Interested in **a verified certificate or transfer credit and accreditation**?

CS50's Introduction to Programming with R

OpenCourseWare

Donate

Carter Zenke
carter@cs50.harvard.edu

David J. Malan
malan@harvard.edu

Menu

Lecture 1

Welcome!

- Welcome to CS50's Introduction to Programming with R!
- *Programming* is a way by which we can communicate instructions to a computer.
- There are many *programming languages* that one can use to program, including *C*, *Python*, *Java*, *R*, and on!
- We can use R to answer questions dealing with data, such as modeling how COVID-19 was spread on a cruise ship. R can also be used to visualize answers to those questions.

IDE

- An *IDE* is an Integrated Development Environment, which is a pre-configured set of tools that can be used to program.
- R has its own IDE called *RStudio*, which is used to exclusively program R.
- In RStudio, notice the `>` symbol. This denotes the R console, where we can issue commands.

Creating Your First Program

- You can create your first program by typing `file.create("hello.R")` in the R console and hitting the `enter` or `return` key on your keyboard.
- Notice that `hello.R` ends in `.R`. You might have seen other files with a `.jpg` or `.gif` extension in the past. `.R` is the specific file extension used by R.
- When you issued the command above, you should see `[1] TRUE` in the R console. More on that later!
- To the right of the R console, you can access the *file explorer*. Notice how `hello.R` is created in our *working directory*—the place where all our files will be saved by default.
- We can open our `hello.R` file by double-clicking it.
- The *file editor* will now appear, a place where we can write many lines of code.
- In the file editor, type out your first program as follows:

  ```
  print("hello, world")
  ```

  Notice all the text and characters that appear here. They are all necessary.
- You can save by clicking on the *save* icon.
- You may be used to running programs by double-clicking an icon. Within R, we have to take a different approach to run our program.
- R is more than a programming language. It is also an interpreter that changes our *source code* into something the computer understands and can run.
- We can execute this process by clicking the *run* button. Notice how `hello, world` is now displayed. Well done!

Functions

- *Functions* are a way by which we can run a set of instructions.
- In your code, `print` is a function to which `"hello world"` is passed. What we pass to a function we call an `argument`.
- The side effect of this function is that `hello, world` is displayed in the R console.

Bugs

- *Bugs* are unintentional mistakes that can manifest in one's code.
- Modify your code as follows:

  ```
  # Demonstrates a bug

  prin("hello, world")
  ```

  Notice the missing `t` in `prin`.
- Running your code, you will notice how an error is produced.
- *Debugging* is the process of finding and eliminating bugs.

`readline`

- Within R, the function `readline` can read input from the user.
- Modify your code as follows:

  ```
  readline("What's your name? ")
  print("Hello, Carter")
  ```

  Notice how `Carter` will always appear if we run this code.
- We need to create a way by which we can read and use what is given by the user as a name.
- Functions don't just have arguments and side effects. They also have *return values*. Return values are provided back by functions. We can store returned values as *variables*. In R, *variables* can also be called *objects* to avoid confusion with statistical variables—a different concept!
- Modify your code as follows:

  ```
  name <- readline("What's your name? ")
  print("Hello, name")
  ```

  Notice how the variable called `name` stores the return value of `readline`. The arrow `<-`

indicates that the return value is traveling from `readline` to `name`. This arrow is called the *assignment operator*.

- Running this code and opening the *environment* window on the right of our IDE, you can see the variables that are within your program and what is stored within them.

`paste`

- Still, running this code, notice how "name" always appears. This is clearly a bug!
- We can correct this bug as follows:

```
name <- readline("What's your name? ")
greeting <- paste("Hello, ", name)
print(greeting)
```

  Notice how the first line of code remains unchanged. Notice how we create a new variable called `greeting` and assign the *string concatenation* of "Hello, " and "name" together to `greeting`. *Strings* are a set of characters. Two separate strings are combined into one using the `paste` function. The resulting variable, `greeting` is printed using the `print` function.

- Running this code, notice the new variable that appears in the environment.
- If you are being particularly observant, there is still a bug! Two spaces are stored in greeting, between "Hello," and the value of `name`.

Documentation

- The *documentation* for `paste` can be accessed by typing `?paste` in the R console. Accordingly, the documentation for `paste` will appear. Reading this documentation, one can learn the various *parameters* one can use with `paste`.
- One parameter relevant to our current work is `sep`.
- Modify your code as follows:

```
name <- readline("What's your name? ")
greeting <- paste("Hello, ", name, sep = "")
print(greeting)
```

  Notice how `sep = ""` is added to the code.

- Running this program, you will see the output now works as intended.
- It just so happens that programmers have often had a need to omit these extra spaces by setting `sep` equal to `""`. Thus, they invented `paste0`, which concatenates strings without any separating characters. `paste0` can be used as follows:

```
name <- readline("What's your name? ")
greeting <- paste0("Hello, ", name)
```

```
print(greeting)
```

Notice how `paste` becomes `paste0`.

- Your program can be further simplified as follows:

```
# Ask user for name
name <- readline("What's your name? ")

# Say hello to user
print(paste("Hello,", name))
```

Notice how `greeting` is eliminated by directly passing the `paste` return value as the input value of `print`.

- In the end, when nesting functions within functions as above, do consider how you and others may be further challenged in reading your code. Sometimes, too much nesting can result in not being able to understand what the code is doing. This is a *design* decision. That is, you will often make decisions about your code to benefit both your users and programmers.
- Further, a *style* decision you might make is to include comments using the `#` symbol, where you describe what a section of code is doing.

Arithmetic

- Let's create a new program that will count votes for some fictional characters.
- Close the `hello.R` file.
- In your console, type `file.create("count.R")`.
- Create your code as follows:

```
mario <- readline("Enter votes for Mario: ")
peach <- readline("Enter votes for Peach: ")
bowser <- readline("Enter votes for Bowser: ")

total <- mario + peach + bowser

print(paste("Total votes:", total))
```

Notice how the return values of `readline` are stored in three variables called `mario`, `peach`, and `bowser`. The variable `total` is assigned the values of `mario`, `peach`, and `bowser` added together. Then, the total is printed.

- R has many arithmetic operators, including `+`, `-`, `*`, `/`, and others!
- Running this code, and typing in the number of votes, produces an error.
- It just so happens that input from the user is treated as a string instead of a number. Looking at the environment, notice how the values for `mario` and others are stored with quotation marks around them. These quotes indicate that these are being stored as character strings instead of numbers. These values need to be numbers to be added

together with `+`.

- In R, there are different modes (sometimes also called "types"!) as which a variable can be stored. Some of these "storage modes" include character, double, and integer.
- We can convert these variables to the storage mode we want as follows:

```
mario <- readline("Enter votes for Mario: ")
peach <- readline("Enter votes for Peach: ")
bowser <- readline("Enter votes for Bowser: ")

mario <- as.integer(mario)
peach <- as.integer(peach)
bowser <- as.integer(bowser)

total <- mario + peach + bowser

print(paste("Total votes:", total))
```

  Notice how *coercion* is employed through `as.integer` to convert `mario` and others to integers.
- Running this code and looking at the environment, you can see how these values are now being stored as integers without quotation marks.
- This program can be further simplified as follows:

```
mario <- as.integer(readline("Enter votes for Mario: "))
peach <- as.integer(readline("Enter votes for Peach: "))
bowser <- as.integer(readline("Enter votes for Bowser: "))

total <- sum(mario, peach, bowser)

print(paste("Total votes:", total))
```

  Notice how the `sum` function is employed to sum the values of the three variables.
- Could there be a way by which we can utilize a pre-existing source of data?

Tables

- *Tables* are one of the many structures we can use to organize data.
- A *table* is a set of rows and columns, where rows often represent some entity being stored, and columns represent attributes of each of those entities.
- Tables can be stored in a variety of file formats. One common format is a comma-separated values (CSV) file.
- In CSV files, each row is stored on a separate line. Columns are separated by commas.
- Before we begin our next program, type `ls()` in the R console to determine all the variables that are active in your environment. Then, type `rm(list = ls())` to remove all those values from your environment. Typing `ls()` again, you'll notice that there are no objects left in your environment.

- Next, type `file.create("tabulate.R")` to create our new program file. Opening your file explorer, open the `tabulate.R` file. Additionally, you should download the `votes.csv` file from this lecture's source code and drag it into your working directory.
- Create your code as follows:

  ```
  votes <- read.table("votes.csv")
  View(votes)
  ```

  Notice how the first line of code reads the table from `votes.csv` into the `votes` variable. Then, `View` allows you to view what was stored in `votes`.
- Running this code, you can now see a separate tab of what is stored in the `votes` object. However, there is an error. Notice how all data has been read into one column. It would seem that `read.table` is reading the data from the `csv` file. But, there seems to be some formatting that is still needed.
- Modify your code as follows:

  ```
  votes <- read.table(
    "votes.csv",
    sep = ","
  )
  View(votes)
  ```

  Notice how `sep` is used to tell `read.table` on which character each column will separate.
- Still, running this code, there is an error. How can we have `read.table` recognize the header of the table?
- Modify your code as follows:

  ```
  votes <- read.table(
    "votes.csv",
    sep = ",",
    header = TRUE
  )
  View(votes)
  ```

  Notice how the `header = TRUE` argument allows `read.table` to recognize that there is a header.
- Running this file, the table displays as intended.
- Programmers have created a shortcut to be able to do this more simply. Modify your code as follows:

  ```
  votes <- read.csv("votes.csv")
  View(votes)
  ```

  Notice how `read.csv` accomplishes with far greater simplicity what the previous code did!

- Now that our data is loaded, how can we access it? Modify your code as follows:

  ```
  votes <- read.csv("votes.csv")

  votes[, 1]
  votes[, 2]
  votes[, 3]
  ```

  Notice how *bracket notation* is used to access values using a `votes[row, column]` format. Thus, `votes[, 2]` will display the numbers in the `poll` column.

Vectors

- *Vectors* are a list of values all of the same storage mode.
- Considering our data frame (or table) of candidates and votes, we can access specific values by creating a new vector.
- We can simplify this program by calling the precise name of each column:

  ```
  votes <- read.csv("votes.csv")

  colnames(votes)

  votes$candidate
  votes$poll
  votes$mail
  ```

  Notice how `votes$poll` returns a vector of all the values within the `poll` column. We can now access the values of the `poll` column with this new vector.
- Running this code, notice how the values of each column appear.
- Turning to our original question about how to sum these values, modify your code as follows:

  ```
  votes <- read.csv("votes.csv")

  sum(votes$poll[1], votes$poll[2], votes$poll[3])
  ```

  Notice how `sum` is employed to sum the values in the first, second, and third rows of poll.
- However, this code is not dynamic. It's quite inflexible. What if there were more than three candidates? Hence, we can simplify our code as follows to be more dynamic:

  ```
  votes <- read.csv("votes.csv")

  sum(votes$poll)
  sum(votes$mail)
  ```

  Notice how the values found in the vectors `votes$poll` and `votes$mail` are summed.
- As illustrated above using bracket notation, we could also try to sum the values in each

row across the `poll` and `mail` columns. Modify your code as follows:

```
votes <- read.csv("votes.csv")

votes$poll[1] + votes$mail[1]
votes$poll[2] + votes$mail[2]
votes$poll[3] + votes$mail[3]
```

Notice how each row for `poll` and `mail` is added together.

- Is this is the best approach R offers, though?

Vector Arithmetic

- There are many times when we want to be able to add the rows of one vector with the rows of another vector. We can do this through vector arithmetic.
- In the same spirit of making our code more dynamic, we can further modify our code as follows:

```
votes <- read.csv("votes.csv")

votes$poll + votes$mail
```

Notice how the vectors are added *element-wise*. That is, the first row of the first vector is added to the first row of the second vector, the second row of the first vector is added to the second row of the second vector, and so on. This results in a final vector with the same number of rows as the `poll` and `mail` vectors.

- Vector arithmetic results in an entirely new vector. We can work with this new vector in a whole host of ways.
- Naturally, we may want to store the result of our arithmetic. We can do so by modifying our code as follows:

```
votes <- read.csv("votes.csv")

votes$total <- votes$poll + votes$mail

write.csv(votes, "totals.csv")
```

Notice how the final total is stored in a new vector called `votes$total`, which in fact is a new `total` column of the `votes` data frame. We then write the resulting `votes` data frame to a file called `totals.csv`.

- An issue arises when you look at the `csv` file. Notice that, by default, "row names" are included. These can be excluded by modifying your code as follows:

```
votes <- read.csv("votes.csv")

votes$total <- votes$poll + votes$mail
```

```
write.csv(votes, "totals.csv", row.names = FALSE)
```

Notice how `row.names` is set to `FALSE`.

External Data

- Today, we have seen many examples about how to use R.
- There are many instances where you may wish to use someone else's dataset.
- You can access data from an online sources as follows:

  ```
  # Demonstrates reading data from a URL

  url <- "https://github.com/fivethirtyeight/data/raw/master/non-voters/nonvoter
  voters <- read.csv(url)
  ```

  Notice how `read.csv` is pulling data from a defined URL.

- Looking at this data frame, you can run `nrow` to get the number of rows. You can run `ncol` to get the number of columns.

  ```
  # Demonstrates finding number of rows and columns in a large data set

  url <- "https://github.com/fivethirtyeight/data/raw/master/non-voters/nonvoter
  voters <- read.csv(url)

  nrow(voters)
  ncol(voters)
  ```

  Notice how the `nrow` and `ncol` are used to determine how many rows and columns exist in this data.

- Datasets sometimes come with a *code book*. A code book is a guide to what columns are included in this data. For example, column `Q1` may represent a specific question asked of participants in a study. By looking at this data set's code book, we can tell there is a column called `voter_category` that defines a specific voting behavior for each participant.
- You might want to understand what were the various options that could have been selected by participants in this column. This can be accomplished through the `unique` function.

  ```
  # Demonstrates finding unique values in a vector

  url <- "https://github.com/fivethirtyeight/data/raw/master/non-voters/nonvoter
  voters <- read.csv(url)

  unique(voters$voter_category)
  ```

Notice how `unique` is used to determine the possible options participants could have selected.

Special Values

- For `Q22`, we discover in the code book that this question deals with why participants are *not* registered to vote. Looking at this data, we see `NA` as one of the values presented. `NA` represents "not available" as a special value within R.
- Other special values in R include `Inf`, `-Inf`, `NaN`, and `NULL`. Respectively, these mean infinite, negatively infinite, not a number, and null (or none) value.
- To see these possible values for `Q22`, we can run the following code:

```
# Demonstrates NA

url <- "https://github.com/fivethirtyeight/data/raw/master/non-voters/nonvoter
voters <- read.csv(url)

voters$Q22
unique(voters$Q22)
```

Notice how `unique` is employed again to discover the possible values for `Q22`.

`factor`

- `Q21` deals with participants' plans to vote in a future election. In this column, a value of `1`, `2`, and `3`, coincided with specific possible answers. For example, `1` might represent "Yes".
- In R, we can use `factor` to convert the numbered values to specific text-based answers. For example, we can use `factor` to change the number `1` to correspond to the text "Yes". We can accomplish this by modifying our code as follows:

```
# Demonstrates converting a vector to a factor

url <- "https://github.com/fivethirtyeight/data/raw/master/non-voters/nonvoter
voters <- read.csv(url)

voters$Q21

factor(
  voters$Q21
)

factor(
  voters$Q21,
  labels = c("?", "Yes", "No", "Unsure/Undecided")
)
```

Notice how `factor(voters$Q21)` will show the specific levels (categories) of data for `Q21`. In the `factor` that appears later in the code, labels are applied to each level. `1`, for example, is associated with "Yes".

- There are many instances in which we may want to exclude values. In `Q21`, we may wish to exclude `-1`, since it's not clear what this value represents. We can do so as follows:

  ```
  # Demonstrates excluding values from the levels of a factor

  url <- "https://github.com/fivethirtyeight/data/raw/master/non-voters/nonvoter
  voters <- read.csv(url)

  voters$Q21 <- factor(
    voters$Q21,
    labels = c("Yes", "No", "Unsure/Undecided"),
    exclude = c(-1)
  )
  ```

  Notice how `-1` is excluded.

Summing Up

In this lesson, you learned how to represent data in R. Specifically, you learned...

- Functions
- Bugs
- `readline`
- `paste`
- Documentation
- Arithmetic
- Tables
- Vectors
- Vector arithmetic
- External data
- Special values
- `factor`

See you next time when we discuss how to transform data.