# Hacettepe University
# Department of Industrial Engineering
# Undergraduate Program
# 2023-2024 Fall

# EMU 430 – Data Analytics
# Week2
# October 13, 2023

**Instructor:** Erdi Dasdemir

edasdemir@hacettepe.edu.tr
www.erdidasdemir.com

20 Ekim 2023
15:00

# Endüstri Müh. ve Veri Bilimi

Hacettepe Üniversitesi Endüstri Mühendisliği Bölümü
Veri Bilimi ve Yöneylem Araştırması Üzerine
Davetli Konuşmalar Serisi

EMÜ 430 ve 679 dersleri

Konuşmacı

**Cem Vardar**

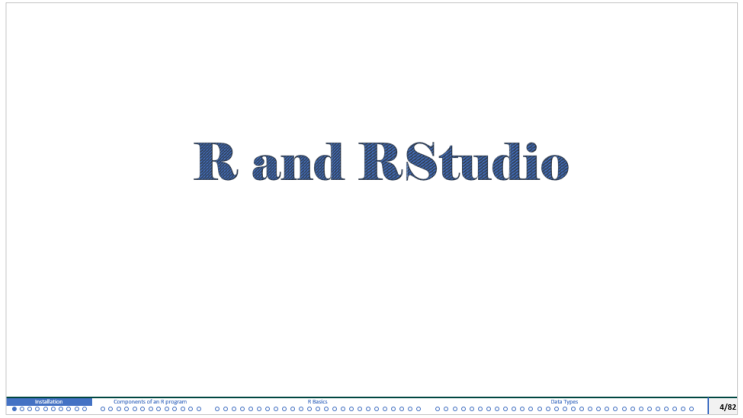
Data Scientist & Co-Founder of Decision Science Lab & Former Data Scientist at Intel, Revionics and Carvana

Moderatör

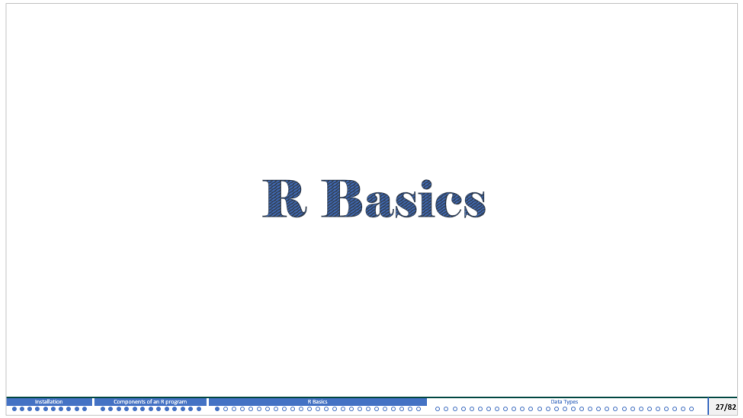
**Erdi Dasdemir**

Data Scientist & Assistant Professor of Industrial Engineering

# Outline


R and RStudio
Installation
Components of an R program
R Basics
Data Types
4/82


Components of an
R program
Installation
Components of an R program
R Basics
Data Types
14/82


R Basics
Installation
Components of an R program
R Basics
Data Types
27/82


Data Types
Installation
Components of an R program
R Basics
Data Types
52/82

# R and RStudio

# Installing R and RStudio

- R is the language itself and RStudio is the most popular integrated development environment for R.
- Installation of R is sufficient to write programs on our computer. However, RStudio provides a more enriched and user-friendly programming experience with many useful tools.

To install R:

1. Go to https://www.r-project.org.
2. Select Download → CRAN from the left menu.
3. Choose a download location close to you.
4. Choose the download option for your operating system.
5. Choose base → Install R for the first time.
6. Download the most recent R version.
7. Run downloaded file for installation.

To install RStudio:

1. Go to https://posit.co.
2. Choose Download RStudio.
3. Choose RStudio Desktop → Download RStudio.
4. Download the option suitable for your operating system.
5. Run downloaded file for installation.

- '>' prompt. This is the R's way of asking you, "What do you want me to do next?"
- We can communicate with our computer using the R language. All we need is to learn to speak the R language

```
> Hi Computer. I don't have time to learn R language. Let's do it in English this time.
Please write Hello World to the screen.
Error: unexpected symbol in "Hi Computer."
> print("Hello World")
[1] "Hello World"
```

- We can store our code within scripts for easy editing later on, debug our code line-by-line, and make our code callable from other scripts.
  - *File > New File > R Script*
  - *Ctrl+S*, *File > Save*

- It's best to use a descriptive name with lowercase letters and **no spaces**; you can use hyphens or underscores to separate words. There are different views on using underscores (_) or hyphens (-) in word separation. My suggestion is to be sure to be consistent with whatever you prefer.

- When we run a script, we are essentially telling R to read and execute the commands in the file. The extension for a script in R is ".R" just as Python has ".py".

- The advantage of working in RStudio instead of R console is that you do not have to finish the script to run it.

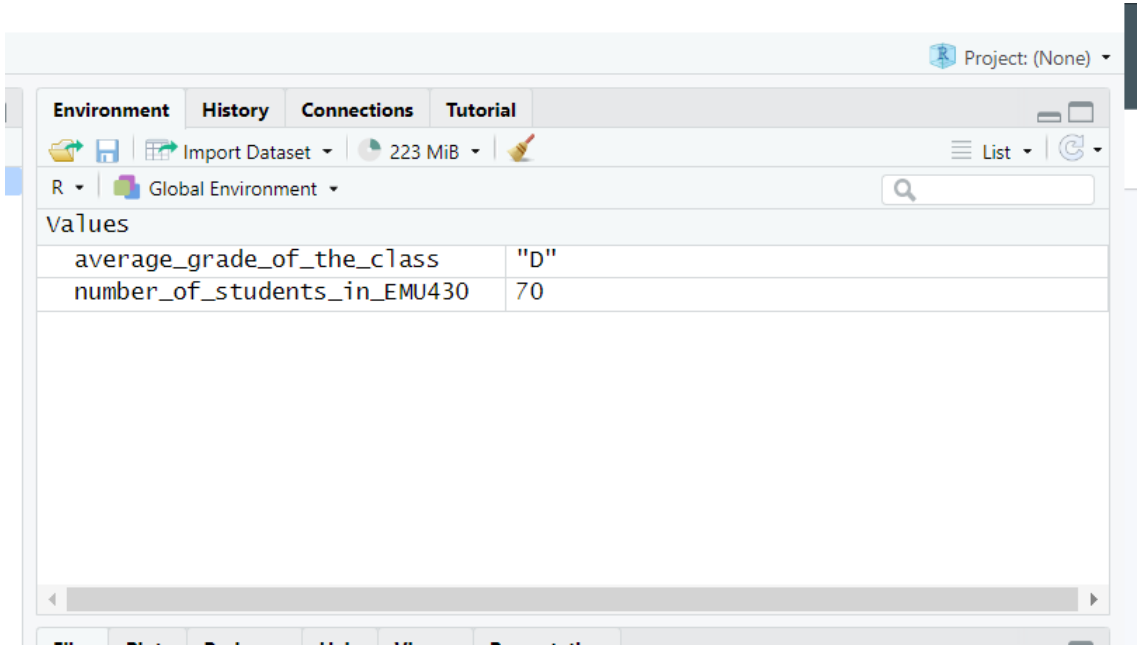
- In RStudio, you can run the part you have completed or even a single line.

- **This allows you to construct the analytical model behind the program as you can check the results of your computations during progress, and do debugging easily.**

- RStudio highlights the syntax and provides better readability. If there are syntax problems, it will indicate errors even without running the code.

- **Environment:**

➢ The top-right pane displays information about the current R environment—specifically, information that you have stored inside of variables.




➢ You will often create dozens of variables within a script, and the Environment pane helps you keep track of which values you have stored in which variables.

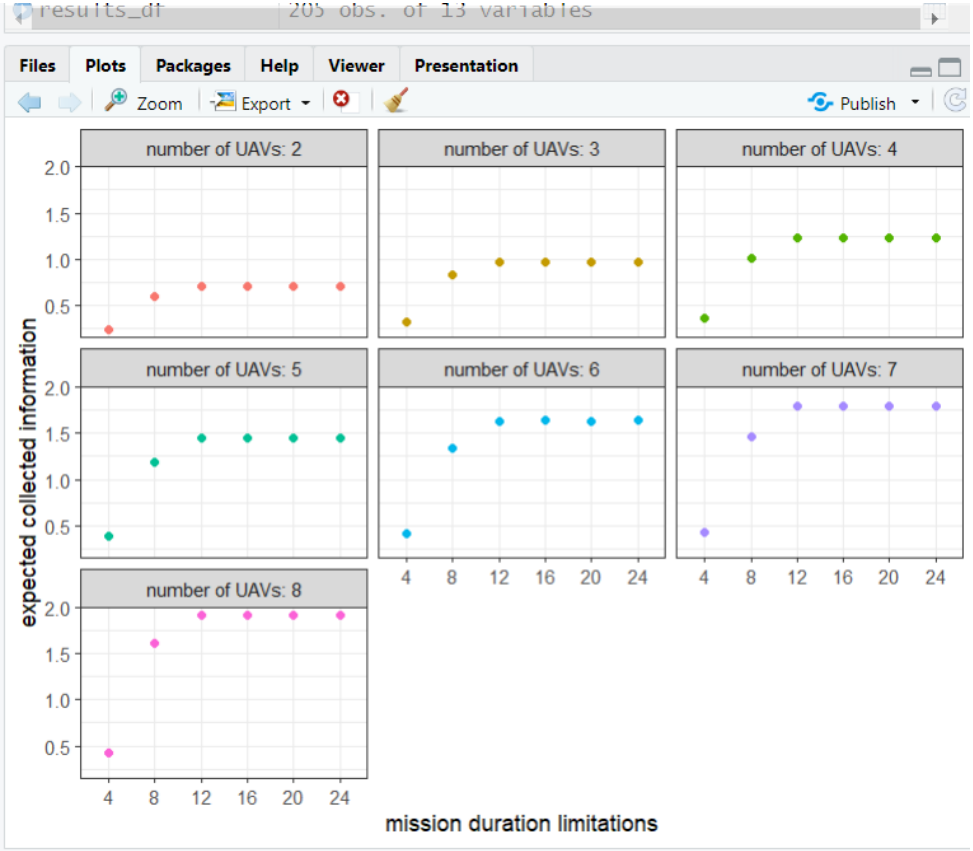
➢ This is incredibly useful for “debugging” (identifying and fixing errors)!

- **The bottom-right pane contains multiple tabs for accessing a variety of information about your program.**

➢ When you create visualizations, those plots will be rendered in this section.

➢ You can also see which packages you have loaded or look up information about files.

➢ You can access the official documentation for the R language in this pane. If you ever have a question about how something in R works, this is a good place to start!



# Writing our First Program

- It is a Computer Science tradition to begin your first program by writing "Hello World" to the screen. To do this, open a new script and save it as "first-program.R". Then, in the R console, enter the command:

```
> print("Hello World")
[1] "Hello World"
```

- We can calculate the age of the Turkish Republic by subtracting "year_established" from "year_now".

```
> year_now = 2023
> year_established = 1923
> age = year_now - year_established
> print(age)
[1] 100
```

# Components of an R program

# Components of an R Program

- Vocabulary
  - reserved words and identifiers make up the language's vocabulary
- Lines
  - we write our code line by line
- Code blocks
  - multiple lines that are related to each other form a cohesive and meaningful sequence

- What is a reserved word?
  - predetermined and built into the language, so that the R parser understands them
  - R is smart enough to recognize that these words have a specific purpose, and whenever we use them R always knows what we mean.
  - It is important to note that these words cannot be used for any other purpose, such as variable names.

Reserved words in R: https://stat.ethz.ch/R-manual/R-devel/library/base/html/Reserved.html

| | |
|---|---|
| if | TRUE |
| else | FALSE |
| repeat | NULL |
| while | Inf |
| function | NaN |
| for | NA |
| in | NA_integer_ |
| next | NA_real_ |
| break | NA_complex_ |
| | NA_character_ |

- Identify our variables, functions, and other objects by giving names to them.
- The names used for identification are not important to R, but they are meaningful to human users to assist them in constructing their program.

```
year_now <- 2023
year_established <- 1923
age <- year_now - year_established
print(age)
```

```
x <- 2023
y <- 1923
z <- y - x print(z)
```

```
asdsadsad <- 2023
xyxyxyxyx <- 1923
ghnjmkj <- asdsadsad - xyxyxyxyx
print(ghnjmkj)
```

- Use variable names that are mnemonic
- you will appreciate it when you start developing larger code blocks and debugging your code to identify the root of errors

- Most programming-language communities have agreed-upon naming conventions, which are sets of rules that govern how functions and variables should be named.
- This is not the case with R; a review of unofficial style guides and naming conventions used on CRAN reveals that a number of different methods are in use.
- Some conventions are here, and as you will see, they differ greatly.
  - Colin Gillespie's R style guide: https://csgillespie.wordpress.com/2010/11/23/r-style-guide/
  - Hadley Wickham's style guide: http://stat405.had.co.nz/r-style.html
  - Google: https://google.github.io/styleguide/Rguide.html

- Ultimately, it is up to you to decide which convention you prefer.

- I follow the Python naming convention, which differs from those commonly used for R. My reasons for doing so are twofold: first, I write code in both languages, so it is easier for me to read my code when switching between them; second, the Python naming conventions are widely accepted and well-established

- You can see the official Python naming conventions here: https://peps.python.org/pep-0008/

- whichever one you choose, s**elect informative and mnemonic names for variables**
- For example,
  - I prefer to use “**year_ie_hacettepe_established**” over “**year_ie_hac_est**”
  - “ **myemu_db_student_info_df**” is better for me than “**myemu_db_ student_info**”

- **Use <- for assignments:**
- **Do not use dots (.) in your identifiers:**
  - R allows you to use "." in your variable names, but this is not a recommended or accepted practice in other languages. If you continue working in data analytics, you'll likely eventually use Python; Python does not permit the use of "." in variable names.
  - Python is an object-oriented programming language, meaning that almost every entity is treated as an object with attributes and methods. The dot (.) notation is used to access the attributes and methods of an object.
  - For example, for `ie_hacettepe` object in Python, and you may use `ie_hacettepe.establishment_year` to access the year IE Hacettepe was established

- What is the difference between ***syntax*** and ***style***?

➢ **Syntax** describes the rules for writing the code so that a computer can interpret it.

➢ **Styles** are optional conventions that make it easier for other humans to interpret your code.

- We communicate with R line by line.
- In general, we should write our code line by line, except in obligatory cases.
- You may include multiple instructions on the same line of your script using additional punctuation, but this is neither common nor a good programming practice
- Each line that can be executed by R is also called a Statement.

```
print(1)
[1] 1
x<-2
print(x)
[1] 2
```

- A code block is connected and meaningful code pieces that are constructed by a set of lines
- I expect a code block to perform a task

```
> year_now <- 2023
> year_established <- 1923
> age <- year_now - year_established
```

This block is responsible for calculating the age of the Tur kish Republic. Finally, our code blocks are incorporated in to R scripts, and we have already discussed the scripts in detail above.

# Including Comments

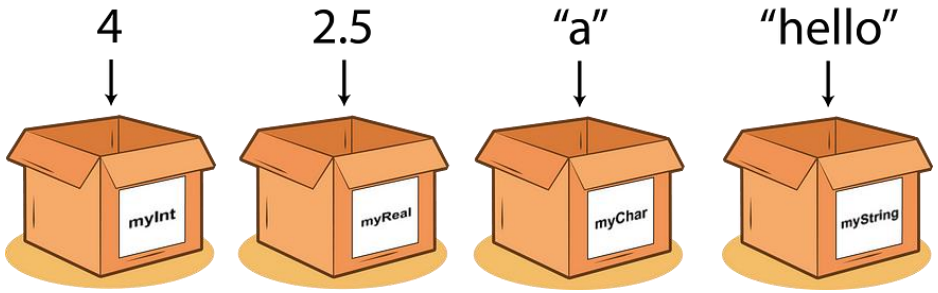
- Developers use comments to help write down the meaning and purpose of their code. This is particularly important when someone else will be looking at your work.

- In R, you mark text as a comment by putting it after the pound symbol (#). Everything from the # until the end of the line is a comment.

# R Basics

- Variables, values and types
- Functions
- Arithmetic Operations
- Relational Operations

- **Variables** are labels for information; in R, you can think of them as “boxes” or “name tags” for data. After putting data in a variable box, you can then refer to that data by the label on the box.




https://sidthakur3519.medium.com/variables-and-its-usage-7a7b32773880

- **How do you store information in a variable?**
  - Using assignment operator <-
- **How do you display the value of a variable?**
  - Using print function

- Variable names
  - can be arbitrarily long.
  - can contain both letters and numbers
  - cannot start with a number.
- It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter.
- Is this a valid name?
  - 2008_hacettepe_endustri

```
2008_hacettepe_endustri <- 2008
Error: unexpected input in "2008_"
```

- A value is one of the basic things a program works with, like a letter or a number.

- Example: `2023` is an *numeric* `"Hello World!"` is a *string.*

1. Numeric
2. Integers
3. Characters
4. Logical
5. Complex

- R is a **dynamically typed language**, which means ??
  - you do not need to explicitly state which type of information will be stored in each variable you create.
- In **statically typed languages**, you need to declare the type of variable you want to create. For example, in the Java programming language, you have to indicate the type of variable you want to create: if you want the integer 10 to be stored in the variable `my_num`, you would have to write `int my_num = 10`

## Numeric

The default computational data type in R is numeric data

You can use mathematical operators on numeric data (+, -, *, /)

```
number <- 4
class(number)
[1] “numeric"
typeof(number)
[1] "double"
```

**Double.** This is probably the most common data type in the R programming language. A variable or a series will be stored as double if the value is numeric. This means that a value such as “4” here, is stored as 4.00 with a type of **double** and a class of **numeric**.

- integer (whole-number) values are technically a different data type than numeric values because of how they are stored and manipulated by the R interpreter.
- This is something that you will rarely encounter, but it's good to know that you can specify that a number is of the integer type rather than the general numeric type.

```
my_integer <- 10L
class(my_integer)
1] "integer"
typeof(my_integer)
[1] "integer"
```

```
my_integer <- as.integer(10)
class(my_integer)
1] "integer"
typeof(my_integer)
[1] "integer"
```

- Character data stores strings of characters (e.g., letters, special characters, numbers) in a variable.
- You specify that information is character data by surrounding it with either single quotes (') or double quotes (").
- the tidyverse style guide suggests always using double quotes.

```
class("Helllo World!")
[1] "character"

class("4")
[1] "character"
```

- Logical (boolean) data types store “yes-or-no” data.
- A logical value can be one of two values: TRUE or FALSE.
- **Importantly, these are not the strings "TRUE" or "FALSE"; logical values are a different type!**
- If you prefer, you can use the shorthand T or F.
- Logical values are most commonly produced by applying a relational operator (also called a comparison operator) to some other data.

```
logi <- FALSE
class(logi)
[1] "logical"
typeof(logi)
[1] "logical"
```

```
number_guitar_strings <- 6
number_mandolin_strings <- 8

# Compare the number of strings on each instrument
number_guitar_strings > number_mandolin_strings # returns logical value FALSE
[1] FALSE
number_guitar_strings != number_mandolin_strings # returns logical value TRUE
[1] TRUE
# Equivalently, you can compare values that are not stored in variables
6 == 8 # returns logical value FALSE
[1] FALSE
# Use relational operators to compare two strings
"mandolin" > "guitar" # returns TRUE (m comes after g alphabetically)
[1] TRUE
```

- The complex data type is to store numbers with an imaginary component.

  Examples of complex values would be 1+2i, 3i, 4-5i, -12+6i, etc.

- We will not be using complex numbers in this book, as they rarely are important for data science.

```
complex_variable <- 2i
class(complex_variable)
[1] "complex" >
typeof(complex_variable)
[1] "complex"
```

- A program that we code for a data analytics task almost always use a series of functions
- A function can be either defined by us or predefined in R's defaults.
- Functions let us avoid writing the same code over and over again whenever we do the same task.
- Functions represent a way for you to add a label to a group of instructions.

**Definition:** functions represent a way for you to add a label to a group of instructions.

- We have already seen examples of function calls: **class** and **typeof**

```
number <- 4
class(number)
[1] "numeric"
typeof(number)
[1] "double"
```

- **A function "takes" arguments and "returns" a result.**
- The argument(parameter) is a value or variable that we are passing into the function as input to the function.
- Lets write "sqrt() in R" to R's help to see the details of this function

```
help("sqrt")

?sqrt

args(sqrt)
```

**Description**

sqrt(x) computes the (principal) square root of x.

**Arguments**

*x* a numeric or `complex` vector or array.

We can see that sqrt() requires only 1 argument, which is x, and x is the number or a set of numbers stored in a vector that we want to take square root of.

Lets now look at log():

```
help("log")
```

```
?log
```

```
args(log)
```

Description

log computes logarithms, by default natural logarithms, log10 computes common (i.e., base 10) logarithms, and log2 computes binary (i.e., base 2) logarithms. The general form log(x, *base*) computes logarithms with base *base*.

Arguments

x a numeric or complex vector.

base a positive or complex number: the base with respect to which logarithms are computed. Defaults to e=exp(1).

- Now, we see that log() requires **two arguments**. **x** is a must argument, and **base** is optional. If we do not define base argument, then R will use the default base defined in the function, which is *e*, e.g. natural logarithm in this case.
- For example, log(20) + log(20, base=2), respectively, refer to ???
  - **ln(20) and $log_2(20)$.**

Lets now look at log():

$log_2(20)$

- `log(x=20, base=2), log(20, base=2), log(20, 2), log(x=20,2), log(base=2, x=20)`
- All functions calls refers to the same tasks: find $log_2(20)$

- So you can either write argument names explicitly or simply write argument values in the same order they are defined in function's default (you may reach this information from the help).
  - if you have argument names expilicitly, yo do not need to follow argument order
  - if you are not using argument names, then order matters.

- I, personally, often try to explicitly write function arguments, to increase the readability of my code (unless the function is too simple and takes only 1-2 arguments, like sqrt(x))

- In this way, when I return to my code after a long time, it becomes easier for me to understand what the functions are doing. Otherwise, I had to go to their helps and understand the arguments.

- Again, this is just for human readability. For R, it does not matter whether you define the argument names expilicitly or not.

## Built-in (Base) Functions

- R provides a number of important built-in functions that we can use without needing to provide the function definition.
- The developers of R wrote a set of functions to solve common problems and included them in R for us to use

- Examples:

```
print("Hello world")
[1] "Hello world"
sqrt(25)
[1] 5
min(1, 0.75, 1.25)
[1] 0.75
nchar('Hello world')
[1] 11
```

| Function Name | Description | Example |
|---|---|---|
| `sum(a, b, ...)` | Calculates the sum of all input values | `sum(1, 5)` # *returns* `6` |
| `round(x, digits)` | Rounds the first argument to the given number of digits | `round(3.1415, 3)` # *returns* `3.142` |
| `toupper(str)` | Returns the characters in uppercase | `toupper("hi mom")` # *returns* `"HI MOM"` |
| `paste(a, b, ...)` | *Concatenates* (combines) characters into one value | `paste("hi", "mom")` # *returns* `"hi mom"` |
| `nchar(str)` | Counts the number of characters in a string (including spaces and punctuation) | `nchar("hi mom")` # *returns* `6` |
| `c(a, b, ...)` | *Concatenates* (combines) multiple items into a *vector* (see Chapter 7) | `c(1, 2)` # *returns* `1, 2` |
| `seq(a, b)` | Returns a sequence of numbers from a to b | `seq(1, 5)` # *returns* `1, 2, 3, 4, 5` |

**R Reference Card: cheat sheet summarizing built-in R functions:** https://cran.r-project.org/doc/contrib/Short-refcard.pdf

## Type Conversion Functions

- R also provides built-in functions that convert values from one type to another.
- Example:

```
user_number <- 1234
as.character(user_number)
[1] "1234"

user_character <- “1234”
as.numeric(user_character)
[1] 1234
```

To emphasize once again, we should follow the below steps when using a function:

1. Go help or google to find the arguments of the function.
2. See if a default value is assigned to an argument of a function. If a default value is assigned, then the argument is optional. If no default is there, then you have to give this argument to function to be able to run it.
3. Run the function with proper arguments.

- In addition to the defaults of R functions, there are tons of functions that are prebuilt in R packages. From time to time, we will be using R packages in this course and call their prebuilt functions.
- Example:

```
install.packages("stringr")
library("stringr")
str_count("Mississippi", "i") # 4
```

**stringr** provides a function **`str_count()`** that returns how many times a "substring" appears in a word

- We can also define our own functions.
- We will talk more about these next week.

- R has 2 sets of functions that can be used without parentheses (). These are arithmetic operators and relational operators.

  - Arithmetic operators
  - Relational operators.

# Arithmetic Operators

```
help("+")
```

## Arithmetic Operators

### Description

These unary and binary operators perform arithmetic on numeric or complex vectors (or objects which can be coerced to them).

### Usage

```
+ x
- x
x + y
x - y
x * y
x / y
x ^ y
x %% y
x %/% y
```

### Arguments

| | |
|---|---|
| `x, y` | numeric or complex vectors or objects which can be coerced to such, or other objects for which methods have been written. |

**Some Arithmetic Operations**

```
> (9/3) + (4*2) - (3^2) + sqrt(4)
 [1] 4
```

The parentheses above are just for readability. We can simply remove them and R will handle the rest.

```
> 9/3 + 4*2 - (3^2) + sqrt(4)
[1] 4
```

```
> exp(2) + log(20) + log(20, base=2) + 14%%3
[1] 16.70672
```

# Relational Operators

```
help("<")
```

## Relational Operators

### Description

Binary operators which allow the comparison of values in atomic vectors.

### Usage

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

### Arguments

| | |
|---|---|
| `x, y` | atomic vectors, symbols, calls, or other objects for which methods have been written. |

# Data Types

Data Frames

Vectors

- Numeric
- Character
- Logical

Factors

## Motivating Case Study

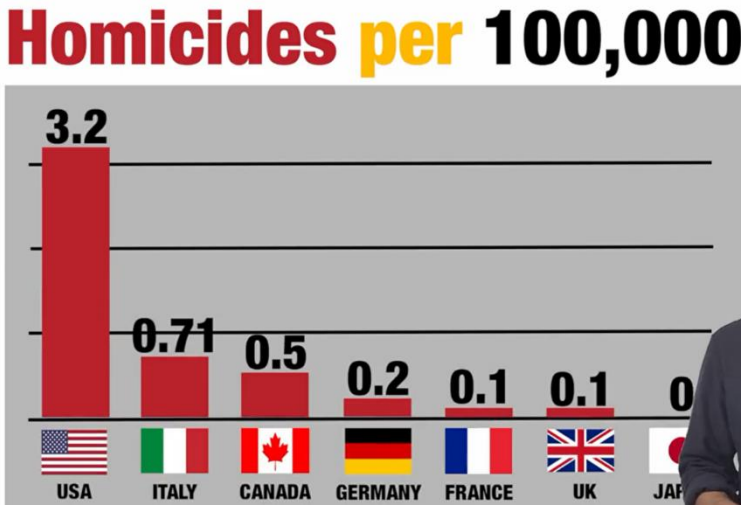
Crime rates in the US.

Our classmate Baran is offered a job in a US company. This company has many locations across all states.

It is a great job but Baran recently read news with the headline “US Gun Homicide Rate Higher Than Other Developed Countries”.

Baran is worried and considering declining the jobs but then he wants to look at the data by himself.

He investigates how safe each state is.

## Data frames

- Most common way of storing data in R
- Conceptually, they are like tables where rows represent observations and columns represent different variables.
- They are useful as we can store different data types into a single object

```
library(dslabs)
data(murders)
class(murders)
[1] "data.frame"
```

Our data is stored in the object murders, but how can we learn more about the “murder” object????

We can start with **str** function.

→Structure of an object

```
str(murders)
```

```
'data.frame':   51 obs. of  5 variables:
 $ state     : chr  "Alabama" "Alaska" "Arizona" "Arkansas" ...
 $ abb       : chr  "AL" "AK" "AZ" "AR" ...
 $ region    : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...
 $ population: num  4779736 710231 6392017 2915918 37253956 ...
 $ total     : num  135 19 232 93 1257 ...
```

We can also look at the first lines of the data frame with the **head()** function.

```
head(murders)
```

```
       state abb region population total
1    Alabama  AL  South    4779736   135
2     Alaska  AK   West     710231    19
3    Arizona  AZ   West    6392017   232
4   Arkansas  AR  South    2915918    93
5 California  CA   West   37253956  1257
6   Colorado  CO   West    5029196    65
```

→Rows are the different observations, states

→The columns represent different variables (state, abbreviation, region, population, and total)

Accessing the variables of the murders object: We use **accessor, $.**

```
murders$population
```

```
 [1]  4779736   710231  6392017  2915918 37253956  5029196  3574097   897934   601723 19687653  9920000
[12]  1360301  1567582 12830632  6483802  3046355  2853118  4339367  4533372  1328361  5773552  6547629
[23]  9883640  5303925  2967297  5988927   989415  1826341  2700551  1316470  8791894  2059179 19378102
[34]  9535483   672591 11536504  3751351  3831074 12702379  1052567  4625364   814180  6346105 25145561
[45]  2763885   625741  8001024  6724540  1852994  5686986   563626
```

But how did we know that there is a variable column named “population” ???

→str() function

→names() function

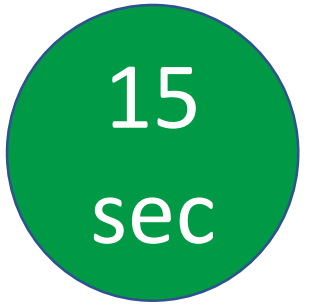
```
names(murders)
[1] "state" "abb" "region" "population" "total"
```

Important!

- The order of the entries in the list '`murders$population`' preserves the order of the rows in our data table.
- This will help us to manipulate one variable based on the results of another.
- For example, we can manipulate state names by the number of murders.

- The *movielens* dataset in the *dslabs* package includes data on a variety of movies and their rating.

```
library(dslabs)
data(movielens)
```


How many rows are in the dataset?

How many different variables are in the dataset?

What is the variable type of title ?

```
murders$population
```

```
 [1]  4779736   710231  6392017  2915918 37253956  5029196  3574097   897934   601723 19687653  9920000
[12]  1360301  1567582 12830632  6483802  3046355  2853118  4339367  4533372  1328361  5773552  6547629
[23]  9883640  5303925  2967297  5988927   989415  1826341  2700551  1316470  8791894  2059179 19378102
[34]  9535483   672591 11536504  3751351  3831074 12702379  1052567  4625364   814180  6346105 25145561
[45]  2763885   625741  8001024  6724540  1852994  5686986   563626
```

Note that the results is not a single value. It has 51 different values.

We call these types of objects ????

→ **Vectors**

```
pop <- murders$population
length(pop)
[1] 51
class(pop)
[1] numeric
```

**Then pop is a numeric vector**

We can also store characters into a vector in R → **character vectors**

```
> murders$state
 [1] "Alabama"              "Alaska"              "Arizona"             "Arkansas"            "California"
 [6] "Colorado"             "Connecticut"         "Delaware"            "District of Columbia" "Florida"
[11] "Georgia"              "Hawaii"              "Idaho"               "Illinois"            "Indiana"
[16] "Iowa"                 "Kansas"              "Kentucky"            "Louisiana"           "Maine"
[21] "Maryland"             "Massachusetts"       "Michigan"            "Minnesota"           "Mississippi"
[26] "Missouri"             "Montana"             "Nebraska"            "Nevada"              "New Hampshire"
[31] "New Jersey"           "New Mexico"          "New York"            "North Carolina"      "North Dakota"
[36] "Ohio"                 "Oklahoma"            "Oregon"              "Pennsylvania"        "Rhode Island"
[41] "South Carolina"       "South Dakota"        "Tennessee"           "Texas"               "Utah"
[46] "Vermont"              "Virginia"            "Washington"          "West Virginia"       "Wisconsin"
[51] "Wyoming"
> class(murders$state)
[1] "character"
```

We can also store logical into a vector in R → **logical vectors**

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

**Vector:**

- The most basic unit available in R to store data are vectors.
- Complex datasets can usually be broken down into components that are vectors.
- For example, in a data frame such as the murders data frame, each column is a vector.

**How do we create a vector?**

"c()" function → concatenate

```
codes <- c(380, 124, 818)
country <- c("italy", "canada", "egypt")
names(codes) <- country
print(codes)
```

```
> codes
 italy canada  egypt
   380    124    818
```

or

```
codes<- c(italy=380, canada=124, egypt=818)
print(codes)
```

```
> codes
 italy canada  egypt
   380    124    818
```

What is the general type of object `codes`? string or numeric?

Another way of creating vectors is using function sequence, `seq()`.

```
seq(1, 10)
```

```
seq(1, 10, 2)
```

Or you can simply say

```
1:10
```

Subsetting: Accessing elements of a vector → use []

```
> codes
 italy canada  egypt
   380    124    818
```

codes[2] → ?

```
canada
   124
```

codes[c(1,3)] → ?

```
italy egypt
  380   818
```

codes[1:2]→ ?

```
italy canada
  380   124
```

Accessing the entries with names

```
codes["canada"]
canada
   124
```

```
codes[c("egypt", "italy")]
egypt italy
  818   380
```

- **Coercion:** In general, coercion is an attempt by R to be flexible with data types.

- When an entry does not match the expected, R tries to guess what we meant before throwing in an error. But this can also lead to confusion.

- Failing to understand coercion can drive programmers crazy when attempting to code in R, since it behaves quite differently from most other languages.

**Examples**

```
x <- c(1, "Canada", 3)

x

[1] "1" "Canada" "3"

class(x)

[1] "character
```

Numbers are converted to character!!

We say “R Coerced the data into a character string.”

**R also has built-in coercion functions.**

```
x <- 1:5

y <- as.character(x)

y

[1] "1" "2" "3" "4" "5"

as.numeric(y)

[1] 1 2 3 4 5
```

These coercion functions are quite useful in practice because many datasets that include numbers, include them in a form that makes them appear to be character strings!!!

- Missing data is very common in practice.
- In R, we have a special value for missing data: NA
- We can get NAs from coercion.
- For example, when R fails to coerce something, we will get NA.
- Example:

```
x<- c("1", "b", "3")
as.numeric(x)
[1] 1 NA 3
Warning message: NAs introduced by coercion
```

Rank states from least to most dangerous:

`sort()` → sorts a vector in increasing order

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

→ What do you see here? Is this an enough information for you?

We only see totals, we don't see state names

Rank states from least to most dangerous:

`order()` → it takes a vector and returns the indices that sorts that vector

Example:

```
x <- c(31, 4, 15, 92, 65)
x
[1] 31 4 15 92 65
sort(x)
[1] 4 15 31 65 92
index <- order(x)
index
[1] 2 3 1 5 4
x[index]
[1] 4 15 31 65 92
```

Now return back to murders data set:

```
> index <- order(murders$total)
> index
 [1] 46 35 30 51 12 42 20 13 27 40  2 16 45 49 28 38  8 24 17  6 32 29  4 48  7
50  9 37 18 22 25
[32]  1 15 41 43  3 31 47 34 21 36 26 19 14 11 23 39 33 10 44  5
> murders$abb[index]
 [1] "VT" "ND" "NH" "WY" "HI" "SD" "ME" "ID" "MT" "RI" "AK" "IA" "UT" "WV" "NE"
"OR" "DE" "MN"
[19] "KS" "CO" "NM" "NV" "AR" "WA" "CT" "WI" "DC" "OK" "KY" "MA" "MS" "AL" "IN"
"SC" "TN" "AZ"
[37] "NJ" "VA" "NC" "MD" "OH" "MO" "LA" "IL" "GA" "MI" "PA" "NY" "FL" "TX" "CA"
```

Vermont has the lowest and California has the highest.

If we only want to see the state with the maximum murder number:

## max() and min()

```
max(murders$total)
[1] 1257
which.max(murders$total)
[1] 5
murders$state[which.max(murders$total)]
[1] "California"
```

```
min(murders$total)
[1] 2
which.min(murders$total)
[1] 46
murders$state[which.min(murders$total)]
[1] "Vermont"
```

## rank()

```
x <- c(31, 4, 15, 92, 65)
x
[1] 31 4 15 92 65
rank(x)
[1] 3 1 2 5 4
```

To summarize,

Indices starting with the index of the smallest element

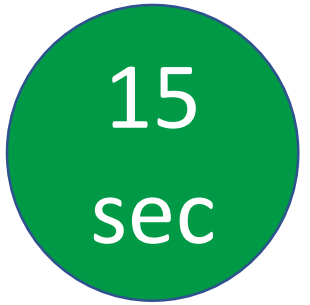
| original | sort | order | rank |
|---|---|---|---|
| 31 | 4 | 2 | 3 |
| 4 | 15 | 3 | 1 |
| 15 | 31 | 1 | 2 |
| 92 | 65 | 5 | 5 |
| 65 | 92 | 4 | 4 |

So far, we have discovered that California has the most murders of any state.

Does this mean it is the most dangerous state????

- what if it has the highest population?


Find the state with the maximum population, use a single line code:




```
murders$state[which.max(murders$population)]
[1] "California"
```

```
max(murders$population)
[1] 37253956
```

- 37,253,956 → unfair to compare California to other states

- It is better to look at what?
  - murder rates per capita

R has powerful vector arithmetic capabilities:

→ arithmetic operations on vectors occur element-wise.

Example:

```
heights <- c(69, 62, 66, 70, 70, 73, 67, 73, 67, 70)  # inches

# convert to centimeters
heights * 2.54
[1] 175.26 157.48 167.64 177.80 177.80 185.42 170.18 185.42 170.18 177.80
```

Assume that average height is 69 inches:

```
heights - 69
[1] 0 -7 -3 1 1 4 -2 4 -2 1
```

```
murder_rate <- (murders$total/murders$population)*100000

murders$state[order(murder_rate, decreasing=TRUE)]
```

```
 [1] "District of Columbia" "Louisiana"      "Missouri"       "Maryland"
 [5] "South Carolina"       "Delaware"       "Michigan"       "Mississippi"
 [9] "Georgia"              "Arizona"        "Pennsylvania"   "Tennessee"
[13] "Florida"              "California"     "New Mexico"     "Texas"
[17] "Arkansas"             "Virginia"       "Nevada"         "North Carolina"
[21] "Oklahoma"             "Illinois"       "Alabama"        "New Jersey"
[25] "Connecticut"          "Ohio"           "Alaska"         "Kentucky"
[29] "New York"             "Kansas"         "Indiana"        "Massachusetts"
[33] "Nebraska"             "Wisconsin"      "Rhode Island"   "West Virginia"
[37] "Washington"           "Colorado"       "Montana"        "Minnesota"
[41] "South Dakota"         "Oregon"         "Wyoming"        "Maine"
[45] "Utah"                 "Idaho"          "Iowa"           "North Dakota"
[49] "Hawaii"               "New Hampshire"  "Vermont"
```

- In the murders data set, we have a column called regions → which state in which region

- Normally, we can think that this would be a character but if we look at class

```
class(murders$region)
[1] factor
```

- **Factors** are useful for storing **"categorical data".**

- **Regions are categoric, there are 4 regions.**

```
levels(murders$region)
[1] "Northeast" "South" "North Central" "West"
```

- Why do we use factors? Can't we just use character type?
  - **Saving categorical data this way is more memory efficient.**

Note: I recommend avoiding factors as much as possible as they can be easily confused with character.


Any
Questions