Learning Objectives: Variables

- **Open an R file in RStudio**
- **Run an R File in RStudio**
- **Create a variable**
- **Store numerical data in a variable**
- **Print the data stored in variables**
- **Apply arithmetic and logical operators to data types and variables**

definition

Limitations

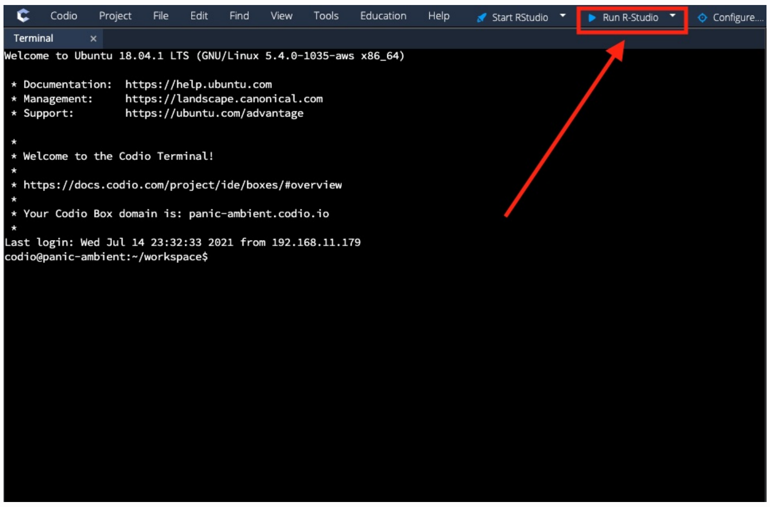
- This section will primarily cover variables and vectors. Data frames and functions will be covered in a later section.

Navigating RStudio

RStudio

RStudio is a free, open-source, Integrated Development Environment for R, which is one of the most commonly used programming languages for data science. In particular, we will be using R and RStudio to analyze data and create visualizations to represent that data. For this course, we will focus on the data analysis component.

Let's start RStudio by clicking on the `Run R-Studio` button towards the top of the screen.

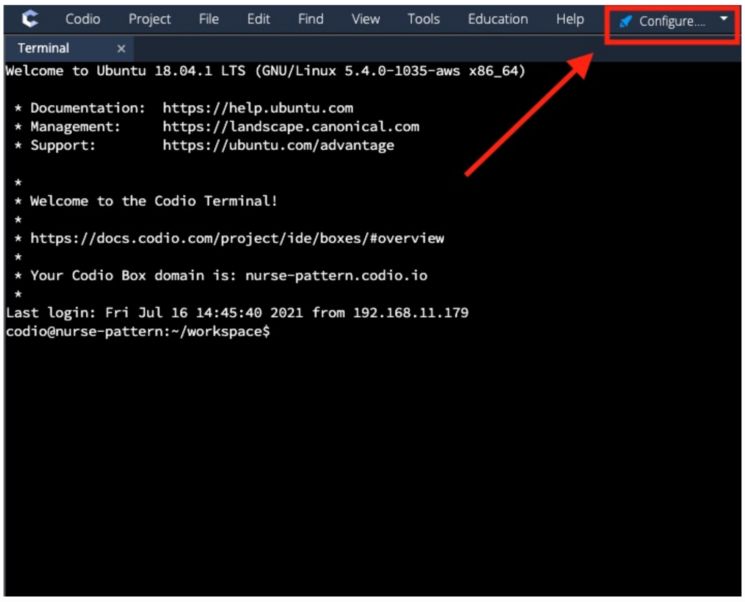


```
Welcome to Ubuntu 18.04.1 LTS (GNU/Linux 5.4.0-1035-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 *
 * Welcome to the Codio Terminal!
 *
 * https://docs.codio.com/project/ide/boxes/#overview
 *
 * Your Codio Box domain is: panic-ambient.codio.io
 *
Last login: Wed Jul 14 23:32:33 2021 from 192.168.11.179
codio@panic-ambient:~/workspace$
```

.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.

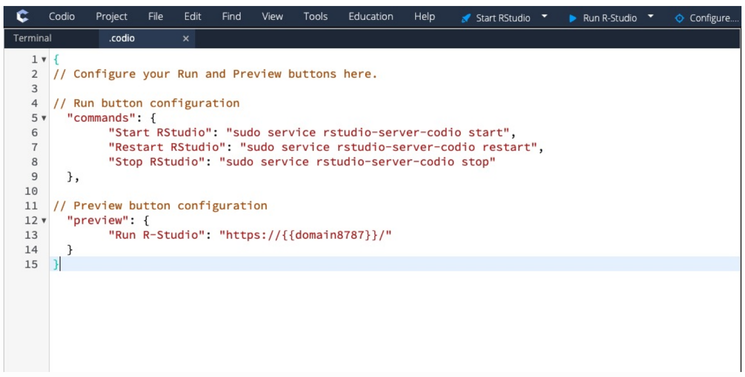


```
Welcome to Ubuntu 18.04.1 LTS (GNU/Linux 5.4.0-1035-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 *
 * Welcome to the Codio Terminal!
 *
 * https://docs.codio.com/project/ide/boxes/#overview
 *
 * Your Codio Box domain is: nurse-pattern.codio.io
 *
Last login: Fri Jul 16 14:45:40 2021 from 192.168.11.179
codio@nurse-pattern:~/workspace$
```

.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}}/"
  }
}
```

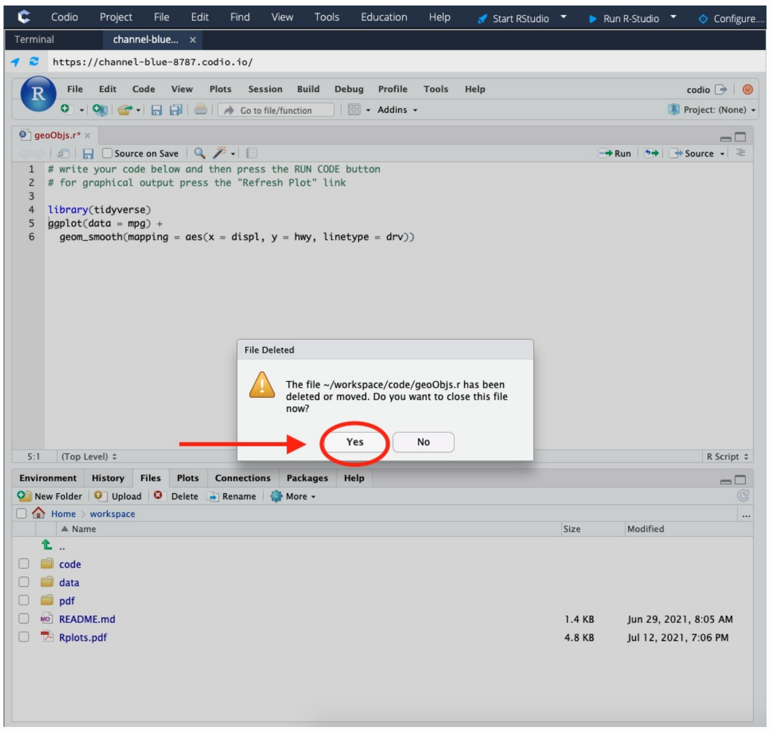


```
{
// Configure your Run and Preview buttons here.

// 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.

Once RStudio opens, you might encounter a message that says that a previous file has been deleted or moved and asks if you want to close the file. Select `Yes`.



```
# write your code below and then press the RUN CODE button
# for graphical output press the "Refresh Plot" link

library(tidyverse)
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, linetype = drv))
```

File Deleted

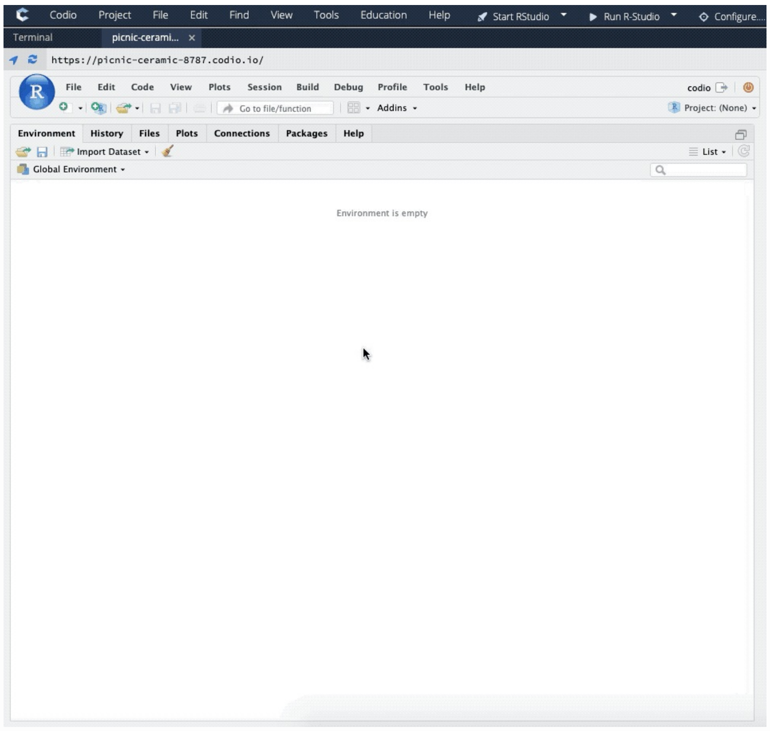
The file ~/workspace/code/geoObjs.r has been deleted or moved. Do you want to close this file now?

[✓ Yes] No

Name	Size	Modified
..		
code		
data		
pdf		
README.md	1.4 KB	Jun 29, 2021, 8:05 AM
Rplots.pdf	4.8 KB	Jul 12, 2021, 7:06 PM

.guides/img/num/close-file

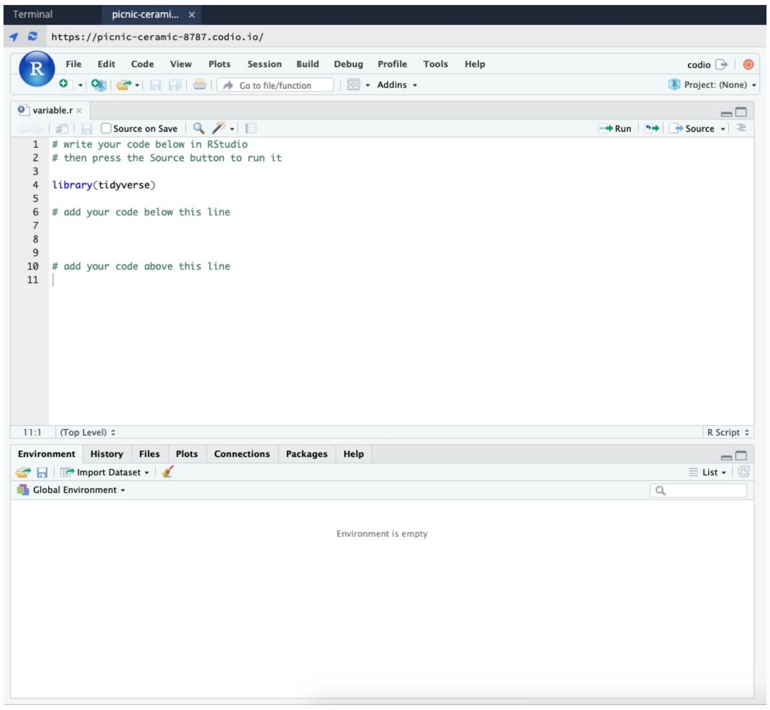
After you select `Yes`, you want to open the file `variable.r` by selecting `File` from the top menu in RStudio, and then choosing `Open File...` —> `code` —> `num` —> `variable.r`



.guides/img/num/open-variables-r

Working in RStudio

Make sure your screen looks something like this:



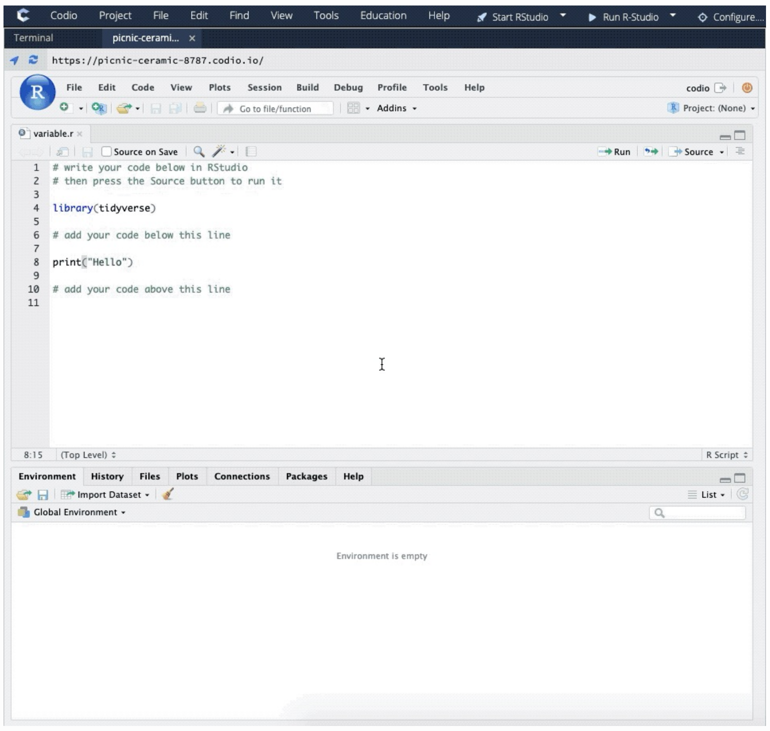
```
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line



# add your code above this line

```

.guides/img/num/variables-r

Once you've opened `variable.r` successfully, you are ready to create your first R program. Add the following code into the text editor in RStudio between the lines that read `# add your code below this line` and `# add your code above this line`. For now, don't worry about the line that reads `library(tidyverse)`, this is a library that we will make use of later.

```r
print("Hello")
```

After entering the code specified, move your mouse to the right side of RStudio, click and drag the right border line towards the center of the panel. Doing so enables you to see the `Console` which shows what gets printed from the program that you create. Then click the `Source` button as shown which will run the program.

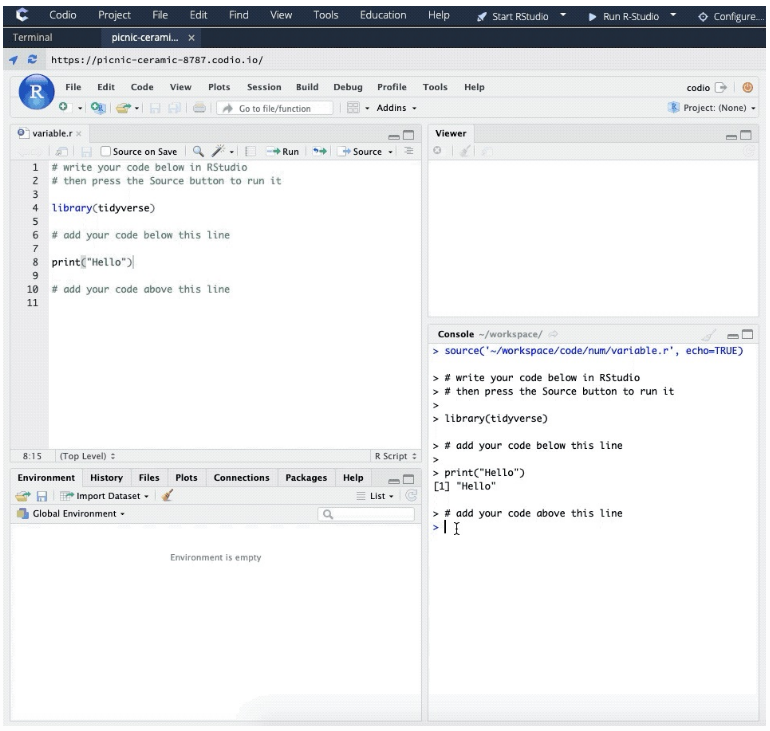


```r
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

print("Hello")

# add your code above this line

```

Environment is empty

.guides/img/num/show-console

Alternatively, in RStudio, you can select `View` —> `Panes` —> `Console on Left` to display the console right below the text editor.

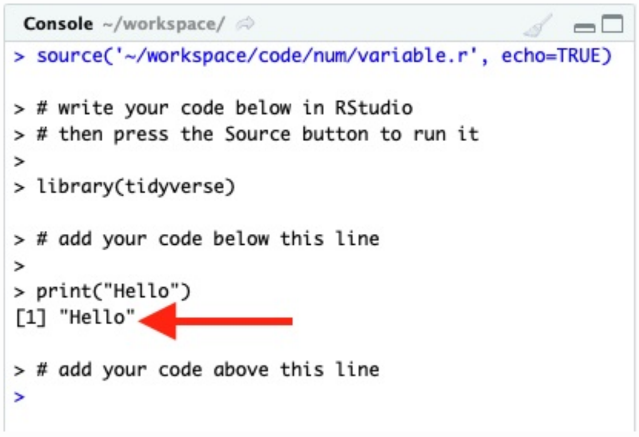


```
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

print("Hello")

# add your code above this line

```

```
> source('~/workspace/code/num/variable.r', echo=TRUE)

> # write your code below in RStudio
> # then press the Source button to run it
>
> library(tidyverse)

> # add your code below this line
>
> print("Hello")
[1] "Hello"

> # add your code above this line
>
```

.guides/img/num/view-console-left

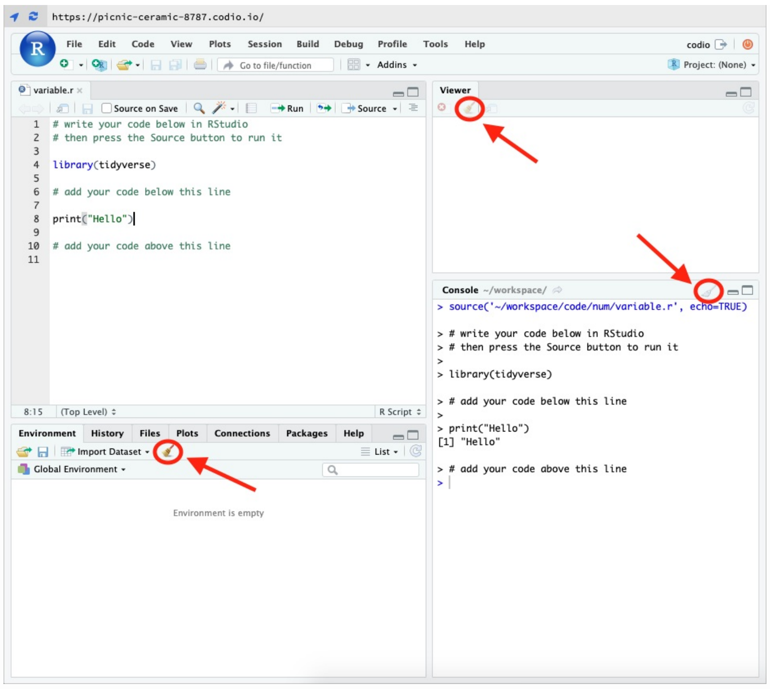
After you click the `Source` button, the program runs and then prints `[1] "Hello"`. Note that the `[1]` stands for the line number and `"Hello"` stands for the value or output that is printed. Any lines containing > are not printed.



```
> source('~/workspace/code/num/variable.r', echo=TRUE)

> # write your code below in RStudio
> # then press the Source button to run it
>
> library(tidyverse)

> # add your code below this line
>
> print("Hello")
[1] "Hello"

> # add your code above this line
>
```

.guides/img/num/print-hello

If at any point you want to clear the items that you entered or printed, you can click on the broomstick button to clear them.

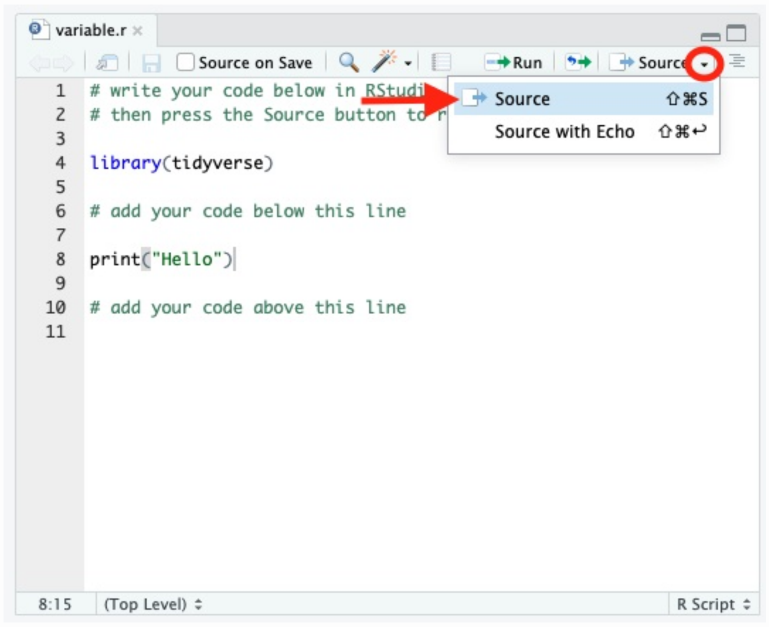


```
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

print("Hello")

# add your code above this line
```

```
> source('~/workspace/code/num/variable.r', echo=TRUE)

> # write your code below in RStudio
> # then press the Source button to run it
>
> library(tidyverse)

> # add your code below this line
>
> print("Hello")
[1] "Hello"

> # add your code above this line
>
```

.guides/img/num/broomstick

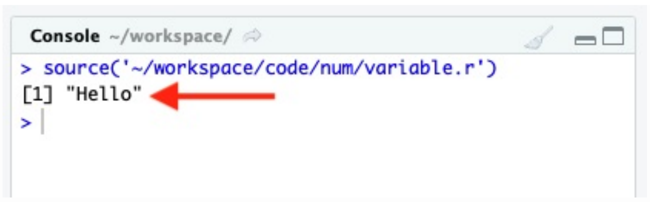
If you **only** want to see what gets **printed** within the console, you can click the small drop-down arrow within the `Source` button, then click the first option.



```
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

print("Hello")

# add your code above this line
```

.guides/img/num/source-without-comments

This will cause the console to only show what actually gets printed and ignore comments.



```
> source('~/workspace/code/num/variable.r')
[1] "Hello"
>
```

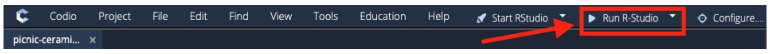
.guides/img/num/print-without-comments

Note that you only have to do this once for the button to only show printed items from now on. If you want to revert back to showing comments in the console, simply select the second option (`Source with Echo`) from the drop-down option next time.

Now that you understand how to navigate RStudio a little better, we will move on to learning more about data types!

Basic Data Types

As you navigate through the pages, if you notice that RStudio is not opened, click on the `Run R-Studio` button at the top menu to activate it.

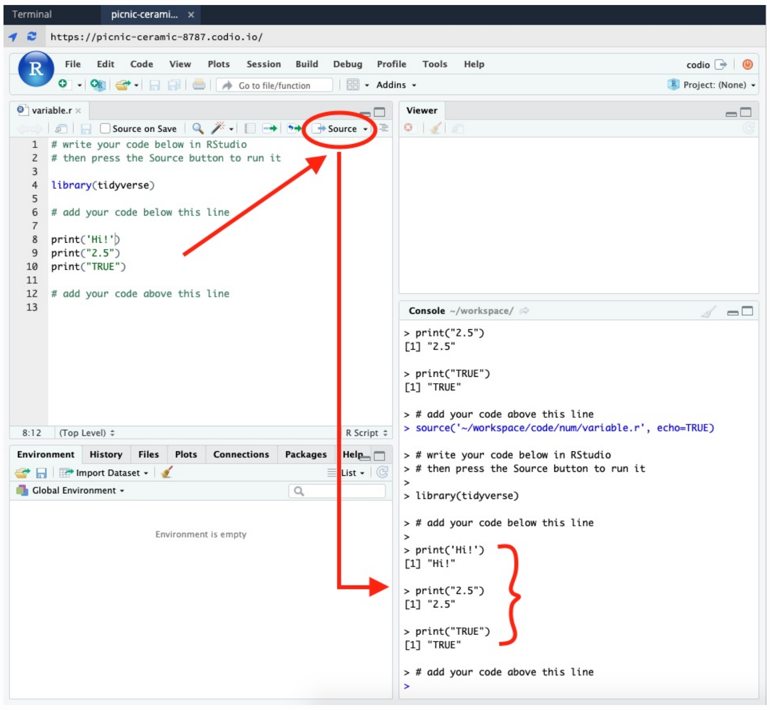


.guides/img/num/activate-rstudio

Character

When you printed `"Hello"` earlier, you have actually encountered the **character** data type. The character data type is presented within double quotes. All of the following are examples of characters. Copy the code below into RStudio and then click the `Source` button to print the characters.

```r
print("Hi!")
print("2.5")
print("TRUE")
```



```r
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

print('Hi!')
print("2.5")
print("TRUE")

# add your code above this line

```

```
> print("2.5")
[1] "2.5"

> print("TRUE")
[1] "TRUE"

> # add your code above this line
> source('~/workspace/code/num/variable.r', echo=TRUE)

> # write your code below in RStudio
> # then press the Source button to run it
>
> library(tidyverse)

> # add your code below this line
>
> print('Hi!')
[1] "Hi!"

> print("2.5")
[1] "2.5"

> print("TRUE")
[1] "TRUE"

> # add your code above this line
>
```

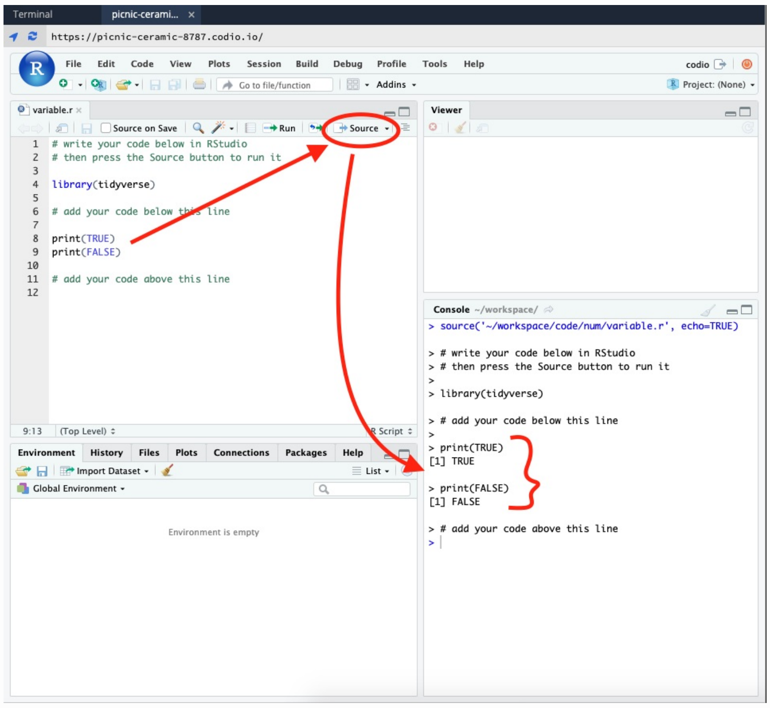
.guides/img/num/print-char

Note that you can also use single quotes (`''`) to denote the data as the character type. For example, `print('Hi!')`, `print('2.5')`, and `print('TRUE')`.

Logical

Logical can be either `TRUE` or `FALSE`. These are used to determine if certain conditions are true or not. We will cover these in more detail on a later page. Note that you can also use the shortcut `T` for `TRUE` and `F` for `FALSE`, but it is a best practice to write out the entire logical to avoid confusion. Remove the print character statements and replace them with the following. Then click the `Source` button to see the output.

```r
print(TRUE)
print(FALSE)
```



```r
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

print(TRUE)
print(FALSE)

# add your code above this line
```

```
> source('~/workspace/code/num/variable.r', echo=TRUE)

> # write your code below in RStudio
> # then press the Source button to run it
>
> library(tidyverse)

> # add your code below this line
>
> print(TRUE)
[1] TRUE

> print(FALSE)
[1] FALSE

> # add your code above this line
> |
```

.guides/img/num/print-logical

challenge

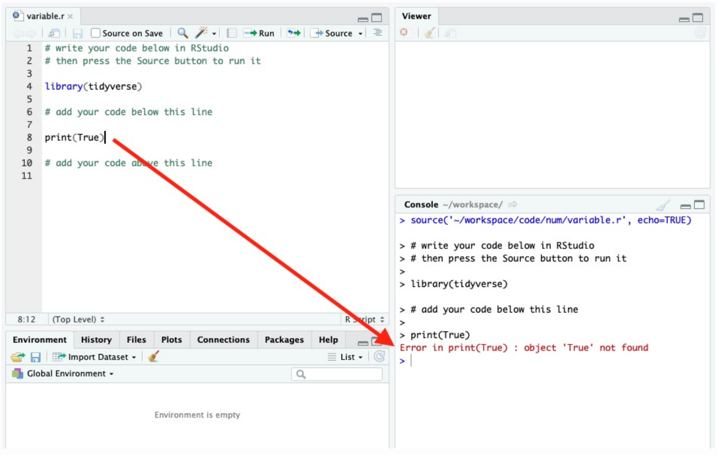
Try this variation(s):

```
print(True)
print(true)
print("TRUE")
```

important

IMPORTANT

`True`, `true`, and `"TRUE"` are not the same as `TRUE`. `TRUE` is a logical in R, but the rest are not. Both `True` and `true` will be considered as variables (or objects) while `"TRUE"` is a character type. You'll notice that printing `True` and `true` results in an error.



```
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

print(True)

# add your code above this line

```

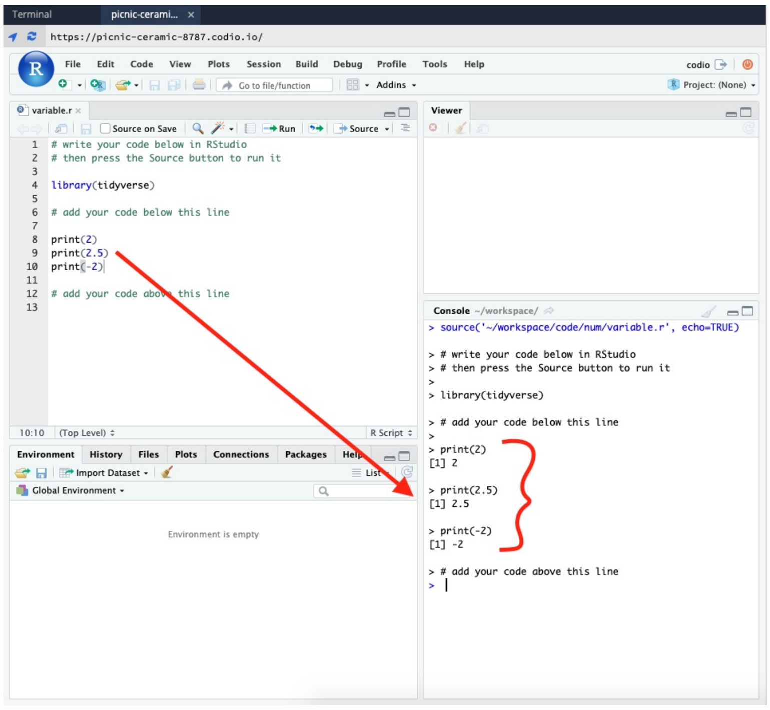
```
> source('~/workspace/code/num/variable.r', echo=TRUE)

> # write your code below in RStudio
> # then press the Source button to run it
>
> library(tidyverse)

> # add your code below this line
>
> print(True)
Error in print(True) : object 'True' not found
>
```

.guides/img/num/true-error

Numeric

The last major data type is **numeric** and as the name suggests, it involves numbers. Numerics can be broken down into further subtypes (e.g. doubles, integers, and complex numbers) but for the purposes of this course, we will focus mainly on the **double** subtype which is the default subtype. Try these numeric examples in RStudio:

```
print(2)
print(2.5)
print(-2)
```



```
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

print(2)
print(2.5)
print(-2)

# add your code above this line
```

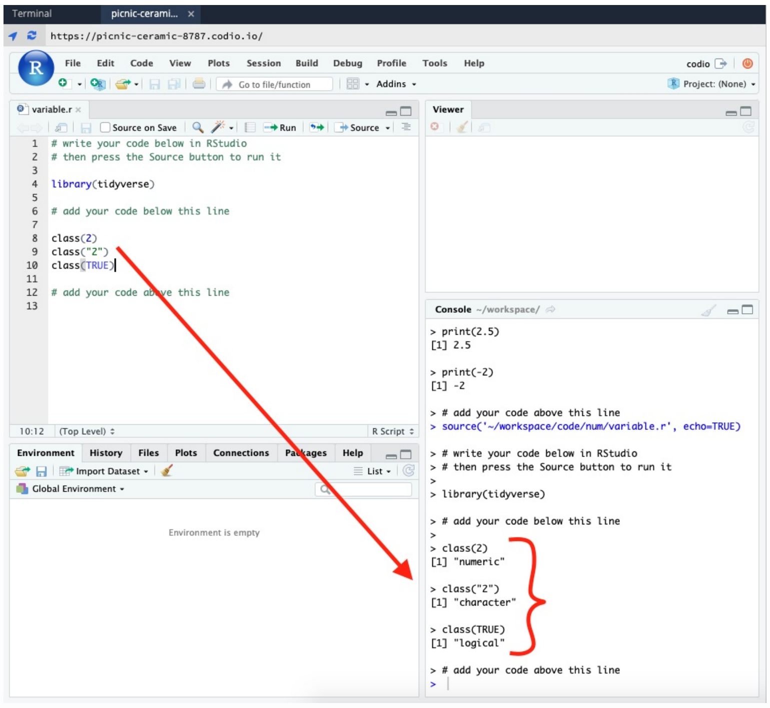
```
> source('~/workspace/code/num/variable.r', echo=TRUE)

> # write your code below in RStudio
> # then press the Source button to run it
>
> library(tidyverse)

> # add your code below this line
>
> print(2)
[1] 2

> print(2.5)
[1] 2.5

> print(-2)
[1] -2

> # add your code above this line
>
```

.guides/img/num/print-numeric

The class() Function

`class()` is similar to `print()` in that they are both functions that print things to the console; however, `class()` will print what type of data you have within the parentheses whereas `print()` just prints the data itself. Try the following `class()` statements in RStudio:

```
class(2)
class("2")
class(TRUE)
```



```
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

class(2)
class("2")
class(TRUE)

# add your code above this line
```

```
> print(2.5)
[1] 2.5

> print(-2)
[1] -2

> # add your code above this line
> source('~/workspace/code/num/variable.r', echo=TRUE)

> # write your code below in RStudio
> # then press the Source button to run it
>
> library(tidyverse)

> # add your code below this line
>
> class(2)
[1] "numeric"

> class("2")
[1] "character"

> class(TRUE)
[1] "logical"

> # add your code above this line
>
```

.guides/img/num/class-function

You'll notice that `class(2)` prints `"numeric"`, `class("2")` prints `"character"`, and `class(TRUE)` prints `"logical"`. If at any time you are unsure of what data type you are using, you can always take advantage of the `class()` function to help you out.

Creating Variables to Store Data

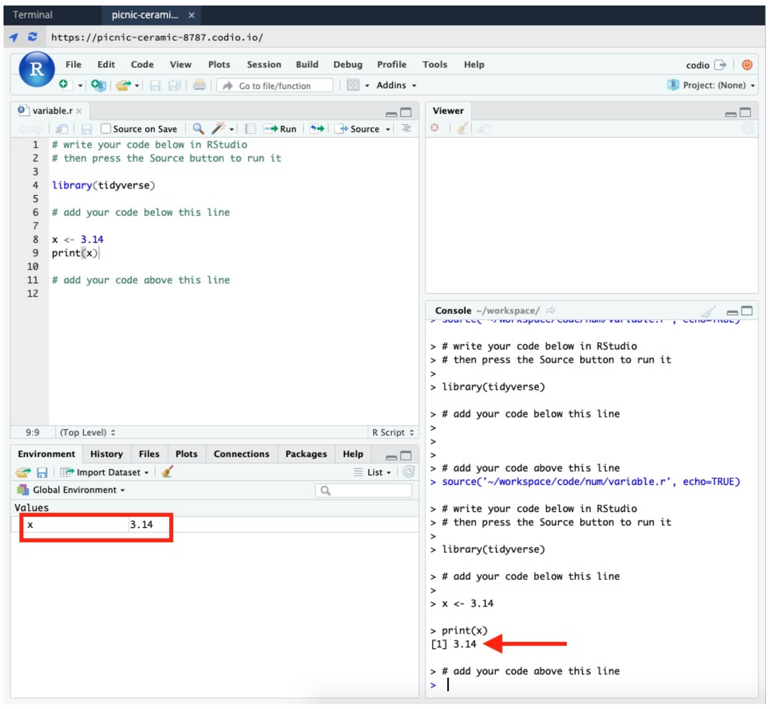
What Are Variables?

Variables in R are similar to variables in algebra where a variable represents some sort of data. In algebra, variables typically only store numbers. However, in R, you can store any kind of data into a variable. To do so, create a variable such as `x` followed by the symbols `<-` (also known as the **assignment** operator), and finally the data you want to store such as `3.14`. Then, to see what is stored in the variable `x`, you can use the `print()` function.

```
x <- 3.14
print(x)
```

Console Result:

```
[1] 3.14
```



```
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

x <- 3.14
print(x)

# add your code above this line
```

Values

x	3.14

```
> # write your code below in RStudio
> # then press the Source button to run it
>
> library(tidyverse)
>
> # add your code below this line
>
>
>
> # add your code above this line
> source('~/workspace/code/num/variable.r', echo=TRUE)

> # write your code below in RStudio
> # then press the Source button to run it
>
> library(tidyverse)

> # add your code below this line
>
> x <- 3.14

> print(x)
[1] 3.14

> # add your code above this line
>
```

.guides/img/num/print-variable

When you start assigning a value to a variable, you'll see their connection within the `Environment` tab in the lower left of RStudio. This helps to keep track of all of the variables you are using and the values that are stored in them.

Variable Names

Variables can be labeled with a single character such as `x` or multiple characters such as `area`. You can also include a combination of numbers, underscores, and periods. However, a variable cannot start with an underscore (_) or a number. If it starts with a period (`.`), it cannot be followed up with a number. Here are some examples of acceptable and unacceptable variable names:

```r
# Legal variable names:
myvar <- "John"
my_var <- "John"
myVar <- "John"
MYVAR <- "John"
myvar2 <- "John"
.myvar <- John

# Illegal variable names:
2myvar <- "John"
my-var <- "John"
my var <- "John"
_my_var <- "John"
my_v@ar <- "John"
TRUE <- "John"
```

Source: w3schools.com

Why Are Variables Useful

Variables are great when you're going to reuse a particular value multiple times. For example, if I want to use `3.14` to help me determine the circumference of a circle, I can store it within a variable such as `x`. Then, I can keep using `x` to help me calculate the circumference of several circles with different diameters.

```r
x <- 3.14
print(x * 10)
print(x * 5)
print(x * 18)
```

Console Result:

```
> x <- 3.14

> print(x * 10)
[1] 31.4

> print(x * 5)
[1] 15.7

> print(x * 18)
[1] 56.52
```

Note that some keywords are **reserved** within the R language, meaning you cannot assign values to them. For example, if you try to assign `2` to `TRUE`, you will get an error because `TRUE` is a reserved keyword.

```
TRUE <- 2
```

Console Result:

```
> TRUE <- 2
Error in TRUE <- 2 : invalid (do_set) left-hand side to
        assignment
```

For an additional list of reserved keywords, click here.

Comments

Comments

Comments are lines of code that are ignored by the program. To comment a line of code so that they are ignored by the program, place a # symbol before it. Note that the # symbol tells the program to start ignoring all code **after** it **until** the next line. Comments are great when used as reminders or instructions. For example, the lines `# add your code below this line` and `# add your code above this line` are used as comments to instruct you as the learner to add code to the space in between those commented lines. Add the following lines of code into RStudio and `Source` the program.

```
x <- "Today is " # assigns character to variable x
print(x)

print("Monday")
# print("Tuesday")
# print("Wednesday")
# print("Thursday")
# print("Friday")
```

Console Result:

```
> print(x)
[1] "Today is "

> print("Monday")
[1] "Monday"
```

You'll notice that the program ignores the comment `assigns character to variable x` since # is in front of it. All print statements regarding the days of the week are also ignored with the exception of `"Monday"` since it does not have a # in front of it.

Try removing and adding the # symbols within the code to produce:

```
[1] "Today is "
[1] "Thursday"
```

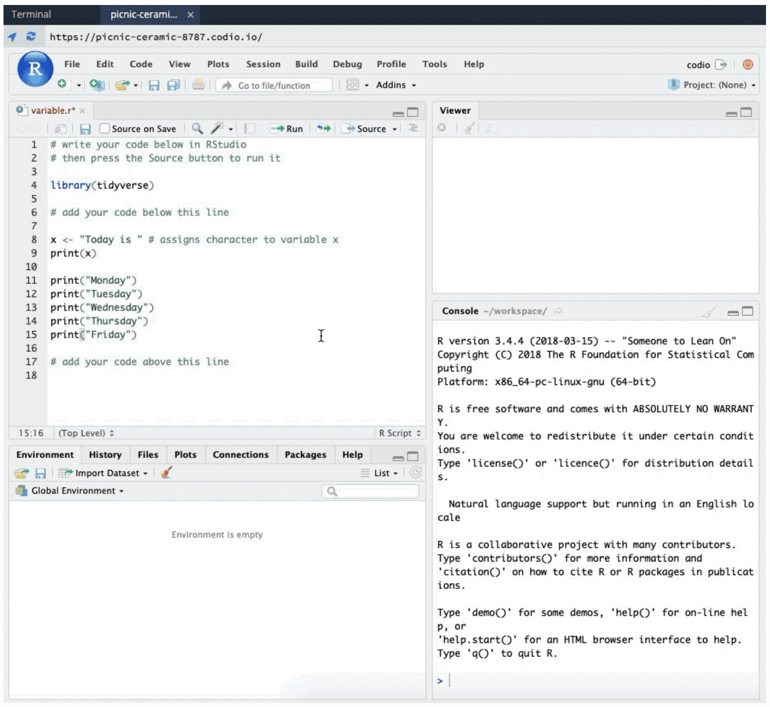
▼ **Sample Solution**

```
x <- "Today is " # assigns character to variable x
print(x)

# print("Monday")
# print("Tuesday")
# print("Wednesday")
print("Thursday")
# print("Friday")
```

By uncommenting `print("Thursday")` and commenting `print("Monday")`, the message `"Today is "` and `"Thursday"` will be printed to the console.

Commenting Multiple Lines of Code

To comment multiple lines of code at once, you can use a combination of shortcut keys such as `Ctrl` + `Shift` + `C` on a PC keyboard or `Command` + `Shift` + `C` on a Mac keyboard. Doing so will add a # to the beginning of every line of code you highlight.



```
# write your code below in RStudio
# then press the Source button to run it

library(tidyverse)

# add your code below this line

x <- "Today is " # assigns character to variable x
print(x)

print("Monday")
print("Tuesday")
print("Wednesday")
print("Thursday")
print("Friday")

# add your code above this line
```

```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Com
puting
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANT
Y.
You are welcome to redistribute it under certain condit
ions.
Type 'license()' or 'licence()' for distribution detail
s.

  Natural language support but running in an English lo
cale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publicat
ions.

Type 'demo()' for some demos, 'help()' for on-line hel
p, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

.guides/img/num/multi-comments.

Arithmetic Operators

You can perform arithmetic operations using certain symbols in R. Here is a list:

- `+` represents addition
- `-` represents subtraction
- `*` represents multiplication
- `/` represents division
- `^` and `**` represent exponent
- `%%` represents modulo
- `%/%` integer division

Note that arithmetic operators **will not** work with the character data type. They will only work with numerics and logicals.

Addition

You can add certain data types together such as numerics using the `+` operator. For example,

```
print(1 + 2)
```

results in

```
[1] 3
```

because 1 + 2 = 3. Additionally, you can also add logicals together. By default, `TRUE` is equivalent to a value of `1` and `FALSE` is equivalent to `0`. For example,

```
print(TRUE + TRUE + FALSE)
```

results in

```
[1] 2
```

because 1 + 1 + 0 = 2. On a final note, you can also add numerics and logicals together. For example,

```
print(3 + TRUE + FALSE + TRUE + 5)
```

results in

```
[1] 10
```

because 3 + 1 + 0 + 1 + 5 = 10.

Subtraction

Like addition, you can subtract numerics and logicals using the `-` operator.

```
print(5 - 2)
print(-6 - 3)
print(FALSE - TRUE)
print(TRUE - -3 - 7)
```

Console Result:

```
[1] 3
[1] -9
[1] -1
[1] -3
```

Multiplication

Examples of the multiplcation `*` operator are shown below.

```
print(5 * 2)
print(-6 * 3)
print(FALSE * TRUE)
print(TRUE * -3 * 7)
```

Console Result:

```
[1] 10
[1] -18
[1] 0
[1] -21
```

Division

Examples of the division `/` operator are shown below.

```
print(5 / 2)
print(-6 / 3)
print(FALSE / TRUE)
print(TRUE / -3 / 7)
print(TRUE / FALSE)
```

Console Result:

```
[1] 2.5
[1] -2
[1] 0
[1] -0.04761905
[1] Inf
```

Note that `Inf` stands for “infinity” or “negative infinity” and is also the result of dividing by zero (`0`).

Exponent

Examples of the exponent `^` and `**` operators are shown below. **Note** that there is no difference between the two operators. You may choose whichever you prefer.

```
print(5 ^ 2)
print(-6 ** 3)
print(FALSE ^ TRUE)
print(TRUE ** -3 ^ 7)
```

Console Result:

```
[1] 25
[1] -216
[1] 0
[1] 1
```

Modulo

Modulo (`%%`) is similar to division because both involve dividing data. However, modulo will return the **remainder** of the quotient instead of the quotient itself. For example, 5 divided by 2 is 2 with a remainder of 1. When you perform modulo using `5 %% 2`, you get the remainder of `1` in return.

```
print(5 %% 2)
print(-6 %% 3)
print(-1 %% -3)
print(24 %% 5 %% 5)
```

Console Result:

```
[1] 1
[1] 0
[1] -1
[1] 4
```

Guide:

```
print(5 %% 2) = 5 divide 2 = 2 remainder 1 = 1
print(-6 %% 3) = -6 divide 3 = -2 remainder 0 = 0
print(-1 %% -3) = -1 divide -3 = 0 remainder -1 = -1
print(24 %% 5 %% 5) =
  24 divide 5 = 4 remainder 4 = 4
  4 divide 5 = 0 remainder 4 = 4
```

Integer Division

When you perform **integer division** using the `%/%` operator, the system will effectively round to the smaller whole number regardless of the decimal values that exist. For example 5 divided by 2 is 2.5 but with integer division, the system will return just 2 instead since 2 is smaller than 3. Again, integer division does not take decimal values into consideration. It will always round **down** to the smaller whole number.

```
print(5 %/% 2)
print(-6 %/% 4)
print(-1 %/% -3)
print(24 %/% 5 %/% 3)
```

Console Result:

```
[1] 2
[1] -2
[1] 0
[1] 1
```

Guide:

```
print(5 %/% 2) = 2.5 = 2
print(-6 %/% 4) = -1.5 = -2 (-2 is smaller than -1)
print(-1 %/% -3) = 0.3333333 = 0
print(24 %/% 5 %/% 3) =
  24 / 5 = 4.8 = 4
  4 / 3 = 1.3333333 = 1
```

PEMDAS

Note that **PEMDAS** rules still apply when using arithmetic operators in R. PEMDAS specifies that operations must be performed from left to right in this order of priority:

- Parentheses `()`
- Exponents `^` and `**`
- Multiplication `*`, division `/`, modulo `%%`, and integer division `%/%`
- Addition `+` and subtraction `-`

Note that **modulo** and **integer division** have the same priority level as multiplication and division. For example,

```
print(2 + 3 * 5 - 7^2 %% 4 + (5 / 2))
```

Console Result:

```
[1] 18.5
```

Guide:

```
print(2 + 3 * 5 - 7^2 %% 4 + (5 / 2))
  parentheses: 5 / 2 = 2.5
  exponent: 7^2 = 49
  multiplication: 3 * 5 = 15
  modulo: 49 %% 4 = 1
  addition: 2 + 15 = 17
  subtraction: 17 - 1 = 16
  addition: 16 + 2.5 = 18.5
```

Logical Operators

In addition to arithmetic operations, you can also perform logical operations using certain symbols in R. Here is a list:

- `<` represents less than
- `<=` represents less than or equal to
- `>` represents greater than
- `>=` represents greater than or equal to
- `==` represents exactly equal to
- `!=` represents not equal to
- `!` represents "not" or negation
- `|` represents "or"
- `&` represents "and"

Logical expressions evaluate as either **TRUE** or **FALSE**.

Less Than

Less than (<) will evaluate as `TRUE` if the left hand side is **smaller** in value than the right hand side. Otherwise, it evaluates as `FALSE`.

For example:

```
print(5 < 7)
```

Prints:

```
[1] TRUE
```

It also works with characters and logicals. For example:

```
print("c" < "b")
print(FALSE < TRUE)
```

Prints:

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

Note that characters that come earlier in the alphabet are considered to be *smaller* or *less than* characters that come later in the alphabet. In addition, the logical `FALSE` is smaller than the logical `TRUE` because `FALSE` equals `0` and `TRUE` equals `1`.

Less Than or Equal To

Less than or equal to (`<=`) will evaluate as `TRUE` if the value on the left hand side is **smaller than or equal to** the value on the right hand side. Otherwise, it evaluates as `FALSE`.

For example:

```
print(5 <= 7)
print(5 <= 5)
print(7 <= 5)
print("x" <= "y")
print("z" <= "a")
```

Prints:

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

Greater Than

Greater than (`>`) will evaluate as `TRUE` if the left hand side is **larger** in value than the right hand side. Otherwise, it evaluates as `FALSE`.

For example:

```
print(5 > 3)
print(5 > 5)
print(7 > 5)
print("x" > "y")
print(TRUE > FALSE)
```

Prints:

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

Greater Than or Equal To

Greater than or equal to (>=) will evaluate as `TRUE` if the value on the left hand side is **larger than or equal to** the value on the right hand side. Otherwise, it evaluates as `FALSE`.

For example:

```
print(5 >= 7)
print(5 >= 5)
print(7 >= 5)
print("x" >= "y")
print("z" >= "a")
print(FALSE >= FALSE)
```

Prints:

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

Equal To

Equal to (==) will evaluate as `TRUE` if the value on the left hand side is **exactly equal to** the value on the right hand side. Otherwise, it evaluates as `FALSE`.

For example:

```
print(5 == 7)
print(5 == 5)
print(0 == FALSE)
print("x" == "y")
print("A" == "a")
print(TRUE == TRUE)
```

Prints:

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

info

NOTE

- `print(0 == FALSE)` evaluates as `TRUE` because `FALSE` has the value of `0`.
- `print("A" == "a")` evaluates as `FALSE` because uppercase and lowercase characters **ARE NOT** considered to be equal in R.

Not Equal To

Not equal to (`!=`) will evaluate as `TRUE` if the value on the left hand side is **not equal to** the value on the right hand side. Otherwise, it evaluates as `FALSE`.

For example:

```
print(5 != 7)
print(5 != 5)
print(0 != FALSE)
print("x" != "y")
print("A" != "a")
print(TRUE != TRUE)
```

Prints:

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

Not

Not or negation (`!`) will return the **opposite** result of the expression.

For example:

```
print(5 < 7)
print(!(5 < 7))
print(FALSE)
print(!(FALSE))
print(TRUE)
```

Prints:

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

info

NOTE

- `print(5 < 7)` is `TRUE` but the negation `!` causes it to become `FALSE`.
- `print(FALSE)` is `FALSE` but the negation `!` causes it to become `TRUE`. Thus, `!(FALSE)` is the same as `TRUE`.

Or

Or (`|`) can only be used to evaluate **more than one** expression. Or (`|`) returns `TRUE` as long as **one of the expressions** evaluates as `TRUE`.

For example:

```
print( (5 > 7) | (5 < 7) )
print( (TRUE == FALSE) | (1 <= 0) )
```

Prints:

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

info

NOTE

- `print( (5 > 7) | (5 < 7) )` is `TRUE` because even though `5 > 7` is `FALSE`, `5 < 7` is `TRUE`. Since there is **at least one** `TRUE` evaluation, `|` causes the whole expression to be evaluated as `TRUE`.
- On the other hand, `print( (TRUE == FALSE) | (1 <= 0) )` evaluates as `FALSE` because both `TRUE == FALSE` and `1 <= 0` evaluate as `FALSE`. Since there is no `TRUE` evaluation, `|` evaluates the whole expression as `FALSE`.

And

Like Or (`|`), And (`&`) can only be used to evaluate **more than one** expression. And (`&`) returns `TRUE` as long as **all of the expressions** evaluate as `TRUE`.

For example:

```
print( (5 > 7) & (5 < 7) )
print( (TRUE == FALSE) & (1 <= 0) )
print( (4 < 5) & (9 >= 9) & (1 == TRUE) )
```

Prints:

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

info

NOTE

- Since `5 > 7` evaluates as `FALSE`, `&` causes the whole expression to be `FALSE`.
- `TRUE == FALSE` is `FALSE` making the whole expression `FALSE` due to `&`.
- Since each of the three expressions `(4 < 5)`, `(9 >= 9)`, and `(1 == TRUE)` evaluates as `TRUE`, `&` will cause the whole expression to be `TRUE`.

Note how arithmetic operations will not work with character values; however, logical operations can.

Order of Logical Expression

Similar to PEMDAS for arithmetic expressions, there is also a priority hierarchy for logical expressions. This order is shown below:

- Comparisons `<`, `>`, `<=`, `>=`, `==`, and `!=`
- Not `!`
- And `&`
- Or `|`

This means that comparisons are evaluated first from left to right, then `!`, then `&`, and finally `|`. Given the following example,

```
print((3 < 4) | (TRUE) & (5 >= 5) | !(TRUE))
```

The system will evaluate it in this order:
* `(3 < 4)` —> FALSE
* `(5 >= 5)` —> TRUE
* `!(TRUE)` —> FALSE
* `(TRUE) & (TRUE)` —> TRUE
* `FALSE | TRUE | FALSE` —> TRUE (Since there's at least 1 `TRUE` statement)

Console Result:

```
[1] TRUE
```