Learning Objectives: Using Statistical Functions on Vectors

- **Apply the following statistical functions to vectors:**
  - **mean()**
  - **median()**
  - **get_mode()**
  - **max()**
  - **minimum()**
  - **range()**

definition

Assumptions

- Learners are comfortable creating vectors with numerical elements.
- Learners are comfortable applying basic arithmetic operations such as add, subtract, multiply, and divide.

Limitations

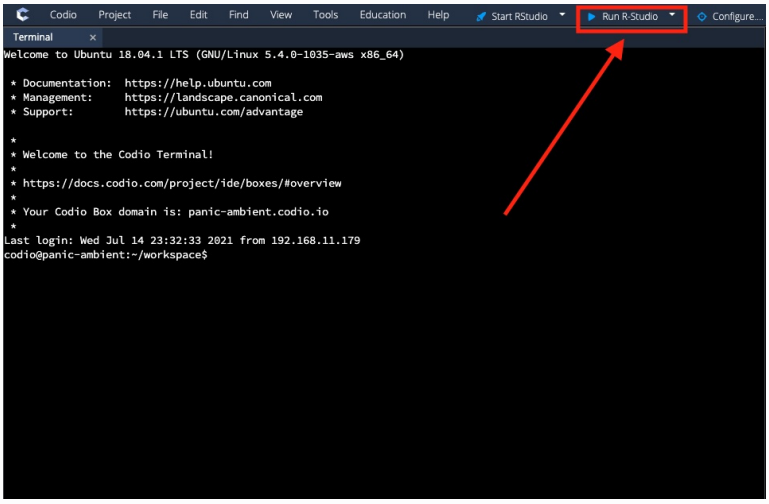
- This section will only cover the most essential statistical functions.

Introduction to Vectors

Vectors

Vectors are objects within R that hold a collection of data. Previously, you've learned how to use basic data types and variables that hold a single piece of data. Now, you'll learn how to use vectors to store multiple pieces of data.

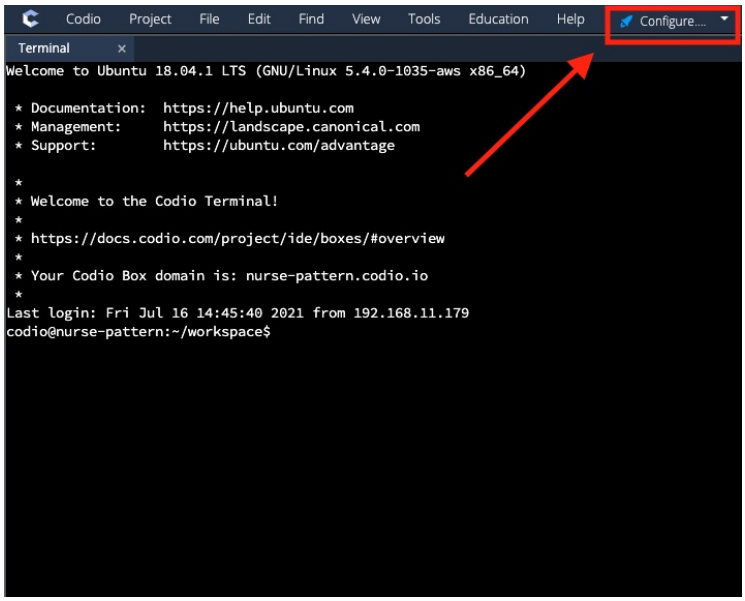
Before we begin, start RStudio by clicking on the `Run R-Studio` button towards the top of the screen if it isn't already opened.



.guides/img/num/run-r-studio

▼ **Help, I don't see that button!**

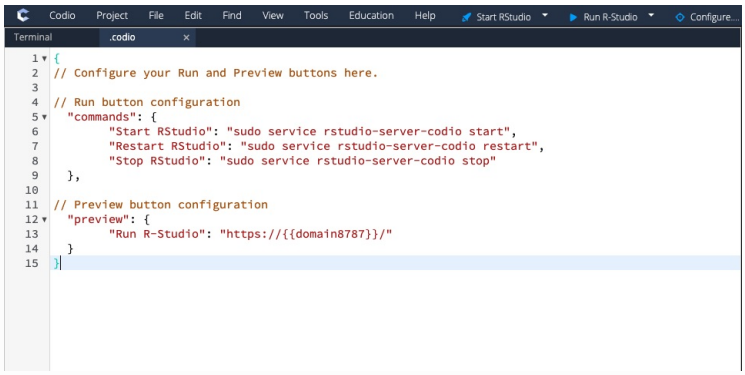
If you do not see the `Run R-Studio` button, click on the `Configure...` button instead.



.guides/img/num/configure

This will open up a file called `.codio`. Then copy the following code into the file:

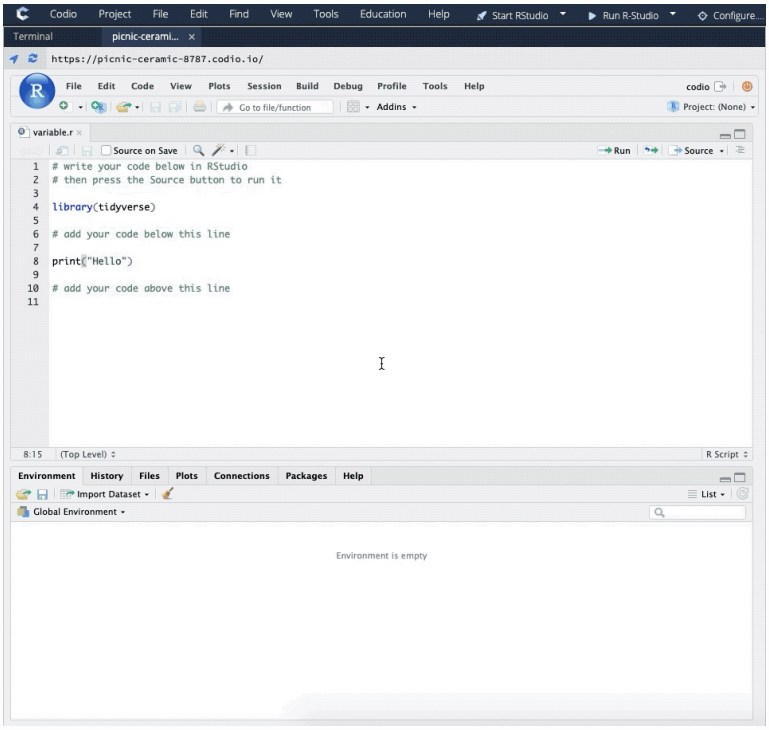
```
{
// Configure your Run and Preview buttons here.

// Run button configuration
  "commands": {
        "Start RStudio": "sudo service rstudio-server-codio
        start",
        "Restart RStudio": "sudo service rstudio-server-codio
        restart",
        "Stop RStudio": "sudo service rstudio-server-codio
        stop"
  },

// Preview button configuration
  "preview": {
        "Run R-Studio": "https://{{domain8787}}/"
  }
}
```



.guides/img/num/configure-codio

Doing so will enable the `Run R-Studio` button to activate.

Then, you'll want to open up the `vector.r` file within RStudio. To do so, select `File` from the top menu in RStudio, and then choose `Open File...` —> `code` —> `num` —> `vector.r`. Then drag the right border line towards the middle to reveal the `Console` if it is hidden (see reminder image below).



.guides/img/num/show-console

Creating a Vector

You can create a vector similarly to how you created a variable in the previous section. The difference is, in order to create a collection of data, you use the `c` function to separate the data you want to be stored in the vector.

For example, if I want to add the numerics `2`, `-5`, `45`, and `13` to my vector `x`, I can use the following syntax:

```
x <- c(2, -5, 45, 13)
```

What this does is it takes all of the numerical pieces of data I chose and stores them inside the vector `x`.

I can print the vector by using the print statement:

```
print(x)
```

Which will then result in:

```
[1]  2 -5 45 13
```

Vector Elements

We sometimes refer to each piece of data within a vector to be an **element**. It's **important** to note that all data **should** be the same type within a vector. If they are not, then RStudio will automatically convert the data into the same type, whichever is the most flexible.

For example, if I have a mixture of numerics, logicals, and characters within my vector.

```
x <- c(2, -5, "apple", "banana", TRUE)
print(x)
```

RStudio will convert all elements into characters because character is the most flexible data type.

If I have just numerics and logicals, RStudio will convert the logicals into numerics because the reverse might not always be possible. For example, `TRUE` will always convert to `1` and the reverse is also possible; however, `2` will not convert to `TRUE`. Here, numerics become the more flexible data type because all logicals can be converted to numerics.

```
x <- c(2, -5, TRUE)
print(x)
```

Selecting Vector Elements

To select and print a **particular** vector element, you can use brackets `[]` to denote which element you want to be selected and/or printed. For example, if I want to select and print just the second element in the vector, I can use:

```
x <- c(2, -5, "apple", "banana", TRUE)
print(x[2])
```

which returns `[1] "-5"` because `-5` is the second element. To print a **range** of vector elements, specify the first element you want selected followed by a colon `:`, and lastly specify the last element you want selected. RStudio will then select **all** elements between and including the first and last elements. For example:

```
x <- c(2, -5, "apple", "banana", TRUE)
print(x[2:4])
```

returns `[1] "-5"  "apple"  "banana"`. The `2` in `print(x[2:4])` refers to the second element `-5` and the `4` refers to the fourth element `"banana"`. RStudio also selects all elements in between them (such as `"apple"`).

Mean

Functions

Functions are extremely important in R because they help to calculate certain information quickly. We'll start with the `mean()` function.

Mean

The `mean()` function calculates the mean or average of the data provided. Average is calculated by adding all of the pieces of data together and then dividing by the number of data pieces used.

For example,

```
x <- c(2, -5, 45, 13)
print(mean(x))
```

results in

```
13.75
```

because (2 + -5 + 45 + 13) divided by 4 is 13.75.

Note that the `mean()` function does not work with the character data type.

Median

Median

The `median()` function calculates the median or middle number of the data provided.

For example,

```
x <- c(2, -5, 45, 13)
print(median(x))
```

results in

```
7.5
```

Note that when calculating median, the system will first sort all of the data from smallest to largest (-5, 2, 13, 45) and then returns the middle number if there is an **odd** number of data. However, if there is an **even** number of data, the system will calculate the average or mean of the two middle numbers. In our case, the two middle numbers are `2` and `13` and their average is `7.5`.

If I add an additional numeric to the vector `x` such as `-5`, and then try to calculate the median,

```
x <- c(2, -5, 45, 13, -5)
print(median(x))
```

I will get

```
[1] 2
```

because `2` is now the middle number within the sorted data set (-5, -5, **2**, 13, 45).

Like `mean()`, the `median()` function **does not** work with the character data type.

Mode

Mode

Unlike mean and median, there is no **mode** function to help us calculate the data that appears the most within a data set or vector. However, we can build our own `get_mode()` function using the syntax below:

```
get_mode <- function(f) {
  uf <- unique(f)
  tab <- tabulate(match(f, uf))
  uf[tab == max(tab)]
}
```

You'll notice that the keyword `function` is used in the code above which helps us to create the `get_mode()` function. Within the curly braces `{}` there are lines of code containing commands to instruct the program what to do when the `get_mode()` function is called. For now, functions will be provided to you so don't worry about creating them. However, if you are curious about how to create R functions, you can visit this link here to learn more.

Now that you have the `get_mode()` function, you can use it to find the mode within your vector `x`. Be sure to continue to include the instructions for the `get_mode()` function within your code, otherwise, it will not work.

```
get_mode <- function(f) {
  uf <- unique(f)
  tab <- tabulate(match(f, uf))
  uf[tab == max(tab)]
}

x <- c(2, -5, 45, 13, -5)
print(get_mode(x))
```

The code above will produce

```
[1] -5
```

which makes sense because `-5` occurs the most within the vector.

If I add additional data to the vector such as `13` and `45`, the function will now return multiple elements because I have multiple modes within my data.

```
get_mode <- function(f) {
  uf <- unique(f)
  tab <- tabulate(match(f, uf))
  uf[tab == max(tab)]
}

x <- c(2, -5, 45, 13, -5, 13, 45)
print(get_mode(x))
```

```
[1] -5 45 13
```

Unlike `mean()` and `median()`, we can use `get_mode()` on character data. For example,

```
get_mode <- function(f) {
  uf <- unique(f)
  tab <- tabulate(match(f, uf))
  uf[tab == max(tab)]
}

y <- c("cat", "dog", "cat", "lizard", "lizard")
print(get_mode(y))
```

results in

```
[1] "cat"    "lizard"
```

because `"cat"` and `"lizard"` appear the most in the vector `y`.

Maximum

Maximum

To find the maximum data within a vector, we can use the `max()` function.

For example,

```
x <- c(2, -5, 45, 13, -5, 13, 45)
print(max(x))
```

results in

```
[1] 45
```

because `45` is largest number in the vector.

Minimum

Minimum

To find the minimum data within a vector, we can use the `min()` function.

For example,

```
x <- c(2, -5, 45, 13, -5, 13, 45)
print(min(x))
```

results in

```
[1] -5
```

because `-5` is the smallest number in the vector.

Range

Range

To find the range, or the largest and smallest data values within a vector, we can use the `range()` function. It will return the smallest data value followed by the largest.

For example,

```
x <- c(2, -5, 45, 13, -5, 13, 45)
print(range(x))
```

results in

```
[1] -5 45
```

because `-5` is the smallest value while `45` is the largest value in the vector.

To actually calculate the range itself, which is the difference between the largest and smallest data values, use `max() - min()`.

For example,

```
x <- c(2, -5, 45, 13, -5, 13, 45)
print(max(x) - min(x))
```

results in

```
[1] 50
```

because if the largest value `45` subtracts the smallest value `-5` (`45 - -5`), we will get `50`.

Sort

Sort

Although not a statical function, you can use the `sort()` function to sort your vector elements. The basic syntax to sort vector elements is:

```
sort(x, decreasing = FALSE, na.last = TRUE)
```

where:
* `x` represents the vector you want sorted
* `decreasing` tells the system whether you want the elements to be sorted in descending order (`TRUE`) or ascending order (`FALSE`)
* By default, the `sort()` function will sort data in **ascending** order.
* `na.last` tells the system if you want "not available" or `NA` data to be sorted last (`TRUE`) or first (`FALSE`)
* An example of `NA` data is `0` being divided by `0` in which the system will return `NaN`

For example, if I have the following vector:

```
x <- c(2, -5, 45, 13, -5, 13, 45, (0/0))
```

I can use the sort function to sort `x` in ascending order:

```
x <- c(2, -5, 45, 13, -5, 13, 45, (0/0))

print(sort(x, decreasing = FALSE, na.last = TRUE))
```

which results in:

```
[1]  -5  -5   2  13  13  45  45 NaN
```

or in descending order:

```
x <- c(2, -5, 45, 13, -5, 13, 45, (0/0))

print(sort(x, decreasing = TRUE, na.last = TRUE))
```

which results in:

```
[1]  45  45  13  13   2  -5  -5 NaN
```

To have the `NA` data sorted first, I can change `na.last = TRUE` to `na.last = FALSE`:

```
x <- c(2, -5, 45, 13, -5, 13, 45, (0/0))

print(sort(x, decreasing = TRUE, na.last = FALSE))
```

which results in:

```
[1] NaN  45  45  13  13   2  -5  -5
```