# Section 1 Overview

Section 1 introduces you to R Basics, Functions, and Datatypes.

In Section 1, you will learn to:

* + Appreciate the rationale for data analysis using R.
  + Define objects and perform basic arithmetic and logical operations.
  + Use pre-defined functions to perform operations on objects.
  + Distinguish between various data types.

There are **two** DataCamp assessments and **one** final assessment for you to practice your coding skills.

We encourage you to use R to interactively test out your answers and further your learning.

# Motivation

**Key Points**

* In the first few videos we will go over logistics and introduce general concepts.
* Rather than cover every R skill you need, we will provide the building blocks so you can grow your knowledge as you go through the series.
* Because we better retain knowledge by learning through solving a specific problem, we will use a motivating case study - crime in the United States - to answer specific questions.

# Installing R

### Why Install R and RStudio?

To complete this course, you should install R locally on your computer. We also highly recommend installing RStudio, an integrated development environment (IDE), to edit and test your code.

In order to complete some assignments in the course, you will need your own copy of R. You may also find it helpful to follow along with the course videos in R or RStudio.

Both R and RStudio can be freely downloaded and installed.

**Key points**

* You need to install R before using RStudio, which is an interactive desktop environment.
* Select base subdirectory in CRAN and click download.
* Select all default choices in the installation process.
* We recommend selecting English for language to help you better follow the course.
* You can try using the R console, but for productivity purposes, we can switch to RStudio.

**Download R from CRAN**

You can find the latest version of R for your operating system at the [CRAN website](https://cran.r-project.org/).

# Installing RStudio

**Key points**

* You can download the latest version of RStudio at the [RStudio website External link](https://www.rstudio.com/products/rstudio/download/).
* The free desktop version is more than enough for this course.
* Make sure to choose the version for your own operating system.
* Choose "Yes" for all defaults in the installation process.

# Using RStudio for the First Time

**Key points**

* The free desktop version of RStudio can be launched like other applications on your computer.
* When you start RStudio for the first time, you will see three panes. The left pane shows you the R console. On the right, the top pane includes three tabs, while the bottom pane shows you five tabs, file, plots, packages, help, and viewer.
* You can download a cheat sheet of the most common RStudio commands directly from RStudio by going to "Help -> Cheat Sheets -> RStudio IDE Cheat Sheet."

# Getting Started Using R

**Key Points**

* R was developed by statisticians and data analysts as an interactive environment for data analysis.
* Some of the advantages of R are that (1) it is free and open source, (2) it has the capability to save scripts, (3) there are numerous resources for learning, and (4) it is easy for developers to share software implementation.
* Expressions are evaluated in the R console when you type the expression into the console and hit Return.
* A great advantage of R over point and click analysis software is that you can save your work as scripts.
* “Base R” is what you get after you first install R. Additional components are available via packages.

**Code: using install.packages and library()**

# installing the dslabs package

install.packages("dslabs")

# loading the dslabs package into the R session

library(dslabs)

**Some additional notes**

In RStudio, you can upload additional functions and datasets in addition to the base R functions and datasets that come with R automatically. A common way to do this is by installing packages, which often contain extra functions and datasets. For this course, there are a few packages you will need to install. You only need to install each individual package once, but after you install a package, there are other steps you have to do whenever you want to use something from that package.

To install a package, you use the code install.packages("package\_name", dependencies = TRUE).

To load a package, you use the code library(package\_name).

If you also want to use a dataset from a package you have loaded, then you use the code data(dataset\_name). To see the dataset, you can take the additional step of View(dataset\_name).

# Installing Packages

### Note

We recommend installing packages through RStudio, rather than through R, and the code provided works in both R and RStudio. Once a package has been installed, it is technically added onto R (even if you use RStudio to install it), which is why packages must be re-installed when R is updated. However, since we use R through RStudio, any packages that are installed can be used in both R and RStudio, regardless of which one was used to install the packages.

**Key points**

* The base version of R is quite minimal, but you can supplement its functions by installing additional packages.
* We will be using **tidyverse** and **dslabs** packages for this course.
* Install packages from R console: install.packages("pkg\_name")
* Install packages from RStudio interface: Tools > Install Packages (allows autocomplete)
* Once installed, we can use library(pkg\_name) to load a package each time we want to use it

**Additional Notes**

* If you try to load a package with library(blahblah) and get a message like *Error in library(blahblah) : there is no package called 'blahblah'*, it means you need to install that package first with install.packages().
* On the DataCamp interface we use for some problems in the course, you cannot install additional packages. The problems have been set up with the packages you need to solve them.
* You can add the option dependencies = TRUE, which tells R to install the other things that are necessary for the package or packages to run smoothly. Otherwise, you may need to install additional packages to unlock the full functionality of a package.
* Throughout the course materials and textbook, package names are in **bold**.

**Code**

install.packages("dslabs") # to install a single package

install.packages(c("tidyverse", "dslabs")） # to install two packages at the same time

installed.packages() # to see the list of all installed packages

# Running Commands While Editing Scripts

**Note**

The code in this video contains a pipe (i.e. %>%) and the ggplot() function. Pipes are from the **tidyverse** package and are covered in a later section of the course, but ggplot() is not covered in R Basics. You do not need to be able to be able to use pipes or ggplot() at this point.

**Textbook link**

This video corresponds to the textbook section on [RStudio External link](https://rafalab.github.io/dsbook/getting-started.html#rstudio).

**Key points**

* RStudio has many useful features as an R editor, including the ability to test code easily as we write scripts and several autocomplete features.
* Keyboard shortcuts:
  + Save a script: Ctrl+S on Windows and Command+S on Mac
  + Run an entire script:  Ctrl+Shift+Enter on Windows Command+Shift+Return on Mac, or click "Source" on the editor pane
  + Run a single line of script: Ctrl+Enter on Windows and Command+Return on Mac while the cursor is pointing to that line, or select the chunk and click "run"
  + Open a new script: Ctrl+Shift+N on Windows and Command+Shift+N on Mac

**Code**

library(tidyverse)

library(dslabs)

data(murders)

murders %>%

ggplot(aes(population, total, label=abb, color=region)) +

geom\_label()

# R Basics

**Key Points**

* To define a variable, we may use the assignment symbol, <-.
* There are two ways to see the value stored in a variable: (1) type the variable name into the console and hit Return, or (2) use the print() function by typing print(variable\_name) and hitting Return.
* Objects are things that are stored in named containers in R.  They can be variables, functions, etc.
* The ls() function shows the names of the objects saved in your workspace.

**Code: solving the equation**

**X2 + x + 1 = 0**

# assigning values to variables

a <-1

b <-1

c <--1

# solving the quadratic equation

(-b + sqrt(b^2 - 4\*a\*c))/(2\*a)

(-b - sqrt(b^2 - 4\*a\*c))/(2\*a)

# Functions

**Key points**

* In general, to evaluate a function we need to use parentheses. If we type a function without parenthesis, R shows us the code for the function. Most functions also require an argument, that is, something to be written inside the parenthesis.
* To access help files, we may use the help function, help(function\_name), or write the question mark followed by the function name, ?function\_name.
* The help file shows you the arguments the function is expecting, some of which are required and some are optional. If an argument is optional, a default value is assigned with the equal sign. The args() function also shows the arguments a function needs.
* To specify arguments, we use the equals sign. If no argument name is used, R assumes you’re entering arguments in the order shown in the help file.
* Creating and saving a script makes code much easier to execute.
* To make your code more readable, use intuitive variable names and include comments (using the “#” symbol) to remind yourself why you wrote a particular line of code.

# Data Types

### Note

The code data("dataset\_name") and data(dataset\_name) do the same thing. The code will work regardless of whether the quotes are present. It is a bit faster to leave out the quotes (as we do in the Code at the bottom of this page), so that is usually what we recommend, but it is your choice.

**Key Points**

* The function class() helps us determine the type of an object.
* Data frames can be thought of as tables with rows representing observations and columns representing different variables.
* To access data from columns of a data frame, we use the dollar sign symbol, $, which is called the accessor.
* A vector is an object consisting of several entries and can be a numeric vector, a character vector, or a logical vector.
* We use quotes to distinguish between variable names and character strings.
* Factors are useful for storing categorical data, and are more memory efficient than storing characters.

### Code

# loading the dslabs package and the murders dataset

library(dslabs)

data(murders)

# determining that the murders dataset is of the "data frame" class

class(murders)

# finding out more about the structure of the object

str(murders)

# showing the first 6 lines of the dataset

head(murders)

# using the accessor operator to obtain the population column

murders$population

# displaying the variable names in the murders dataset

names(murders)

# determining how many entries are in a vector

pop <- murders$population

length(pop)

# vectors can be of class numeric and character

class(pop)

class(murders$state)

# logical vectors are either TRUE or FALSE

z <- 3 == 2

z

class(z)

# factors are another type of class

class(murders$region)

# obtaining the levels of a factor

levels(murders$region)

# Section 2 Overview

**Section 2 introduces you to vectors and functions such as sorting.**

**In Section 2.1, you will:**

* + Create numeric and character vectors.
  + Name the columns of a vector.
  + Generate numeric sequences.
  + Access specific elements or parts of a vector.
  + Coerce data into different data types as needed.

In Section 2.2, you will:

* + Sort vectors in ascending and descending order.
  + Extract the indices of the sorted elements from the original vector.
  + Find the maximum and minimum elements, as well as their indices, in a vector.
  + Rank the elements of a vector in increasing order.

In Section 2.3, you will:

* + Perform arithmetic between a vector and a single number.
  + Perform arithmetic between two vectors of the same length.

# Vectors

**Key Points**

* The function c(), which stands for concatenate, is useful for creating vectors.
* Another useful function for creating vectors is the seq() function, which generates sequences.
* Subsetting lets us access specific parts of a vector by using square brackets to access elements of a vector.

**Code**

# We may create vectors of class numeric or character with the concatenate function

codes <- c(380, 124, 818)

country <- c("italy", "canada", "egypt")

# We can also name the elements of a numeric vector

# Note that the two lines of code below have the same result

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

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

# We can also name the elements of a numeric vector using the names() function

codes <- c(380, 124, 818)

country <- c("italy","canada","egypt")

names(codes) <- country

# Using square brackets is useful for subsetting to access specific elements of a vector

codes[2]

codes[c(1,3)]

codes[1:2]

# If the entries of a vector are named, they may be accessed by referring to their name

codes["canada"]

codes[c("egypt","italy")]

# Vector Coercion

**Key Points**

* In general, *coercion* is an attempt by R to be flexible with data types by guessing what was meant when an entry does not match the expected. For example, when defining x as

x <- c(1, "canada", 3)

R *coerced* the data into characters. It guessed that because you put a character string in the vector, you meant the 1 and 3 to actually be character strings, "1" and "3".

* The function as.character() turns numbers into characters.
* The function as.numeric() turns characters into numbers.
* In R, missing data is assigned the value NA.

# Sorting

**Key Points**

* The function sort() sorts a vector in increasing order.
* The function order() produces the indices needed to obtain the sorted vector, e.g. a result of  2 3 1 5 4 means the sorted vector will be produced by listing the 2nd, 3rd, 1st, 5th, and then 4th item of the original vector.
* The function rank() gives us the ranks of the items in the original vector.
* The function max() returns the largest value, while which.max() returns the index of the largest value. The functions min() and which.min() work similarly for minimum values.

### Code

library(dslabs)

data(murders)

sort(murders$total)

x <- c(31, 4, 15, 92, 65)

x

sort(x) # puts elements in order

index <- order(x) # returns index that will put x in order

x[index] # rearranging by this index puts elements in order

order(x)

murders$state[1:10]

murders$abb[1:10]

index <- order(murders$total)

murders$abb[index] # order abbreviations by total murders

max(murders$total) # highest number of total murders

i\_max <- which.max(murders$total) # index with highest number of murders

murders$state[i\_max] # state name with highest number of total murders

x <- c(31, 4, 15, 92, 65)

x

rank(x) # returns ranks (smallest to largest)

# Vector Arithmetic

**Key Points**

* In R, arithmetic operations on vectors occur element-wise.

### Code

# The name of the state with the maximum population is found by doing the following

murders$state[which.max(murders$population)]

# how to obtain the murder rate

murder\_rate <- murders$total / murders$population \* 100000

# ordering the states by murder rate, in decreasing order

murders$state[order(murder\_rate, decreasing=TRUE)]

### Section 3 Overview

Section 3 introduces the R commands and techniques that help you wrangle, analyze, and visualize data.

#### In Section 3.1, you will:

* + Subset a vector based on properties of another vector.
  + Use multiple logical operators to index vectors.
  + Extract the indices of vector elements satisfying one or more logical conditions.
  + Extract the indices of vector elements matching with another vector.
  + Determine which elements in one vector are present in another vector.

#### In Section 3.2, you will:

* + Wrangle data tables using functions in the **dplyr** package.
  + Modify a data table by adding or changing columns.
  + Subset rows in a data table.
  + Subset columns in a data table.
  + Perform a series of operations using the pipe operator.
  + Create data frames.

#### In Section 3.3, you will:

* + Plot data in scatter plots, box plots, and histograms.

#### In Section 3.4, you will:

* + Use summarize() to facilitate summarizing data in **dplyr**.
  + Learn about the dot placeholder.
  + Learn how to group and then summarize in **dplyr**.
  + Learn how to sort data tables in **dplyr**.

#### In Section 3.5, you will:

* + Learn how to subset and summarize data using **data.table**.
  + Learn how to sort data frames using **data.table**.

# Indexing

**Key Points**

* We can use logicals to index vectors.
* Using the function sum()on a logical vector returns the number of entries that are true.
* The logical operator “&” makes two logicals true only when they are both true.

### Code

# defining murder rate as before

murder\_rate <- murders$total / murders$population \* 100000

# creating a logical vector that specifies if the murder rate in that state is less than or equal to 0.71

index <- murder\_rate <= 0.71

# determining which states have murder rates less than or equal to 0.71

murders$state[index]

# calculating how many states have a murder rate less than or equal to 0.71

sum(index)

# creating the two logical vectors representing our conditions

west <- murders$region == "West"

safe <- murder\_rate <= 1

# defining an index and identifying states with both conditions true

index <- safe & west

murders$state[index]

# Indexing Functions

**Key Points**

* The function which() gives us the entries of a logical vector that are true.
* The function match() looks for entries in a vector and returns the index needed to access them.
* We use the function %in% if we want to know whether or not each element of a first vector is in a second vector.

### Code

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

which(x) # returns indices that are TRUE

# to determine the murder rate in Massachusetts we may do the following

index <- which(murders$state == "Massachusetts")

index

murder\_rate[index]

# to obtain the indices and subsequent murder rates of New York, Florida, Texas, we do:

index <- match(c("New York", "Florida", "Texas"), murders$state)

index

murders$state[index]

murder\_rate[index]

x <- c("a", "b", "c", "d", "e")

y <- c("a", "d", "f")

y %in% x

# to see if Boston, Dakota, and Washington are states

c("Boston", "Dakota", "Washington") %in% murders$state

# Basic Data Wrangling

**Key Points**

* To change a data table by adding a new column, or changing an existing one, we use the mutate() function.
* To filter the data by subsetting rows, we use the function filter().
* To subset the data by selecting specific columns, we use the select() function.
* We can perform a series of operations by sending the results of one function to another function using the pipe operator, %>%.

### Code

# installing and loading the dplyr package

install.packages("dplyr")

library(dplyr)

# adding a column with mutate

library(dslabs)

data("murders")

murders <- mutate(murders, rate = total / population \* 100000)

# subsetting with filter

filter(murders, rate <= 0.71)

# selecting columns with select

new\_table <- select(murders, state, region, rate)

# using the pipe

murders %>% select(state, region, rate) %>% filter(rate <= 0.71)

# Creating Data Frames

**Key Points**

* We can use the data.frame() function to create data frames.
* Formerly, the data.frame() function turned characters into factors by default. To avoid this, we could utilize the stringsAsFactors argument and set it equal to false. As of R 4.0, it is no longer necessary to include the stringsAsFactors argument, because R no longer turns characters into factors by default.

### Code

# creating a data frame with stringAsFactors = FALSE

grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),

exam\_1 = c(95, 80, 90, 85),

exam\_2 = c(90, 85, 85, 90),

stringsAsFactors = FALSE)

# Basic Plots

**Key Points**

* We can create a simple scatterplot using the function plot().
* Histograms are graphical summaries that give you a general overview of the types of values you have.  In R, they can be produced using the hist() function.
* Boxplots provide a more compact summary of a distribution than a histogram and are more useful for comparing distributions. They can be produced using the boxplot() function.

### Code

library(dplyr)

library(dslabs)

data("murders")

# a simple scatterplot of total murders versus population

x <- murders$population /10^6

y <- murders$total

plot(x, y)

# a histogram of murder rates

murders <- mutate(murders, rate = total / population \* 100000)

hist(murders$rate)

# boxplots of murder rates by region

boxplot(rate~region, data = murders)

# The summarize function

**Key Points**

* Summarizing data is an important part of data analysis.
* Some summary statistics are the **mean**, **median**, and **standard deviation**.
* The summarize() function from **dplyr** provides an easy way to compute summary statistics.

### Code

library(tidyverse)

library(dplyr)

library(dslabs)

data(murders)

murders <- mutate(murders, rate = total / population \* 10^5)

# minimum, median, and maximum murder rate for the states in the West region

s <- murders %>%

filter(region == "West") %>%

summarize(minimum = min(rate),

median = median(rate),

maximum = max(rate))

s

# accessing the components with the accessor $

s$median

s$maximum

# average rate unadjusted by population size

mean(murders$rate)

# average rate adjusted by population size

us\_murder\_rate <- murders %>%

summarize(rate = sum(total) / sum(population) \* 10^5)

us\_murder\_rate

# Summarizing with more than one value

**Key Points**

* The quantile() function can be used to return the min, median, and max in a single line of code.

### Code

library(tidyverse)

library(dplyr)

library(dslabs)

data(murders)

murders <- mutate(murders, rate = total / population \* 10^5)

# minimum, median, and maximum murder rate for the states in the West region using quantile

# note that this returns a vector

murders %>%

filter(region == "West") %>%

summarize(range = quantile(rate, c(0, 0.5, 1)))

# returning minimum, median, and maximum as a data frame

my\_quantile <- function(x){

r <- quantile(x, c(0, 0.5, 1))

data.frame(minimum = r[1], median = r[2], maximum = r[3])

}

murders %>%

filter(region == "West") %>%

summarize(my\_quantile(rate))

# Pull to access columns

**Key Points**

* The pull() function can be used to access values stored in data when using pipes: when a data object is piped that object and its columns can be accessed using the pull() function.

### Code

library(tidyverse)

library(dplyr)

library(dslabs)

data(murders)

murders <- mutate(murders, rate = total / population \* 10^5)

# average rate adjusted by population size

us\_murder\_rate <- murders %>%

summarize(rate = sum(total) / sum(population) \* 10^5)

us\_murder\_rate

# us\_murder\_rate is stored as a data frame

class(us\_murder\_rate)

# the pull function can return it as a numeric value

us\_murder\_rate %>% pull(rate)

# using pull to save the number directly

us\_murder\_rate <- murders %>%

summarize(rate = sum(total) / sum(population) \* 10^5) %>%

pull(rate)

us\_murder\_rate

# us\_murder\_rate is now stored as a number

class(us\_murder\_rate)

# The dot placeholder

**Key Points**

* The dot (.) can be thought of as a placeholder for the data being passed through the pipe.

### Code

library(tidyverse)

library(dplyr)

library(dslabs)

data(murders)

murders <- mutate(murders, rate = total / population \* 10^5)

# average rate adjusted by population size

us\_murder\_rate <- murders %>%

summarize(rate = sum(total) / sum(population) \* 10^5)

us\_murder\_rate

# using the dot to access the rate

us\_murder\_rate <- murders %>%

summarize(rate = sum(total) / sum(population) \* 10^5) %>%

.$rate

us\_murder\_rate

class(us\_murder\_rate)

# Group then summarize

**Key Points**

* Splitting data into groups and then computing summaries for each group is a common operation in data exploration.
* We can use the **dplyr** group\_by() function to create a special grouped data frame to facilitate such summaries.

### Code

library(tidyverse)

library(dplyr)

library(dslabs)

data(murders)

murders <- mutate(murders, rate = total / population \* 10^5)

# group by region

murders %>% group\_by(region)

# summarize after grouping

murders %>%

group\_by(region) %>%

summarize(median = median(rate))

# Sorting data tables

**Key Points**

* To order an entire table, we can use the **dplyr** function arrange().
* We can also use nested sorting to order by additional columns.
* The function head() returns on the first few lines of a table.
* The function top\_n() returns the top n rows of a table.

### Code

library(tidyverse)

library(dplyr)

library(dslabs)

data(murders)

murders <- mutate(murders, rate = total / population \* 10^5)

# order the states by population size

murders %>% arrange(population) %>% head()

# order the states by murder rate - the default is ascending order

murders %>% arrange(rate) %>% head()

# order the states by murder rate in descending order

murders %>% arrange(desc(rate)) %>% head()

# order the states by region and then by murder rate within region

murders %>% arrange(region, rate) %>% head()

# return the top 10 states by murder rate

murders %>% top\_n(10, rate)

# return the top 10 states ranked by murder rate, sorted by murder rate

murders %>% arrange(desc(rate)) %>% top\_n(10)

# Introduction to data.table

**Key Points**

* In this course, we often use tidyverse packages to illustrate because these packages tend to have code that is very readable for beginners.
* There are other approaches to wrangling and analyzing data in R that are faster and better at handling large objects, such as the **data.table** package.
* Selecting in **data.table** uses notation similar to that used with matrices.
* To add a column in **data.table**, you can use the := function.
* Because the **data.table** package is designed to avoid wasting memory, when you make a copy of a table, it does not create a new object. The := function changes by reference. If you want to make an actual copy, you need to use the copy() function.
* Side note: the R language has a new, built-in pipe operator as of version 4.1: |>. This works similarly to the pipe %>% you are already familiar with.

### Code

# install the data.table package before you use it!

install.packages("data.table")

# load data.table package

library(data.table)

# load other packages and datasets

library(tidyverse)

library(dplyr)

library(dslabs)

data(murders)

# convert the data frame into a data.table object

murders <- setDT(murders)

# selecting in dplyr

select(murders, state, region)

# selecting in data.table - 2 methods

murders[, c("state", "region")] |> head()

murders[, .(state, region)] |> head()

# adding or changing a column in dplyr

murders <- mutate(murders, rate = total / population \* 10^5)

# adding or changing a column in data.table

murders[, rate := total / population \* 100000]

head(murders)

murders[, ":="(rate = total / population \* 100000, rank = rank(population))]

# y is referring to x and := changes by reference

x <- data.table(a = 1)

y <- x

x[,a := 2]

y

y[,a := 1]

x

# use copy to make an actual copy

x <- data.table(a = 1)

y <- copy(x)

x[,a := 2]

y

# Subsetting with data.table

**Key Points**

* Subsetting in **data.table** uses notation similar to that used with matrices.

### Code

# load packages and prepare the data

library(tidyverse)

library(dplyr)

library(dslabs)

data(murders)

library(data.table)

murders <- setDT(murders)

murders <- mutate(murders, rate = total / population \* 10^5)

murders[, rate := total / population \* 100000]

# subsetting in dplyr

filter(murders, rate <= 0.7)

# subsetting in data.table

murders[rate <= 0.7]

# combining filter and select in data.table

murders[rate <= 0.7, .(state, rate)]

# combining filter and select in dplyr

murders %>% filter(rate <= 0.7) %>% select(state, rate)

# Summarizing with data.table

**Key Points**

* In **data.table** we can call functions inside .() and they will be applied to rows.
* The group\_by followed by summarize in **dplyr** is performed in one line in **data.table** using the by argument.

### Code

# load packages and prepare the data - heights dataset

library(tidyverse)

library(dplyr)

library(dslabs)

data(heights)

heights <- setDT(heights)

# summarizing in dplyr

s <- heights %>%

summarize(average = mean(height), standard\_deviation = sd(height))

# summarizing in data.table

s <- heights[, .(average = mean(height), standard\_deviation = sd(height))]

# subsetting and then summarizing in dplyr

s <- heights %>%

filter(sex == "Female") %>%

summarize(average = mean(height), standard\_deviation = sd(height))

# subsetting and then summarizing in data.table

s <- heights[sex == "Female", .(average = mean(height), standard\_deviation = sd(height))]

# previously defined function

median\_min\_max <- function(x){

qs <- quantile(x, c(0.5, 0, 1))

data.frame(median = qs[1], minimum = qs[2], maximum = qs[3])

}

# multiple summaries in data.table

heights[, .(median\_min\_max(height))]

# grouping then summarizing in data.table

heights[, .(average = mean(height), standard\_deviation = sd(height)), by = sex]

# Sorting data frames

**Key Points**

* To order rows in a data frame using **data.table**, we can use the same approach we used for filtering.
* The default sort is an ascending order, but we can also sort tables in descending order.
* We can also perform nested sorting by including multiple variables in the desired sort order.

### Code

# load packages and datasets and prepare the data

library(tidyverse)

library(dplyr)

library(data.table)

library(dslabs)

data(murders)

murders <- setDT(murders)

murders[, rate := total / population \* 100000]

# order by population

murders[order(population)] |> head()

# order by population in descending order

murders[order(population, decreasing = TRUE)]

# order by region and then murder rate

murders[order(region, rate)]

[Programming Basics](https://learning.edx.org/course/course-v1:HarvardX+PH125.1x+1T2022/block-v1:HarvardX+PH125.1x+1T2022+type@sequential+block@bd2635e6b3eb44869432d655d3676357)

# Section 4 Overview

**Section 4 introduces you to general programming features like 'if-else' and 'for loop' commands so that you can write your own functions to perform various operations on datasets**.

**In Section 4.1**, you will:

* + Understand some of the programming capabilities of R.

**In Section 4.2**, you will:

* + Use basic conditional expressions to perform different operations.
  + Check if any or all elements of a logical vector are TRUE.

**In Section 4.3**, you will:

* + Define and call functions to perform various operations.
  + Pass arguments to functions, and return variables/objects from functions.

**In Section 4.4**, you will:

* + Use for-loops to perform repeated operations.
  + Articulate in-built functions of R that you could try for yourself.

# Basic Conditionals

**Key Points**

* The most common conditional expression in programming is an if-else statement, which has the form "if [condition], perform [expression], else perform [alternative expression]".
* The ifelse() function works similarly to an if-else statement, but it is particularly useful since it works on vectors by examining each element of the vector and returning a corresponding answer accordingly.
* The any() function takes a vector of logicals and returns true if any of the entries are true.
* The all() function takes a vector of logicals and returns true if all of the entries are true.

### Code

# an example showing the general structure of an if-else statement

a <- 0

if(a!=0){

print(1/a)

} else{

print("No reciprocal for 0.")

}

# an example that tells us which states, if any, have a murder rate less than 0.5

library(dslabs)

data(murders)

murder\_rate <- murders$total / murders$population\*100000

ind <- which.min(murder\_rate)

if(murder\_rate[ind] < 0.5){

print(murders$state[ind])

} else{

print("No state has murder rate that low")

}

# changing the condition to < 0.25 changes the result

if(murder\_rate[ind] < 0.25){

print(murders$state[ind])

} else{

print("No state has a murder rate that low.")

}

# the ifelse() function works similarly to an if-else conditional

a <- 0

ifelse(a > 0, 1/a, NA)

# the ifelse() function is particularly useful on vectors

a <- c(0,1,2,-4,5)

result <- ifelse(a > 0, 1/a, NA)

# the ifelse() function is also helpful for replacing missing values

data(na\_example)

no\_nas <- ifelse(is.na(na\_example), 0, na\_example)

sum(is.na(no\_nas))

# the any() and all() functions evaluate logical vectors

z <- c(TRUE, TRUE, FALSE)

any(z)

all(z)

# Functions

**Key points**

* The R function called function() tells R you are about to define a new function.
* Functions are objects, so must be assigned a variable name with the arrow operator.
* The general way to define functions is: (1) decide the function name, which will be an object, (2) type function() with your function's arguments in parentheses, (3) write all the operations inside brackets.
* Variables defined inside a function are not saved in the workspace.

### Code

# example of defining a function to compute the average of a vector x

avg <- function(x){

s <- sum(x)

n <- length(x)

s/n

}

# we see that the above function and the pre-built R mean() function are identical

x <- 1:100

identical(mean(x), avg(x))

# variables inside a function are not defined in the workspace

s <- 3

avg(1:10)

s

# the general form of a function

my\_function <- function(VARIABLE\_NAME){

perform operations on VARIABLE\_NAME and calculate VALUE

VALUE

}

# functions can have multiple arguments as well as default values

avg <- function(x, arithmetic = TRUE){

n <- length(x)

ifelse(arithmetic, sum(x)/n, prod(x)^(1/n))

}

# For Loops

**Key points**

* For-loops perform the same task over and over while changing the variable.  They let us define the range that our variable takes, and then changes the value with each loop and evaluates the expression every time inside the loop.
* The general form of a for-loop is: "For i in [some range], do operations".  This i changes across the range of values and the operations assume i is a value you're interested in computing on.
* At the end of the loop, the value of i is the last value of the range.

### Code

# creating a function that computes the sum of integers 1 through n

compute\_s\_n <- function(n){

x <- 1:n

sum(x)

}

# a very simple for-loop

for(i in 1:5){

print(i)

}

# a for-loop for our summation

m <- 25

s\_n <- vector(length = m) # create an empty vector

for(n in 1:m){

s\_n[n] <- compute\_s\_n(n)

}

# creating a plot for our summation function

n <- 1:m

plot(n, s\_n)

# a table of values comparing our function to the summation formula

head(data.frame(s\_n = s\_n, formula = n\*(n+1)/2))

# overlaying our function with the summation formula

plot(n, s\_n)

lines(n, n\*(n+1)/2)