CS50's Introduction to Programming with R

OpenCourseWare

Carter Zenke
carter@cs50.harvard.edu

David J. Malan
malan@harvard.edu

Lecture 4

Welcome!

- Welcome back to CS50's Introduction to Programming with R!
- Today, we will be learning about tidying data. Indeed, you can imagine many times when tables and data may not be in the shape one would hope!
- *Packages* are bits of code created by developers that we can install and load into our R programs. These packages can give one functionality within R that does not come natively.
- Packages are stored in R's *library*. As such, you can load packages with the `library` function.

dplyr

- dplyr is a package within the tidyverse that includes functions to manipulate data.
- Within dplyr, a data set called `storms` is included, which includes observations of storm data from NOAA, the United States' National Oceanic and Atmospheric Administration.
- After loading dplyr or the tidyverse, the `storms` data set can be loaded by simply typing `storms` in the R console.
- Upon typing `storms` notice that a *tibble* is displayed. A *tibble* is tidyverse's "reimagining" of R's data frame. Notice how rows, row numbers, and various columns are included and labeled. Further, notice the text color that is employed in the *tibble*.

`select`

- Let's locate the strongest storm in the data set. First, let's remove the columns we don't need. Consider the following program:

  ```
  # Remove selected columns

  dplyr::select(
    storms,
    !c(lat, long, pressure, tropicalstorm_force_diameter, hurricane_force_diamet
  )
  ```

  Notice how the `select` function within dplyr allows one to determine which columns will be included in a data frame or tibble. `select`'s first argument is the data frame (or tibble) to operate on: `storms`. `select`'s second argument is the vector of columns to be selected. In this case, however, a `!` is employed: a `!` indicates that the proceeding column names are instead to be excluded. Alternatively, a `-` has the same functionality. Running this code will simplify the tibble by removing the above columns removed.
- Typing out all these columns is a bit cumbersome!
- Helper functions like `contains`, `starts_with`, or `ends_with` can help with this. Consider the following code:

```
# Introduce ends_with

select(
  storms,
  !c(lat, long, pressure, ends_with("diameter"))
)
```

Notice how `ends_with` is employed to exclude all columns that end with *diameter*. Less code is employed, but the result is the same as before.

`filter`

- Another helpful function is `filter`, which can be used to filter rows from the data frame.
- Consider the following code:

```
# Find only rows about hurricanes

filter(
  select(
    storms,
    !c(lat, long, pressure, ends_with("diameter"))
  ),
  status == "hurricane"
)
```

  Notice how the only rows included are those that include `hurricane` in the `status` column.
- Notice how the latest examples have dropped the `dplyr::` syntax in the first example. Turns out you don't need to name the specific package in which a function is defined, unless two or more packages define a function with the same name. In that case, you'll need to remove ambiguity by specifying which package's function you want to use.

Pipe Operator

- In R, the *pipe operator* is signified by `|>`, which allows one to "pipe" data into a specific function. For example, consider the following code:

```
# Introduce pipe operator

storms |>
  select(!c(lat, long, pressure, ends_with("diameter"))) |>
  filter(status == "hurricane")
```

  Notice how `storms` is piped to `select`, implicitly becoming `select`'s first argument. Then, notice how the return value of `select` is piped to `filter`, implicitly becoming `filter`'s first argument. When you use the pipe operator, you can avoid nesting function calls and write your code more sequentially.

`arrange`

- Now let's use the `arrange` function to sort our rows:

  ```
  # Find only rows about hurricanes, and arrange highest wind speed to least

  storms |>
    select(!c(lat, long, pressure, ends_with("force_diameter"))) |>
    filter(status == "hurricane") |>
    arrange(desc(wind))
  ```

  Notice how the return value of the `select` function is piped to `filter`, the return value of which is then piped to `arrange`. The rows in the resulting data frame are arranged in descending order by value of the `wind` column.

`distinct`

- You may notice that this tibble includes many rows of the same storm. Because this data includes many observations of the same storms, this is not a surprise. However, would it not be nice to be able to find only *distinct* storms?
- The `distinct` function allows one to get back distinct items in our tibble.
- Distinct returns distinct rows finding duplicate rows and returning the first row from the set of duplicates.
- By default, `distinct` will consider rows to be duplicate only if *all* values in a row match *all* values in another row.
- However, you can tell `distinct` which values to consider when determining whether rows are duplicates. Consider the following code that leverages this ability:

  ```
  # Keep only first observation about each hurricane

  storms |>
    select(!c(lat, long, pressure, ends_with("force_diameter"))) |>
    filter(status == "hurricane") |>
    arrange(desc(wind), name) |>
    distinct(name, year, .keep_all = TRUE)
  ```

  Notice that `distinct` is told to only look at the `name` and `year` of each storm to determine if it is a distinct item. `.keep_all = TRUE` tells `distinct` to still return all the columns for each row.

Writing Data

- It's possible for us to save our data for later in a CSV file.
- Consider the following code:

```r
# Write subset of columns to a CSV

hurricanes <- storms |>
  select(!c(lat, long, pressure, ends_with("force_diameter"))) |>
  filter(status == "hurricane") |>
  arrange(desc(wind), name) |>
  distinct(name, year, .keep_all = TRUE)

hurricanes |>
  select(c(year, name, wind)) |>
  write.csv("hurricanes.csv", row.names = FALSE)
```

Notice how the result of the first block of code is stored as `hurricanes`. To store `hurricanes` as a CSV file, `select` first choose 3 particular columns (`year`, `name`, and `wind`) which are written to a file named `hurricanes.csv`.

`group_by`

- Let's now find the most powerful hurricane in each year.
- Consider the following code:

```r
# Find most powerful hurricane for each year

hurricanes <- read.csv("hurricanes.csv")

hurricanes |>
  group_by(year) |>
  arrange(desc(wind)) |>
  slice_head()
```

Notice how `hurricanes.csv` is read into `hurricanes`. Then, the function `group_by` is employed to group together all hurricanes in each year. For each group, the group is arranged in descending order by `wind` using `arrange(desc(wind))`. Finally, `slice_head` is used to output the top row from each group. Thus, the strongest storm from each year is presented.

- `slice_max` selects the largest values within a variable. Consider how this can be employed in our code:

```r
# Introduce slice_max

hurricanes <- read.csv("hurricanes.csv")

hurricanes |>
  group_by(year) |>
  slice_max(order_by = wind)
```

Notice that `hurricanes` is grouped by `year`. Then, the highest value of `wind` is presented using `slice_max`. Doing so eliminates the need for `arrange(desc(wind))`.

`summarize`

- What if we wanted to know the number of hurricanes each year? Consider the following code:

  ```
  # Find number of hurricanes per year

  hurricanes <- read.csv("hurricanes.csv")

  hurricanes |>
    group_by(year) |>
    summarize(hurricanes = n())
  ```

  Notice how the function `summarize`, employing `n`, counts the number of rows in each group.

`ungroup`

- Looking at our `hurricanes` data frame, you will notice that there are groups present. Indeed, these groups are by `year`. There will be times in future activities where you may wish to ungroup items within your data. Accordingly, consider the following:

  ```
  # Show ungroup

  hurricanes <- read.csv("hurricanes.csv")

  hurricanes |>
    group_by(year) |>
    slice_max(order_by = wind) |>
    ungroup()
  ```

  Notice that the `ungroup` command is employed to remove the groups of the tibble.

tidyr

- dplyr is quite useful when data is already well organized.
- What about situations where the data is not already well organized?
- For that, the tidyr package can be useful!

Tidy Data

- According to the philosophy of the tidyverse, there are three principles that guide what we would call *tidy data*.

  ```
  1. Each observation is a row; each row is an observation.
  2. Each variable is a column; each column is a variable.
  3. Each value is a cell; each cell is a single value.
  ```

- When evaluating data, best to look at the above three principles to see if they are observed.

Normalizing

- *Normalizing* is the process of converting data such that they fulfill the aforementioned principles.
- Normalizing can also refer to converting data such that they fulfill better design principles outside the above guidelines.
- Download the `students.csv` file from the course files and place it in your working directory. Create new code as follows:

  ```
  # Read CSV

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

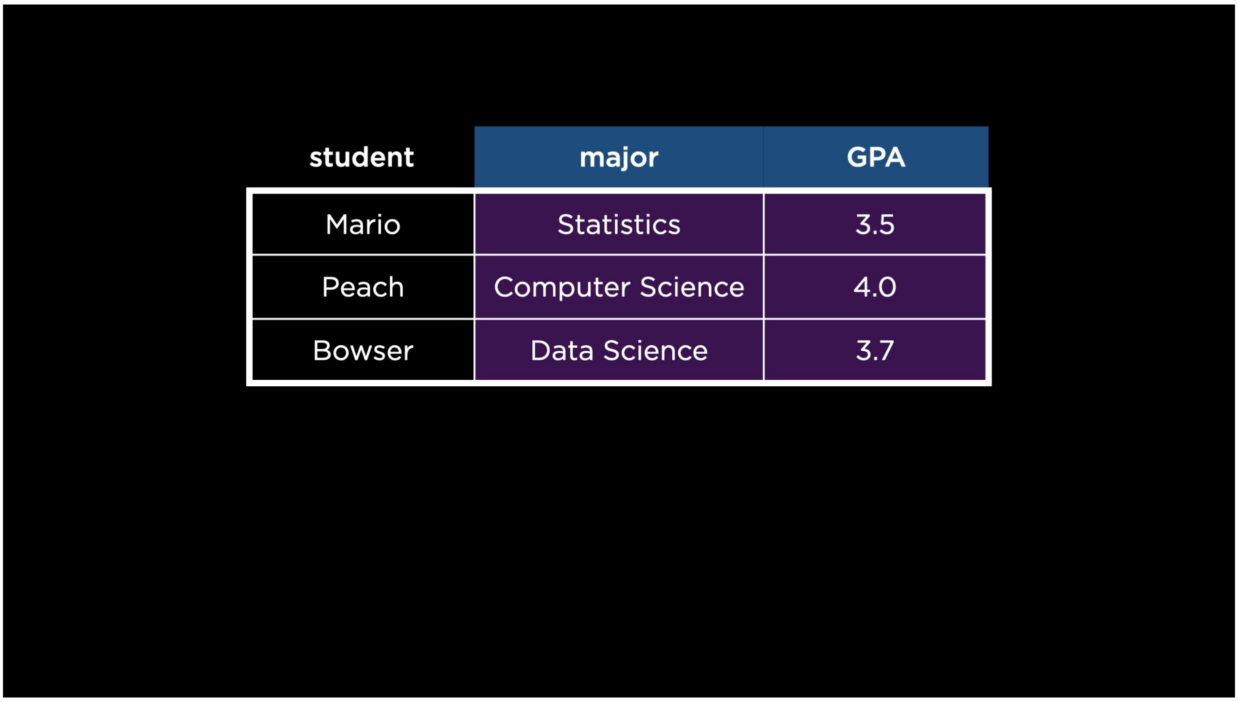
  Notice that this code loads a CSV file called `students.csv` and stores these values in `students`.
- Examining this data, you may see how they do not follow the principles we mentioned previously. Which principles do you observe not being followed?

Pivoting

- In the `students` data set, you might notice there are row values that should instead be column names: "major" and "GPA." To be clear, this data set violates the second principle of tidy data: each way a student can vary is *not* a column.
- We can *pivot* the data set to turn those variables into columns, thanks to `pivot_wider`! `pivot_wider` transforms a data set that is "longer" than it should be (i.e., one with variables as row values) and makes it "wider" (i.e., turns those variables into columns).
- `pivot_wider` will transform the `students` data set from the below:

student	attribute	value
Mario	major	Statistics
Mario	GPA	3.5
Peach	major	Computer Science
Peach	GPA	4.0
Bowser	major	Data Science
Bowser	GPA	3.7

into the following:



student	major	GPA
Mario	Statistics	3.5
Peach	Computer Science	4.0
Bowser	Data Science	3.7

- But how? Consider the following usage:

```
# Demonstrates pivot_wider

students <- read.csv("students.csv")

students <- pivot_wider(
  students,
  id_cols = student,
  names_from = attribute,
  values_from = value
)
```

Notice how `pivot_wider` takes several arguments, explained here:

- The first is the data set to operate on, `students`.
- The second argument, `id_cols`, specifies which column should ultimately be unique in the transformed data set. Notice how, before `pivot_wider`'s transformation, there are duplicate values in the `student` column. After `pivot_wider`'s transformation, there are unique values in the `student` column.
- The third argument, `names_from`, specifies which row contains values that should instead be variables (columns). Notice how the values in the `attribute` column become columns themselves after `pivot_wider`'s transformation.
- Finally, the fourth argument, `values_from`, specifies the column from which to populate the values of the new columns. Notice how the values in the `value` column are used to populate the new columns after `pivot_wider`'s transformation.

- Because our data is so much more tidy, we can do so much more with the data!
- Consider the following:

```
# Demonstrates calculating average GPA by major

students <- read.csv("students.csv")

students <- pivot_wider(
  students,
  id_cols = student,
  names_from = attribute,
  values_from = value
)

students$GPA <- as.numeric(students$GPA)

students |>
  group_by(major) |>
  summarize(GPA = mean(GPA))
```

Notice how this program leverages `pivot_wider` and tidyr to discover the average GPA of the students. `GPA` in `students` is converted to a numeric value. Then, pipe syntax is used to find the mean of the GPAs.

stringr

- The process we described above works well when the values themselves are clean. However, what about when the values themselves aren't tidy?
- `stringr` offers us a means by which to tidy strings. Download `shows.csv` from the course files and place this file in your working directory. Consider the following program:

```
# Tally votes for favorite shows

shows <- read.csv("shows.csv")

shows |>
  group_by(show) |>
  summarize(votes = n()) |>
  ungroup() |>
  arrange(desc(votes))
```

Notice how shows are grouped by `show`. Then, the number of `votes` is computed. Finally, the `votes` are sorted in descending order.

- Looking at the result of this program, you can see that there are many versions of *Avatar: The Last Airbender*. We should probably address the whitespace issues first.

```
# Clean up inner whitespace

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

```
shows$show <- shows$show |>
  str_trim() |>
  str_squish()

shows |>
  group_by(show) |>
  summarize(votes = n()) |>
  ungroup() |>
  arrange(desc(votes))
```

Notice how `str_trim` is used to remove whitespace in the front or end of each record. `str_squish` is then used to remove extra whitespace *between* the characters.

- While all this is very good, there are still some inconsistencies with capitalization. We can resolve as follows:

```
# Clean up capitalization

shows <- read.csv("shows.csv")

shows$show <- shows$show |>
  str_trim() |>
  str_squish() |>
  str_to_title()

shows |>
  group_by(show) |>
  summarize(votes = n()) |>
  ungroup() |>
  arrange(desc(votes))
```

Notice how `str_to_title` is used to force title casing on each string.

- Finally, we can address spelling variants of *Avatar: The Last Airbender*:

```
# Clean up spelling

shows <- read.csv("shows.csv")

shows$show <- shows$show |>
  str_trim() |>
  str_squish() |>
  str_to_title()

shows$show[str_detect(shows$show, "Avatar")] <- "Avatar: The Last Airbender"

shows |>
  group_by(show) |>
  summarize(votes = n()) |>
  ungroup() |>
  arrange(desc(votes))
```

Notice how `str_detect` is used to locate instances of `Avatar`. Each of these is converted to `Avatar: The Last Airbender`.

- While these tools can be quite helpful, consider cases where you may need to employ caution and not overwrite correct entries. For example, there are many movies called *Avatar*! How do we know whether voters didn't mean to vote for those movies?

Summing Up

In this lesson, you learned how to tidy data in R. Specifically, you learned three new packages, which are each part of the tidyverse:

- dplyr
- tidyr
- string

See you next time when we discuss how to visualize our data.