Day 2

(Very rough) time plan

Friday Nov 17

13:15-14:00

- Introduction to R and RStudio
- Set up and get going
- Do Exercise 1

14:15 - 16:00

- Go through Exercise 1
- R packages and the Tidyverse
- Rectangular and tidy data
- Working with files
- Exercise 2
- Go through Exercise 2

Thursday Nov 23

09:15 - 10:30

- Manipulating data with dplyr
- Exercise 3

10:30 - 12:00

- Go through Exercise 3
- Basic plotting
- Exercise 4
- Go through exercise 4 together

14:15 - 17:00

- Programming basics
  - For loops + Ex 5 (14:15 – 15:00)
  - Ex 5 + If statements + Ex 6 (15:15 - 16:00)
  - Go through exercise 6 + wrap-up (16:00 – 17:00)

Friday Nov 24

09:15 – 12:00

- R scripts
  - Running R on the command line
  - Command line arguments
- Plotting with ggplot2 (not curriculum – brief demo + exercise)

Manipulating rectangular data with the dplyr package

The dplyr package

The **dplyr** package of the tidyverse has functions for doing some of the most common operations when working with data frames. For example:

```
mutate() # adds new variables (columns) by manipulating existing variables
select() # picks variables based on their names.
filter() # picks cases (rows) based on their values.
summarise() # reduces multiple values down to a single summary.
arrange() # changes the ordering of the rows based on variable values.
group_by() # perform operations “by group” (e.g., state).
```

Selecting columns with **select()**

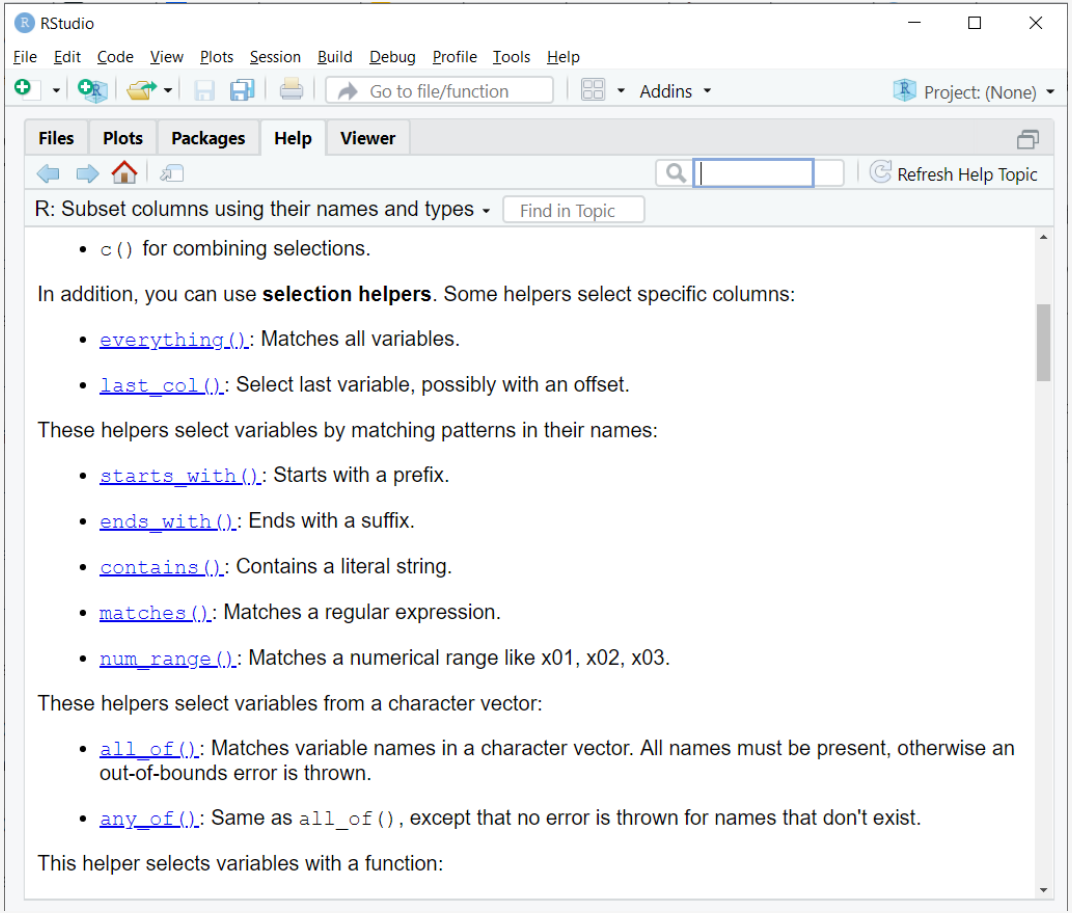
select() allows you to select different columns (variables) based on a wide range of different criteria. Check the cheat sheet or the help pages for all the options.

```
> murders <- as_tibble(murders)

> new_table <- select(murders, state, population,
total)
> new_table
# A tibble: 51 x 3
   state                population total
   <chr>                     <dbl> <dbl>
 1 Alabama                 4779736   135
 2 Alaska                   710231    19
 3 Arizona                 6392017   232
 4 Arkansas                2915918    93
 5 California             37253956  1257
 6 Colorado                5029196    65
 7 Connecticut             3574097    97
 8 Delaware                 897934    38
 9 District of Columbia     601723    99
10 Florida                19687653   669

    # ... with 41 more rows
```

Selecting columns with **select()**



Adding columns with **mutate()**

total and *population* are columns in the data. *rate* is created by mutate()

mutate() allows to add a column by doing operations on other columns in the data frame.

```
> murders <- mutate(murders, rate = total / population * 100000)
> murders
# A tibble: 51 x 6
   state                abb   region    population total  rate
   <chr>                <chr> <fct>          <dbl> <dbl> <dbl>
 1 Alabama              AL    South        4779736   135  2.82
 2 Alaska               AK    West          710231    19  2.68
 3 Arizona              AZ    West         6392017   232  3.63
 4 Arkansas             AR    South        2915918    93  3.19
 5 California           CA    West        37253956  1257  3.37
 6 Colorado             CO    West         5029196    65  1.29
 7 Connecticut          CT    Northeast    3574097    97  2.71
 8 Delaware             DE    South         897934    38  4.23
 9 District of Columbia DC    South         601723    99 16.5
10 Florida              FL    South       19687653   669  3.40
# ... with 41 more rows
```

Subsetting rows with **filter()**

filter() allows to select rows based on various criteria. E.g. select states with murder rate below or equal to 0.7.

```
> filter(murders, rate <= 0.7)
# A tibble: 5 x 6
  state          abb   region        population total  rate
  <chr>          <chr> <fct>              <dbl> <dbl> <dbl>
1 Hawaii         HI    West             1360301     7 0.515
2 Iowa           IA    North Central    3046355    21 0.689
3 New Hampshire  NH    Northeast        1316470     5 0.380
4 North Dakota   ND    North Central     672591     4 0.595
5 Vermont        VT    Northeast         625741     2 0.320
```

The “pipe”

NB2! Since 4.1.0 there is also a pipe in base R (“|>”). “|>” is largely similar to “%>%”.

NB! The “pipe” is not part of base R, but needs to be activated by loading a package (e.g. library(tidyverse)).

Just like “|” in unix/bash, the %>% (NB: look for the RStudio shortcut) symbol allows you to chain operations together. The pipe is particularly useful when using “tidyverse-style” functions (you will learn about that soon).

%>% “inserts” (you can’t see it) the left-hand side argument (e.g., the murders object) as the first argument of the function (e.g., mutate) on the right-hand side.

```
> murders %>% mutate(rate = total / population * 100000) %>%
filter(rate <= 0.7)
# A tibble: 5 x 6
  state          abb   region        population total  rate
  <chr>          <chr> <fct>              <dbl> <dbl> <dbl>
1 Hawaii         HI    West             1360301     7 0.515
2 Iowa           IA    North Central    3046355    21 0.689
3 New Hampshire  NH    Northeast        1316470     5 0.380
4 North Dakota   ND    North Central     672591     4 0.595
5 Vermont        VT    Northeast         625741     2 0.320
```

Notice how the data object (murders) is no longer the first argument in the mutate() and filter() functions.

group_by()

group_by() allows you to split the data into groups and perform operations on each group.

```
> murders %>% group_by(region)
# A tibble: 51 x 5
# Groups:   region [4]
   state                abb   region    population total
   <chr>                <chr> <fct>          <dbl> <dbl>
 1 Alabama              AL    South        4779736   135
 2 Alaska               AK    West          710231    19
 3 Arizona              AZ    West         6392017   232
 4 Arkansas             AR    South        2915918    93
                        CA    West        37253956  1257
                        CO    West         5029196    65
 7 Connecticut          CT    Northeast    3574097    97
 8 Delaware             DE    South         897934    38
 9 District of Columbia DC    South         601723    99
10 Florida              FL    South       19687653   669

# ... with 41 more rows
```

Notice the new Groups information

group_by(), then summarize

The function summarize() works particularly well on grouped data frames. Summarize can be used to quickly generate descriptive statistics.

```
> murders %>% group_by(region) %>%
summarize(count = n())
# A tibble: 4 x 2
  region        count
* <fct>         <int>
1 Northeast         9
2 South            17
3 North Central    12
4 West             13
```

group_by(), then summarize

The function summarize() works particularly well on grouped data frames. Summarize can be used to quickly generate descriptive statistics.

```
> murders %>% mutate(rate = total / population * 100000)
%>% group_by(region) %>% summarize(median_rate =
median(rate)) %>% filter(median_rate < 2.0)
# A tibble: 3 x 2
  region        median_rate
  <fct>               <dbl>
1 Northeast            1.80
2 North Central        1.97
3 West                 1.29
```

Do Exercise 3

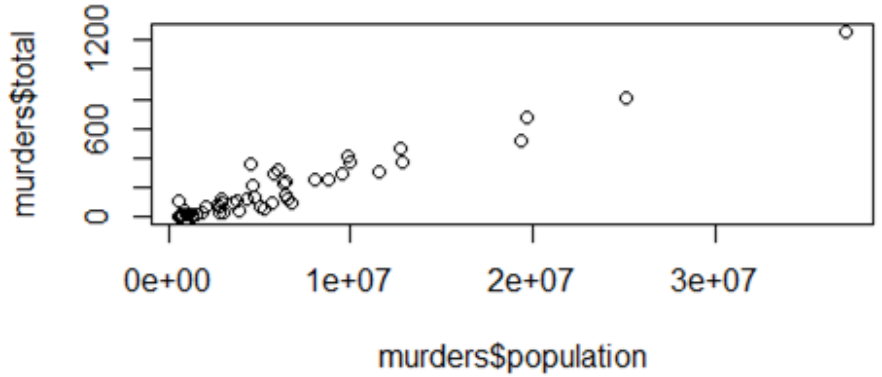
(we’ll go through it together)

Basic plotting

Basic plotting in R - scatterplot

R has several functions for making plots to quickly visualize your data. The **plot()** function can plot two variables against each other. plot() takes two arguments, x = and y = .

```
> plot(x = murders$population, y = murders$total)
```

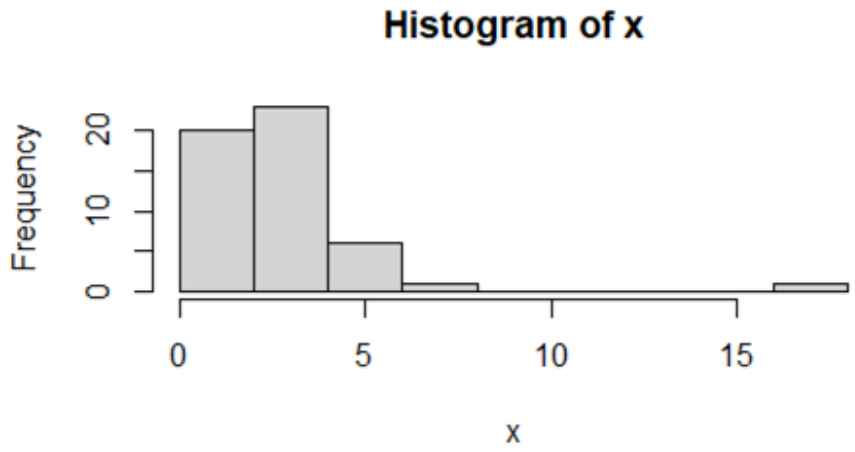


The “$” is called “the accessor” and is the way to access the different variables (columns) in data frames in base-R language.

Basic plotting in R - histogram

The **hist()** function is a quick method to get a summary of your data.

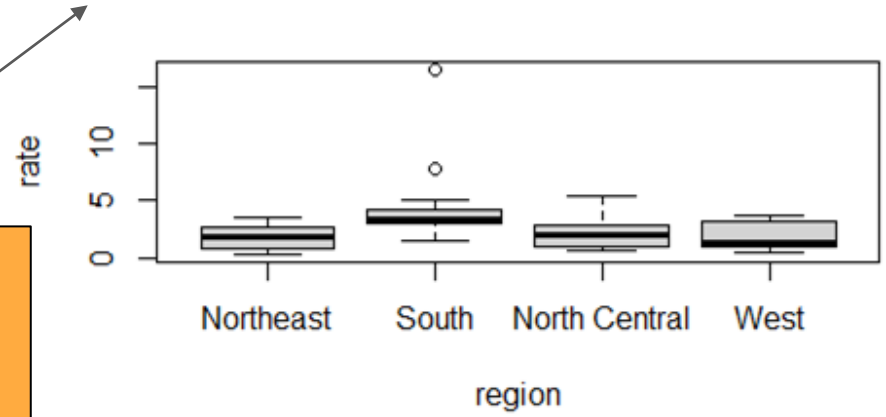
```
> x <- murders$total / murders$population*100000
> x <- hist(x)
```



Basic plotting in R - boxplot

The **boxplot()** function is great for quickly comparing groups of data.

```
murders <- mutate(murders, rate = total / population * 100000)
boxplot(rate~region, data = murders)
```



The “~” symbol (tilde) is used here to design a “formula”. The formula tells R to calculate statistics of the “rate” (e.g. median) across the different “regions”.

Do Exercise 4

(we’ll go through it together)

Iteration

In programming it's important to **reduce duplication**. A rule of thumb is to *never copy and paste the same code more than twice*.

Iteration helps you to *do the same thing to multiple inputs* (e.g. repeating the same operation on different columns, or on different datasets...).

There are a few ways to iterate in R:

- loops (for loops and while loops - we only focus on the for loop)
- The tidyverse map() functions (even more condense than for loops, but require more knowledge about R than you will get here...)

Iteration

We have this simple data frame and want to compute the median of each column. We can copy and paste the median() function like this:

```
> df
# A tibble: 10 x 4
         a       b      c       d
     <dbl>   <dbl>  <dbl>   <dbl>
 1  1.31   -0.0818 -0.316 -1.06
 2 -0.0405 -0.886  -1.30   0.185
 3  0.455  -1.50    0.799 -0.283
 4  0.0979 -0.552   0.891  1.09
 5  0.305   0.160  -0.699  1.18
 6 -1.48    0.418  -0.946 -0.580
 7 -0.242   1.12   -0.286  1.47
 8  0.900   0.136  -0.215 -1.44
 9  0.201  -0.697  -0.154  0.0178
10 -1.57    0.919   0.631  0.691
```

```
median(df$a)
#> [1] -0.2457625
median(df$b)
#> [1] -0.2873072
median(df$c)
#> [1] -0.05669771
median(df$d)
#> [1] 0.1442633
```

Remember the "accessor"?

for loop

We have this simple data frame and want to compute the median of each column. We can use a **for loop**:

```
output <- vector("double", ncol(df))  # 1. output
for (i in 1:ncol(df)) {               # 2. sequence
  output[i] <- median(df[[i]])        # 3. body
}
output
#> [1] -0.24576245 -0.28730721 -0.05669771  0.14426335
```

For loop

Let’s look at a simpler example...

```
for(i in 1:5){
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
```

for loop

Every for loop has three components:

The **output**

The **sequence**

The **body**

```
output <- vector("double", ncol(df))  # 1. output
for (i in 1:ncol(df)) {               # 2. sequence
  output[i] <- median(df[[i]])        # 3. body
}
output
#> [1] -0.24576245 -0.28730721 -0.05669771  0.14426335
```

for loop

Before we start the loop we need to allocate sufficient space for the output (if not it can be very slow).

A general way of creating an empty vector of given length is the vector() function. It has two arguments: the type of the vector (“logical”, “integer”, “double”, “character”, etc) and the length of the vector.

```
output <- vector("double", ncol(df))  # 1. output
```

The vector data type

In R a vector is a kind of list of elements (list is actually something else in R, but never mind…). Vectors are created with the function c().

```
> x <- c(1, 2, 3)
> x
[1] 1 2 3
> x[2]
[1] 2
```

The vector data type

In R a vector is a kind of list of elements (list is actually something else in R, but never mind…). Vectors are created with the function c().

Vectors can be numeric, character, logic, and more.

```
> murders$state
 [1] "Alabama"              "Alaska"               "Arizona"
 [4] "Arkansas"             "California"           "Colorado"
 [7] "Connecticut"          "Delaware"             "District of Columbia"
[10] "Florida"              "Georgia"              "Hawaii"
[13] "Idaho"                "Illinois"             "Indiana"
[16] "Iowa"                 "Kansas"               "Kentucky"

> class(murders$state)

[1] "character"

> murders$total
 [1]  135   19  232   93 1257   65   97   38   99  669  376    7   12  364  142   21
[17]   63  116  351   11  293  118  413   53  120  321   12   32   84    5  246   67
[33]  517  286    4  310  111   36  457   16  207    8  219  805   22    2  250   93
[49]   27   97    5

> class(murders$total)

[1] "numeric"
```

Subsetting

Notice the numbers in brackets in the output. These give hint about how to retrieve certain elements of a vector. This is called subsetting, or indexing (this works on many other data types in R as well).

```
> murders$state[1]
[1] "Alabama"
> murders$state[3:6]
[1] "Arizona"    "Arkansas"   "California" "Colorado"
> murders$state[c(6, 3, 5, 4)]
[1] "Colorado"   "Arizona"    "California" "Arkansas"
> murders$state[c(-6, -3, -5, -4)]
 [1] "Alabama"              "Alaska"               "Connecticut"
 [4] "Delaware"             "District of Columbia" "Florida"
...
```

for loop

Before we start the loop we need to allocate sufficient space for the output (if not it can be very slow).

A general way of creating an empty vector of given length is the vector() function. It has two arguments: the type of the vector (“logical”, “integer”, “double”, “character”, etc) and the length of the vector.

```
output <- vector("double", ncol(df))  # 1. output
```

The **result of the loop** (output) is **a vector** of type “double” (decimal numbers) and the same length as the number of columns in the data frame.

The sequence

The sequence determines **what to loop over**. (1:ncol() will generate a sequence of numbers from 1 to the number of columns in the dataframe – 1, 2, 3, 4 in this case).

“i” can be whatever character or word you like.

```
for (i in 1:ncol(df))                    # 2. sequence
```

The loop will iterate for the same number of times as there are columns in the data frame (i.e. 4 columns). “i” will be updated for every iteration (i.e. first iteration i = 1, second iteration i = 2, third i = 3 and fourth i = 4.

The body

The body is the code that does the work. It's run repeatedly, each time with a different value for "i".

```
output[i] <- median(df[[i]])      # 3. body
```

`df[[i]]` extracts column "i" as a vector of numbers. The function median() calculates the median of these numbers. The median is then entered into position "i" in the "output" vector. (the double brackets are needed to extract only the values in the column, and not the entire column with header).

The first iteration of the loop will be:

```
output[1] <- median(df[[1]])
```

Do Exercise 5

(we’ll go through it together)

if statements

if statements (conditional expressions)

Conditional expressions are one of the basic features of programming. They are used for what is called *flow control*. The most common conditional expression is the if statement (or the if-else statement)

Syntax of the if-statement

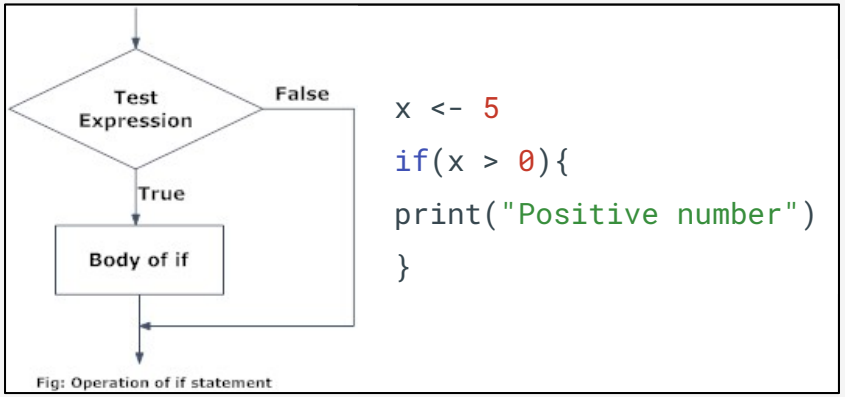


Fig: Operation of if statement

```
x <- 5
if(x > 0){
print("Positive number")
}
```

Syntax of the if-else statement

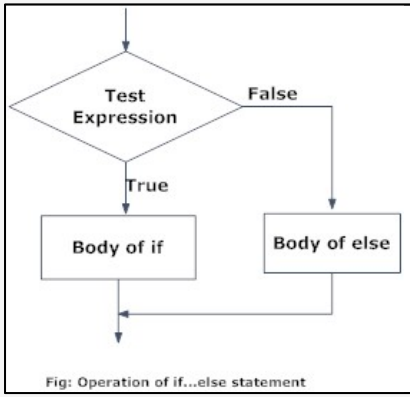


Fig: Operation of if...else statement

```
if (test_expression)
{
statement1
} else {
statement2
}
```

if-else statement

Here is a very simple example that tells us which states, if any, have a murder rate lower than 0.5 per 100,000. The else statement protects us from the case in which no state satisfies the condition.

```
min <- which.min(murders$rate)

if(murders$rate[min] < 0.5){
  print(murders$state[min])
} else{
  print("No state has murder rate that low")
}
#> [1] "Vermont"
```

Function that gives the position (row nr.) of the smallest murder rate

```
if(murder_rate[min] < 0.25){
  print(murders$state[min])
}
>
```

Nothing is printed when the expression is FALSE

```
if(murder_rate[min] < 0.25){
  print(murders$state[min])
} else{
  print("No state has a murder rate that low.")
}
#> [1] "No state has a murder rate that low."
```

Do Exercise 6

(we’ll go through it together)