -10, 20 10) LINESTRING
#> 13 multi MULTILINESTRING ((-20 -20, -20 10, -10 -10, -10 20), (10 -20, 10 20, 20 ~ MULTILINEST~
#> 14 a                         POLYGON ((-20 -20, -20 10, -10 20, -10 -10, -20 -20)) POLYGON
#> 15 b                              POLYGON ((10 -20, 10 20, 20 10, 20 -10, 10 -20)) POLYGON
#> 16 multi MULTIPOLYGON (((-20 -20, -20 10, -10 20, -10 -10, -20 -20)), ((10 -20, 1~ MULTIPOLYGON
```

```
ggplot(data = geometry_sf) +
  geom_sf(aes(color = grp, fill = grp), size = 2, show.legend = FALSE) +
  facet_wrap(~ shape, nrow = 2) +
  labs(title = "GEOMETRY")
```


GEOMETRY
POINT
LINESTRING
POLYGON
MULTIPOINT
MULTILINESTRING
MULTIPOLYGON