Course 7 - Data Analysis with R Programming

Week 1 – Programming and data analytics

The R-versus-Python debate

Languages	R	Python
Common features	- Open-source - Data stored in data frames - Formulas and functions readily available - Community for code development and support	- Open-source - Data stored in data frames - Formulas and functions readily available - Community for code development and support
Unique advantages	- Data manipulation, data visualization, and statistics packages - "Scalpel" approach to data: *find packages to do what you want with the data*	- Easy syntax for machine learning needs - Integrates with cloud platforms like Google Cloud, Amazon Web Services, and Azure
Unique challenges	- Inconsistent naming conventions make it harder for beginners to select the right functions - Methods for handling variables may be a little complex for beginners to understand	- Many more decisions for beginners to make about data input/output, structure, variables, packages, and objects - "Swiss army knife" approach to data: *figure out a way to do what you want with the data*

Popular programming languages by profession

Let's go through some potential job titles you might encounter and the most popular programming languages used in those professions. Also included is a list of additional resources for you to explore and learn more about each of the programming languages introduced.

Data analyst

A data analyst collects, transforms, and organizes data to draw conclusions, make predictions, and drive informed decision-making. The most popular programming languages used by data analysts are R and Python.

R offers convenient statistical features for data analysis and is useful for creating advanced data visualizations. Check out these resources to learn more about R:

- The R Project for Statistical Computing: a website for downloading R, documentation, and help
- R Manuals: links to manuals from the R core team, including introduction, administration, and help
- Coding Club R Tutorials: a collection of coding tutorials for R

- R for Beginners: a starting guide to help you work with data, graphics, and statistics in R

Python is a general-purpose language that you can use to create what you need for data analysis. Here are a few resources to begin learning Python:

- The Python Software Foundation (PSF): a website with guides to help you get started as a beginner
- Python Tutorial: a Python 3 tutorial from the PSF site
- Coding Club Python Tutorials: a collection of coding tutorials for Python

Vectors and Lists in R

In programming, a **data structure** is a format for organizing and storing data. Data structures are important to understand because you will work with them frequently when you use R for data analysis. The most common data structures in the R programming language include:

- Vectors
- Data frames
- Matrices
- Arrays

Think of a data structure like a house that contains your data.

This reading will focus on vectors. Later on, you'll learn more about data frames, matrices, and arrays.

There are two types of vectors: **atomic vectors** and **lists**. Coming up, you'll learn about the basic properties of atomic vectors and lists, and how to use R code to create them.

Atomic vectors

First, we will go through the different types of atomic vectors. Then, you will learn how to use R code to create, identify, and name the vectors.

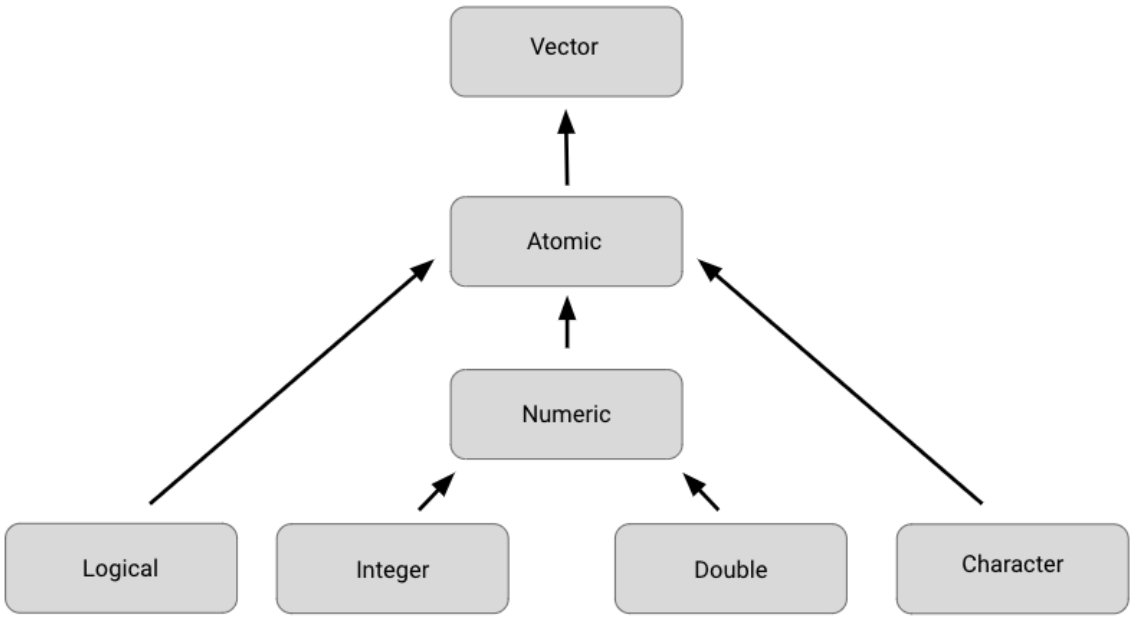
Earlier, you learned that a **vector** is a group of data elements of the *same* type, stored in a sequence in R. You cannot have a vector that contains both logicals and numerics.

There are six primary types of atomic vectors: logical, integer, double, character (which contains strings), complex, and raw. The last two–complex and raw–aren't as common in data analysis, so we will focus on the first four. Together, integer and double vectors are

known as numeric vectors because they both contain numbers. This table summarizes the four primary types:

Type	Description	Example
Logical	True/False	`TRUE`
Integer	Positive and negative whole values	`3`
Double	Decimal values	`101.175`
Character	String/character values	`"Coding"`

This diagram illustrates the hierarchy of relationships among these four main types of vectors:



Bottom: logical (arrow points to atomic), integer (arrow points to numeric), double (arrow points to numeric), character (arrow points to atomic) Second to bottom: numeric (arrow points to atomic) second level: atomic (arrow points to vector) top: vector

Creating vectors

One way to create a vector is by using the **c()** function (called the "combine" function). The c() function in R combines multiple values into a vector. In R, this function is just the letter "c" followed by the values you want in your vector inside the parentheses, separated by a comma: c(x, y, z, …).

For example, you can use the c() function to store numeric data in a vector.

```
c(2.5, 48.5, 101.5)
```

To create a vector of integers using the c() function, you must place the letter "L" directly after each number.

```
c(1L, 5L, 15L)
```

You can also create a vector containing characters or logicals.

```
c("Sara" , "Lisa" , "Anna")
```

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

Determining the properties of vectors

Every vector you create will have two key properties: type and length.

You can determine what type of vector you are working with by using the **typeof()** function. Place the code for the vector inside the parentheses of the function. When you run the function, R will tell you the type. For example:

```
typeof(c("a" , "b"))
```

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

Notice that the output of the typeof function in this example is `"character"`. Similarly, if you use the typeof function on a vector with integer values, then the output will include `"integer"` instead:

```
typeof(c(1L , 3L))
```

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

You can determine the length of an existing vector–meaning the number of elements it contains–by using the **length()** function. In this example, we use an assignment operator to assign the vector to the variable *x*. Then, we apply the length() function to the variable. When we run the function, R tells us the length is `3`.

```
x <- c(33.5, 57.75, 120.05)
```

```
length(x)
```

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

You can also check if a vector is a specific type by using an **is** function: **is.logical(), is.double(), is.integer(), is.character()**. In this example, R returns a value of `TRUE` because the vector contains integers.

```
x <- c(2L, 5L, 11L)
```

```
is.integer(x)
```

```
#> [1] TRUE
```

In this example, R returns a value of `FALSE` because the vector does *not* contain characters, rather it contains logicals.

```
y <- c(TRUE, TRUE, FALSE)
```

```
is.character(y)
```

```
#> [1] FALSE
```

Naming vectors

All types of vectors can be named. Names are useful for writing readable code and describing objects in R. You can name the elements of a vector with the **names()** function. As an example, let's assign the variable x to a new vector with three elements.

```
x <- c(1, 3, 5)
```

You can use the names() function to assign a different name to each element of the vector.

```
names(x) <- c("a", "b", "c")
```

Now, when you run the code, R shows that the first element of the vector is named `a`, the second `b`, and the third `c`.

```
x
```

```
#> a b c
```

```
#> 1 3 5
```

Remember that an atomic vector can only contain elements of the same type. If you want to store elements of different types in the same data structure, you can use a list.

Creating lists

Lists are different from atomic vectors because their elements can be of any type—like dates, data frames, vectors, matrices, and more. Lists can even contain other lists.

You can create a list with the **list()** function. Similar to the c() function, the list() function is just `list` followed by the values you want in your list inside parentheses: **list(x, y, z, ...)**. In this example, we create a list that contains four different kinds of elements: character (`"a"`), integer (`1L`), double (`1.5`), and logical (`TRUE`).

```
list("a", 1L, 1.5, TRUE)
```

Like we already mentioned, lists can contain other lists. If you want, you can even store a list inside a list inside a list—and so on.

```
list(list(list(1 , 3, 5)))
```

Determining the structure of lists

If you want to find out what types of elements a list contains, you can use the **str()** function. To do so, place the code for the list inside the parentheses of the function. When you run the function, R will display the data structure of the list by describing its elements and their types.

Let's apply the str() function to our first example of a list.

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

We run the function, then R tells us that the list contains four elements, and that the elements consist of four different types: character (`chr`), integer (`int`), number (`num`), and logical (`logi`).

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

Let's use the str() function to discover the structure of our second example. First, let's assign the list to the variable *z* to make it easier to input in the str() function.

```
z <- list(list(list(1 , 3, 5)))
```

Let's run the function.

```
str(z)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 3
#>   .. ..$ : num 1
#>   .. ..$ : num 3
#>   .. ..$ : num 5
```

The indentation of the `$` symbols reflect the nested structure of this list. Here, there are three levels (so there is a list within a list within a list).

Naming lists

Lists, like vectors, can be named. You can name the elements of a list when you first create it with the list() function:

```
list('Chicago' = 1, 'New York' = 2, 'Los Angeles' = 3)

$Chicago

[1] 1

$`New York`

[1] 2

$`Los Angeles`

[1] 3
```

Dates and times in R

Loading tidyverse and lubridate packages

Before you get started working with dates and times, you should load both **tidyverse** and **lubridate**. Lubridate is part of tidyverse.

First, open RStudio.

If you haven't already installed tidyverse, you can use the **install.packages()** function to do so:

- `install.packages("tidyverse")`

Next, load the tidyverse and lubridate packages using the **library()** function. First, load the core tidyverse to make it available in your current R session:

- `library(tidyverse)`

Then, load the lubridate package:

- `library(lubridate)`

Now you're ready to be introduced to the tools in the lubridate package.

Working with dates and times

This section covers the data types for dates and times in R and how to convert strings to date-time formats.

Types

In R, there are three types of data that refer to an instant in time:

- A date `("2016-08-16")`
- A time within a day `("20:11:59 UTC")`
- And a date-time. This is a date plus a time `("2018-03-31 18:15:48 UTC")`

The time is given in UTC, which stands for Universal Time Coordinated, more commonly called Universal Coordinated Time. This is the primary standard by which the world regulates clocks and time.

For example, to get the current date you can run the **today()** function. The date appears as year, month, and day.

```
today()
```

```
#> [1] "2021-01-20"
```

To get the current date-time you can run the **now()** function. Note that the time appears to the nearest second.

```
now()
```

```
#> [1] "2021-01-20 16:25:05 UTC"
```

When working with R, there are three ways you are likely to create date-time formats:

- From a string
- From an individual date
- From an existing date/time object

R creates dates in the standard yyyy-mm-dd format by default.

Let's go over each.

Converting from strings

Date/time data often comes as strings. You can convert strings into dates and date-times using the tools provided by lubridate. These tools automatically work out the date/time format. First, identify the order in which the year, month, and day appear in your dates. Then, arrange the letters *y*, *m*, and *d* in the same order. That gives you the name of the lubridate function that will parse your date. For example, for the date *2021-01-20,* you use the order *ymd*:

```
ymd("2021-01-20")
```

When you run the function, R returns the date in yyyy-mm-dd format.

```
#> [1] "2021-01-20"
```

It works the same way for any order. For example, month, day, and year. R still returns the date in yyyy-mm-dd format.

```
mdy("January 20th, 2021")
```

```
#> [1] "2021-01-20"
```

Or, day, month, and year. R still returns the date in yyyy-mm-dd format.

```
dmy("20-Jan-2021")
```

```
#> [1] "2021-01-20"
```

These functions also take unquoted numbers and convert them into the yyyy-mm-dd format.

```
ymd(20210120)
```

```
#> [1] "2021-01-20"
```

Creating date-time components

The ymd() function and its variations create dates. To create a date-time from a date, add an underscore and one or more of the letters *h*, *m*, and s (hours, minutes, seconds) to the name of the function:

```
ymd_hms("2021-01-20 20:11:59")
```

```
#> [1] "2021-01-20 20:11:59 UTC"
```

```
mdy_hm("01/20/2021 08:01")
```

```
#> [1] "2021-01-20 08:01:00 UTC"
```

Optional: Switching between existing date-time objects

Finally, you might want to switch between a date-time and a date.

You can use the function **as_date()** to convert a date-time to a date. For example, put the current date-time—now()—in the parentheses of the function.

```
as_date(now())
```

```
#> [1] "2021-01-20"
```

Other common data structures

Data structures

Recall that a data structure is like a house that contains your data.

Data frames

Data frames are the most common way of storing and analyzing data in R, so it's important to understand what they are and how to create them. A **data frame** is a collection of columns–similar to a spreadsheet or SQL table. Each column has a name at the top that represents a variable, and includes one observation per row. Data frames help summarize data and organize it into a format that is easy to read and use.

For example, the data frame below shows the "diamonds" dataset, which is one of the preloaded datasets in R. Each column contains a single variable that is related to diamonds: carat, cut, color, clarity, depth, and so on. Each row represents a single observation.

Filter

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75
6	0.24	Very Good	J	VVS2	62.8	57.0	336	3.94	3.96	2.48
7	0.24	Very Good	I	VVS1	62.3	57.0	336	3.95	3.98	2.47
8	0.26	Very Good	H	SI1	61.9	55.0	337	4.07	4.11	2.53
9	0.22	Fair	E	VS2	65.1	61.0	337	3.87	3.78	2.49
10	0.23	Very Good	H	VS1	59.4	61.0	338	4.00	4.05	2.39
11	0.30	Good	J	SI1	64.0	55.0	339	4.25	4.28	2.73
12	0.23	Ideal	J	VS1	62.8	56.0	340	3.93	3.90	2.46
13	0.22	Premium	F	SI1	60.4	61.0	342	3.88	3.84	2.33
14	0.31	Ideal	J	SI2	62.2	54.0	344	4.35	4.37	2.71
15	0.20	Premium	E	SI2	60.2	62.0	345	3.79	3.75	2.27
16	0.32	Premium	E	I1	60.9	58.0	345	4.38	4.42	2.68
17	0.30	Ideal	I	SI2	62.0	54.0	348	4.31	4.34	2.68
18	0.30	Good	J	SI1	63.4	54.0	351	4.23	4.29	2.70
19	0.30	Good	J	SI1	63.8	56.0	351	4.23	4.26	2.71
20	0.30	Very Good	J	SI1	62.7	59.0	351	4.21	4.27	2.66
21	0.30	Good	I	SI2	63.3	56.0	351	4.26	4.30	2.71

Showing 1 to 22 of 53,940 entries, 10 total columns

There are a few key things to keep in mind when you are working with data frames:

- First, columns should be named.
- Second, data frames can include many different types of data, like numeric, logical, or character.
- Finally, elements in the same column should be of the same type.

You will learn more about data frames later on in the program, but this is a great starting point.

If you need to manually create a data frame in R, you can use the **data.frame()** function. The data.frame() function takes vectors as input. In the parentheses, enter the name of the column, followed by an equals sign, and then the vector you want to input for that column. In this example, the *x* column is a vector with elements 1, 2, 3, and the *y* column is a vector with elements 1.5, 5.5, 7.5.

```
data.frame(x = c(1, 2, 3) , y = c(1.5, 5.5, 7.5))
```

If you run the function, R displays the data frame in ordered rows and columns.

```
  x y
1 1 1.5
```

```
2  2 5.5

3  3 7.5
```

In most cases, you won't need to manually create a data frame yourself, as you will typically import data from another source, such as a .csv file, a relational database, or a software program.

Files

Let's go over how to create, copy, and delete files in R. For more information on working with files in R, check out R documentation: files. **R documentation** is a tool that helps you easily find and browse the documentation of almost all R packages on CRAN. It's a useful reference guide for functions in R code. Let's go through a few of the most useful functions for working with files.

Use the **dir.create** function to create a new folder, or directory, to hold your files. Place the name of the folder in the parentheses of the function.

```
dir.create ("destination_folder")
```

Use the **file.create()** function to create a blank file. Place the name and the type of the file in the parentheses of the function. Your file types will usually be something like .txt, .docx, or .csv.

```
file.create ("new_text_file.txt")

file.create ("new_word_file.docx")

file.create ("new_csv_file.csv")
```

If the file is successfully created when you run the function, R will return a value of `TRUE` (if not, R will return `FALSE`).

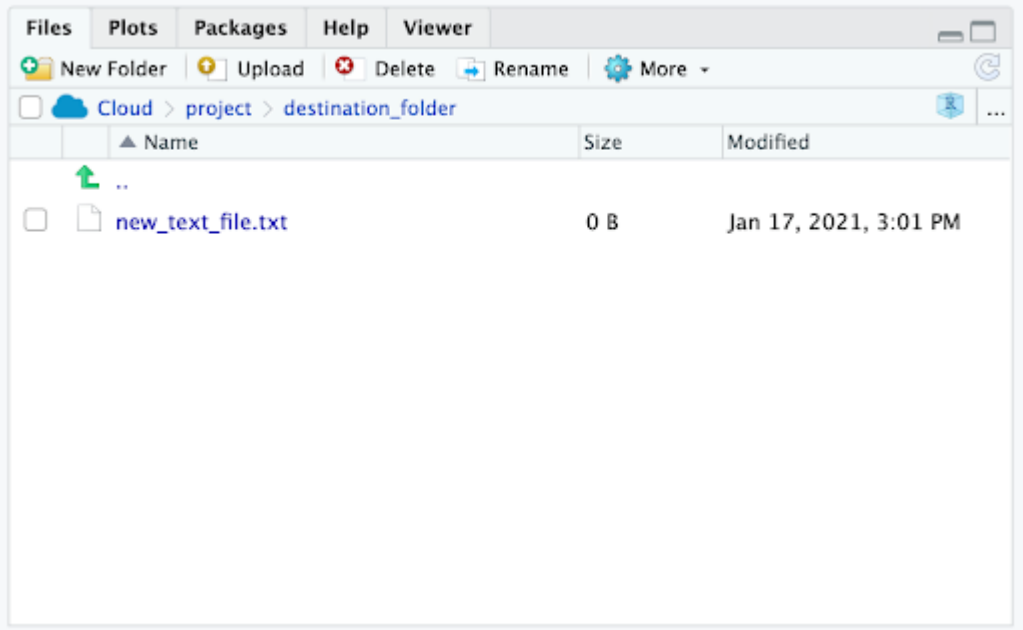
```
file.create ("new_csv_file.csv")

[1] TRUE
```

Copying a file can be done using the **file.copy()** function. In the parentheses, add the name of the file to be copied. Then, type a comma, and add the name of the destination folder that you want to copy the file to.

```
file.copy ("new_text_file.txt" , "destination_folder")
```

If you check the Files pane in RStudio, a copy of the file appears in the relevant folder:



You can delete R files using the **unlink()** function. Enter the file's name in the parentheses of the function.

```
unlink ("some_.file.csv")
```

Additional resource

If you want to learn more about working with data frames, matrices, and arrays in R, check out the Data Wrangling section of Stat Education's Introduction to R course. The section includes modules on data frames, matrices, and arrays (and more), and each module contains helpful examples of key coding concepts.

Optional: Matrices

A **matrix** is a two-dimensional collection of data elements. This means it has both rows and columns. By contrast, a vector is a one-dimensional sequence of data elements. But like vectors, matrices can only contain a single data type. For example, you can't have both logicals and numerics in a matrix.

To create a matrix in R, you can use the **matrix()** function. The matrix() function has two main arguments that you enter in the parentheses. First, add a vector. The vector contains the values you want to place in the matrix. Next, add at least one matrix dimension. You can choose to specify the number of rows or the number of columns by using the code `nrow =` or `ncol =`.

For example, imagine you want to create a 2x3 (two rows by three columns) matrix containing the values 3-8. First, enter a vector containing that series of numbers: `c(3:8)`. Then, enter a comma. Finally, enter `nrow = 2` to specify the number of rows.

```
matrix(c(3:8), nrow = 2)
```

If you run the function, R displays a matrix with three columns and two rows (typically referred to as a "2x3") that contain the numeric values 3, 4, 5, 6, 7, 8. R places the first value (3) of the vector in the uppermost row, and the leftmost column of the matrix, and continues the sequence from left to right.

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

You can also choose to specify the number of columns (`ncol =`) instead of the number of rows (`nrow =`).

```
matrix(c(3:8), ncol = 2)
```

When you run the function, R infers the number of rows automatically.

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

Logical operators and conditional statements

Logical operators

Logical operators return a logical data type such as TRUE or FALSE.

There are three primary types of logical operators:

- AND (sometimes represented as & or && in R)
- OR (sometimes represented as | or || in R)
- NOT (!)

Review the summarized logical operators below.

AND operator "&"

- The AND operator takes two logical values. It returns `TRUE` only if both individual values are TRUE. This means that TRUE & TRUE evaluates to `TRUE`. However, FALSE & TRUE, TRUE & FALSE, and FALSE & FALSE all evaluate to `FALSE`.

- If you run the corresponding code in R, you get the following results: `> TRUE & TRUE [1] TRUE > TRUE & FALSE [1] FALSE > FALSE & TRUE [1] FALSE > FALSE & FALSE [1] FALSE` You can illustrate this using the results of our comparisons. Imagine you create a variable x that is equal to 10. `x <- 10` To check if x is greater than 3 but less than 12, you can use x > 3 and x < 12 as the values of an "AND" expression. `x > 3 & x < 12` When you run the function, R returns the result TRUE. `[1] TRUE` The first part, `x > 3` will evaluate to `TRUE` since 10 is greater than 3. The second part, `x < 12` will also evaluate to `TRUE` since 10 is less than 12. So, since both values are TRUE, the result of the AND expression is `TRUE`. The number 10 lies between the numbers 3 and 12. However, if you make x equal to 20, the expression `x > 3 & x < 12` will return a different result. `x <- 20 x > 3 & x < 12 [1] FALSE` Although `x > 3` is `TRUE` (20 > 3), `x < 12` is `FALSE` (20 < 12). If one part of an AND expression is FALSE, the entire expression is FALSE (TRUE & FALSE = FALSE). So, R returns the result `FALSE`.

OR operator "|"

- The OR operator (|) works in a similar way to the AND operator (&). The main difference is that at least one of the values of the OR operation must be TRUE for the entire OR operation to evaluate to `TRUE`. This means that TRUE | TRUE, TRUE | FALSE, and FALSE | TRUE all evaluate to `TRUE`. When both values are FALSE, the result is `FALSE`.
- If you write out the code, you get the following results: `> TRUE | TRUE [1] TRUE > TRUE | FALSE [1] TRUE > FALSE | TRUE [1] TRUE > FALSE | FALSE [1] FALSE` For example, suppose you create a variable y equal to 7. To check if y is less than 8 or greater than 16, you can use the following expression: `y <- 7 y < 8 | y > 16` The comparison result is TRUE (7 is less than 8) | FALSE (7 is not greater than 16). Since only one value of an OR expression needs to be TRUE for the entire expression to be TRUE, R returns a result of TRUE. `[1] TRUE` Now, suppose y is 12. The expression y < 8 | y > 16 now evaluates to FALSE (12 < 8) | FALSE (12 > 16). Both comparisons are FALSE, so the result is `FALSE`. `y <- 12 y < 8 | y > 16 [1] FALSE`

NOT operator "!"

- The NOT operator (!) simply negates the logical value it applies to. In other words, !TRUE evaluates to `FALSE`, and !FALSE evaluates to `TRUE`.
- When you run the code, you get the following results: `> !TRUE [1] FALSE > !FALSE [1] TRUE` Just like the OR and AND operators, you can use the NOT operator in combination with logical operators. Zero is considered FALSE and non-zero numbers are taken as TRUE. The NOT operator evaluates to the opposite logical value. Let's imagine you have a variable x that equals 2: `x <- 2` The NOT operation evaluates to FALSE because it takes the opposite logical value of a non-zero number (TRUE). `> !x [1] FALSE`

Let's check out an example of how you might use logical operators to analyze data. Imagine you are working with the *airquality* dataset that is preloaded in RStudio. It contains data on daily air quality measurements in New York from May to September of 1973.

The data frame has six columns: *Ozone* (the ozone measurement), *Solar.R* (the solar measurement), *Wind* (the wind measurement), *Temp* (the temperature in Fahrenheit), and the *Month* and *Day* of these measurements (each row represents a specific month and day combination).

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4

Let's go through how the AND, OR, and NOT operators might be helpful in this situation.

AND example

Imagine you want to specify rows that are extremely sunny and windy, which you define as having a *Solar* measurement of over 150 and a *Wind* measurement of over 10.

In R, you can express this logical statement as `Solar.R > 150 & Wind > 10`.

Only the rows where *both* of these conditions are true fulfill the criteria:

	Ozone	Solar.R	Wind	Temp	Month	Day
1	18	313	11.5	62	5	4

OR example

Next, imagine you want to specify rows where it's extremely sunny or it's extremely windy, which you define as having a *Solar* measurement of over 150 or a *Wind* measurement of over 10.

In R, you can express this logical statement as `Solar.R > 150 | Wind > 10`.

All the rows where *either* of these conditions are true fulfill the criteria:

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	12	149	12.6	74	5	3
3	18	313	11.5	62	5	4

NOT example

Now, imagine you just want to focus on the weather measurements for days that aren't the first day of the month.

In R, you can express this logical statement as `Day != 1`.

The rows where this condition is true fulfill the criteria:

	Ozone	Solar.R	Wind	Temp	Month	Day
1	36	118	8.0	72	5	2
2	12	149	12.6	74	5	3
3	18	313	11.5	62	5	4

Finally, imagine you want to focus on scenarios that aren't extremely sunny and not extremely windy, based on your previous definitions of extremely sunny and extremely windy. In other words, the following statement should not be true: either a *Solar* measurement greater than 150 or a *Wind* measurement greater than 10.

Notice that this statement is the opposite of the OR statement used above. To express this statement in R, you can put an exclamation point (!) in front of the previous OR statement: `!(Solar.R > 150 | Wind > 10)`. R will apply the NOT operator to everything within the parentheses.

In this case, only one row fulfills the criteria:

	Ozone	Solar.R	Wind	Temp	Month	Day
1	36	118	8.0	72	5	2

Optional: Conditional statements

A **conditional statement** is a declaration that if a certain condition holds, then a certain event must take place. For example, "*If* the temperature is above freezing, *then* I will go outside for a walk." If the first condition is true (the temperature is above freezing), then

the second condition will occur (I will go for a walk). Conditional statements in R code have a similar logic.

Let's discuss how to create conditional statements in R using three related statements:

- **if()**
- **else()**
- **else if()**

if statement

The **if** statement sets a condition, and if the condition evaluates to `TRUE`, the R code associated with the if statement is executed.

In R, you place the code for the condition inside the parentheses of the if statement. The code that has to be executed if the condition is TRUE follows in curly braces (`expr`). Note that in this case, the second curly brace is placed on its own line of code and identifies the end of the code that you want to execute.

```
if (condition) {

 expr

}
```

For example, let's create a variable *x* equal to 4.

```
x <- 4
```

Next, let's create a conditional statement: if *x* is greater than 0, then R will print out the string `"x is a positive number".`

```
if (x > 0) {

  print("x is a positive number")

}
```

Since x = 4, the condition is true (4 > 0). Therefore, when you run the code, R prints out the string `"x is a positive number"`.

```
[1] "x is a positive number"
```

But if you change x to a negative number, like -4, then the condition will be FALSE (-4 > 0). If you run the code, R will not execute the print statement. Instead, a blank line will appear as the result.

else statement

The **else** statement is used in combination with an if statement. This is how the code is structured in R:

```
if (condition) {
  expr1
} else {
 expr2
}
```

The code associated with the else statement gets executed whenever the condition of the if statement is *not* TRUE. In other words, if the condition is TRUE, then R will execute the code in the if statement (*expr1*); if the condition is *not* TRUE, then R will execute the code in the else statement (*expr2*).

Let's try an example. First, create a variable *x* equal to 7.

```
x <- 7
```

Next, let's set up the following conditions:

- If x is greater than 0, R will print `"x is a positive number"`.
- If x is less than or equal to 0, R will print `"x is either a negative number or zero"`.

In our code, the first condition (x > 0) will be part of the if statement. The second condition of x less than or equal to 0 is implied in the else statement. If x > 0, then R will print `"x is a positive number"`. Otherwise, R will print `"x is either a negative number or zero"`.

```
x <- 7
if (x > 0) {
 print ("x is a positive number")
} else {
 print ("x is either a negative number or zero")
}
```

Since 7 is greater than 0, the condition of the if statement is true. So, when you run the code, R prints out `"x is a positive number"`.

```
[1] "x is a positive number"
```

But if you make x equal to -7, the condition of the if statement is *not* true (-7 is not greater than 0). Therefore, R will execute the code in the else statement. When you run the code, R prints out `"x is either a negative number or zero"`.

```
x <- -7

if (x > 0) {

 print("x is a positive number")

} else {

 print ("x is either a negative number or zero")

}

[1] "x is either a negative number or zero"
```

else if statement

In some cases, you might want to customize your conditional statement even further by adding the **else if** statement. The else if statement comes in between the if statement and the else statement. This is the code structure:

```
if (condition1) {

 expr1

} else if (condition2) {

 expr2

} else {

 expr3

}
```

If the if condition (*condition1*) is met, then R executes the code in the first expression (*expr1*). If the if condition is not met, and the else if condition (*condition2*) is met, then R executes the code in the second expression (*expr2*). If neither of the two conditions are met, R executes the code in the third expression (*expr3*).

In our previous example, using only the if and else statements, R can only print `"x is either a negative number or zero"` if x equals 0 or x is less than zero. Imagine you want R to print the string `"x is zero"` if x equals 0. You need to add another condition using the else if statement.

Let's try an example. First, create a variable *x* equal to negative 1 ("-1").

```
x <- -1
```

Now, you want to set up the following conditions:

- If x is less than 0, print `"x is a negative number"`
- If x equals 0, print `"x is zero"`
- Otherwise, print `"x is a positive number"`

In the code, the first condition will be part of the if statement, the second condition will be part of the else if statement, and the third condition will be part of the else statement. If x < 0, then R will print `"x is a negative number"`. If x = 0, then R will print `"x is zero"`. Otherwise, R will print `"x is a positive number"`.

```
x <- -1

if (x < 0) {

 print("x is a negative number")

} else if (x == 0) {

 print("x is zero")

} else {

 print("x is a positive number")

}
```

Since -1 is less than 0, the condition for the if statement evaluates to `TRUE`, and R prints `"x is a negative number"`.

```
[1] "x is a negative number"
```

If you make x equal to 0, R will first check the if condition `(x < 0)`, and determine that it is FALSE. Then, R will evaluate the else if condition. This condition, `x==0`, is TRUE. So, in this case, R prints `"x is zero"`.

If you make x equal to 1, both the if condition and the else if condition evaluate to `FALSE`. So, R will execute the else statement and print `"x is a positive number"`.

As soon as R discovers a condition that evaluates to TRUE, R executes the corresponding code and ignores the rest.

File-naming conventions

An important part of cleaning data is making sure that all of your files are accurately named. Although individual preferences will vary a bit, most analysts generally agree that file names should be accurate, consistent, and easy to read. This reading provides some general guidelines for you to follow when naming or renaming your data files.

What's in a (file)name?

When you first start working with R (or any other programming language, analysis tool, or platform, for that matter), you or your company should establish naming conventions for your files. This helps ensure that anyone reviewing your analysis–yourself included–can quickly and easily find what they need. Next are some helpful "do's" and "don'ts" to keep in mind when naming your files.

Do

- Keep your filenames to a reasonable length
- Use underscores and hyphens for readability
- Start or end your filename with a letter or number
- Use a standard date format when applicable; example: YYYY-MM-DD
- Use filenames for related files that work well with default ordering; example: in chronological order, or logical order using numbers first

Examples of good filenames
2020-04-10_march-attendance.R
2021_03_20_new_customer_ids.csv
01_data-sales.html
02_data-sales.html

Don't

- Use unnecessary additional characters in filenames
- Use spaces or "illegal" characters; examples: &, %, #, <, or >
- Start or end your filename with a symbol
- Use incomplete or inconsistent date formats; example: M-D-YY
- Use filenames for related files that do not work well with default ordering; examples: a random system of numbers or date formats, or using letters first

Examples of filenames to avoid
4102020marchattendance<workinprogress>.R
_20210320*newcustomeridsforfebonly.csv
firstfile_for_datasales/1-25-2020.html
secondfile_for_datasales/2-5-2020.html