Exploring Data Frames

- Add and remove rows or columns.
- Append two data frames.
- Display basic properties of data frames including size and class of the columns, names, and first few rows.

- How can I manipulate a data frame?

At this point, you've seen it all: in the last lesson, we toured all the basic data types and data structures in R. Everything you do will be a manipulation of those tools. But most of the time, the star of the show is the data frame—the table that we created by loading information from a csv file. In this lesson, we'll learn a few more things about working with data frames.

Adding columns and rows in data frames

We already learned that the columns of a data frame are vectors, so that our data are consistent in type throughout the columns. As such, if we want to add a new column, we can start by making a new vector:

```
age <- c(2, 3, 5)
cats
```

```
##     coat weight likes_string
## 1 calico    2.1            1
## 2  black    5.0            0
## 3  tabby    3.2            1
```

We can then add this as a column via:

```
cbind(cats, age)
```

```
##     coat weight likes_string age
## 1 calico    2.1            1   2
## 2  black    5.0            0   3
## 3  tabby    3.2            1   5
```

Note that if we tried to add a vector of ages with a different number of entries than the number of rows in the data frame, it would fail:

```
age <- c(2, 3, 5, 12)
cbind(cats, age)
```

```
## Error in data.frame(..., check.names = FALSE): arguments imply differing number of rows: 3, 4
```

```
age <- c(2, 3)
cbind(cats, age)
```

```
## Error in data.frame(..., check.names = FALSE): arguments imply differing number of rows: 3, 2
```

Why didn't this work? Of course, R wants to see one element in our new column for every row in the table:

```
nrow(cats)
```

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

```
length(age)
```

```
## [1] 2
```

So for it to work we need to have `nrow(cats)` = `length(age)`. Let's overwrite the content of cats with our new data frame.

```
age <- c(2, 3, 5)
cats <- cbind(cats, age)
```

Now how about adding rows? We already know that the rows of a data frame are lists:

```
newRow <- list("tortoiseshell", 3.3, TRUE, 9)
cats <- rbind(cats, newRow)
```

Let's confirm that our new row was added correctly.

```
cats
```

```
##            coat weight likes_string age
## 1        calico    2.1            1   2
## 2         black    5.0            0   3
## 3         tabby    3.2            1   5
## 4 tortoiseshell    3.3            1   9
```

Removing rows

We now know how to add rows and columns to our data frame in R. Now let's learn to remove rows.

```
cats
```

```
##            coat weight likes_string age
## 1        calico    2.1            1   2
## 2         black    5.0            0   3
## 3         tabby    3.2            1   5
## 4 tortoiseshell    3.3            1   9
```

We can ask for a data frame minus the last row:

```
cats[-4, ]
```

```
##     coat weight likes_string age
## 1 calico    2.1            1   2
## 2  black    5.0            0   3
## 3  tabby    3.2            1   5
```

Notice the comma with nothing after it to indicate that we want to drop the entire fourth row.

Note: we could also remove several rows at once by putting the row numbers inside of a vector, for example: `cats[c(-3,-4), ]`

Removing columns

We can also remove columns in our data frame. What if we want to remove the column "age". We can remove it in two ways, by variable number or by index.

```
cats[,-4]
```

```
##            coat weight likes_string
## 1        calico    2.1            1
## 2         black    5.0            0
## 3         tabby    3.2            1
## 4 tortoiseshell    3.3            1
```

Notice the comma with nothing before it, indicating we want to keep all of the rows.

Alternatively, we can drop the column by using the index name and the `%in%` operator. The `%in%` operator goes through each element of its left argument, in this case the names of `cats`, and asks, "Does this element occur in the second argument?"

```
drop <- names(cats) %in% c("age")
cats[,!drop]
```

```
##            coat weight likes_string
## 1        calico    2.1            1
## 2         black    5.0            0
## 3         tabby    3.2            1
## 4 tortoiseshell    3.3            1
```

We will cover subsetting with logical operators like `%in%` in more detail in the next episode. See the section Subsetting through other logical operations

Appending to a data frame

The key to remember when adding data to a data frame is that *columns are vectors and rows are lists.* We can also glue two data frames together with `rbind`:

```
cats <- rbind(cats, cats)
cats
```

```
##            coat weight likes_string age
## 1        calico    2.1            1   2
## 2         black    5.0            0   3
## 3         tabby    3.2            1   5
## 4 tortoiseshell    3.3            1   9
## 5        calico    2.1            1   2
## 6         black    5.0            0   3
## 7         tabby    3.2            1   5
## 8 tortoiseshell    3.3            1   9
```

Challenge 1

You can create a new data frame right from within R with the following syntax:

```
df <- data.frame(id = c("a", "b", "c"),
                 x = 1:3,
                 y = c(TRUE, TRUE, FALSE))
```

Make a data frame that holds the following information for yourself:

- first name
- last name
- lucky number

Then use `rbind` to add an entry for the people sitting beside you. Finally, use `cbind` to add a column with each person's answer to the question, "Is it time for coffee break?"

Solution to Challenge 1

```
df <- data.frame(first = c("Grace"),
                 last = c("Hopper"),
                 lucky_number = c(0))
df <- rbind(df, list("Marie", "Curie", 238) )
df <- cbind(df, coffeetime = c(TRUE,TRUE))
```

Realistic example

So far, you have seen the basics of manipulating data frames with our cat data; now let's use those skills to digest a more realistic dataset. Let's read in the `gapminder` dataset that we downloaded previously:

```
library(readr)

eiad <- read_csv("https://github.com/davemcg/2024_04_20_NEI_Vision_Course/raw/main/guides/data/eyeIntegration23_meta_2023_09_01.built.csv.gz")
```

```
## Warning: One or more parsing issues, call `problems()` on your data frame for details,
## e.g.:
##   dat <- vroom(...)
##   problems(dat)
```

```
## Rows: 7136 Columns: 29
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## chr (24): BioSample, run_accession, sample_accession, study_accession, Cohor...
## dbl  (2): Age_Years, Sex_Score
## lgl  (3): Library_Notes, Comment, geo
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Miscellaneous Tips

- Another type of file you might encounter are tab-separated value files (.tsv). To specify a tab as a separator, use `"\\t"` or `read.delim()`.
- Files can also be downloaded directly from the Internet into a local folder of your choice onto your computer using the `download.file` function. The `read_csv` function can then be executed to read the downloaded file from the download location, for example,
- Alternatively, you can also read in files directly into R from the Internet by replacing the file paths with a web address in `read.csv`. One should note that in doing this no local copy of the csv file is first saved onto your computer. For example,

```
eiad <- read.csv("https://hpc.nih.gov/~mcgaugheyd/eyeIntegration/2023/eyeIntegration23_meta_2023_09_01.built.csv.gz")
```

- You can read directly from excel spreadsheets without converting them to plain text first by using the readxl package.
- The argument “stringsAsFactors” can be useful to tell R how to read strings either as factors or as character strings. In R versions after 4.0, all strings are read-in as characters by default, but in earlier versions of R, strings are read-in as factors by default. For more information, see the call-out in the previous episode.

Let’s investigate eiad a bit; the first thing we should always do is check out what the data looks like with `str`:

```
str(eiad)
```

```
## spc_tbl_ [7,136 × 29] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ BioSample             : chr [1:7136] "donor_1_CHO" "donor_1_RPE" "donor_2_CHO" "donor_2_RPE" ...
##  $ run_accession         : chr [1:7136] "donor_1_CHO" "donor_1_RPE" "donor_2_CHO" "donor_2_RPE" ...
##  $ sample_accession      : chr [1:7136] "donor_1_CHO" "donor_1_RPE" "donor_2_CHO" "donor_2_RPE" ...
##  $ study_accession       : chr [1:7136] "ScheetzInternal" "ScheetzInternal" "ScheetzInternal" "ScheetzInt
ernal" ...
##  $ Cohort                : chr [1:7136] "Eye" "Eye" "Eye" "Eye" ...
##  $ Tissue                : chr [1:7136] "Choroid" "RPE" "Choroid" "RPE" ...
##  $ Age                   : chr [1:7136] "Adult" "Adult" "Adult" "Adult" ...
##  $ Sub_Tissue            : chr [1:7136] NA NA NA NA ...
##  $ Source                : chr [1:7136] "Native" "Native" "Native" "Native" ...
##  $ Source_details        : chr [1:7136] NA NA NA NA ...
##  $ study_title           : chr [1:7136] "Scheetz internal dissected clearn tissue" "Scheetz internal diss
ected clearn tissue" "Scheetz internal dissected clearn tissue" "Scheetz internal dissected clearn tissue" .
..
##  $ study_abstract        : chr [1:7136] "Scheetz internal dissected clearn tissue" "Scheetz internal diss
ected clearn tissue" "Scheetz internal dissected clearn tissue" "Scheetz internal dissected clearn tissue" .
..
##  $ sample_title          : chr [1:7136] "donor_1_CHO" "donor_1_RPE" "donor_2_CHO" "donor_2_RPE" ...
##  $ sample_attribute      : chr [1:7136] NA NA NA NA ...
##  $ data_location         : chr [1:7136] "local" "local" "local" "local" ...
##  $ region                : chr [1:7136] NA NA NA NA ...
##  $ Library_Notes         : logi [1:7136] NA NA NA NA NA NA ...
##  $ Comment               : logi [1:7136] NA NA NA NA NA NA ...
##  $ Origin                : chr [1:7136] NA NA NA NA ...
##  $ Age_Days              : chr [1:7136] NA NA NA NA ...
##  $ Sample_comment        : chr [1:7136] NA NA NA NA ...
##  $ geo                   : logi [1:7136] NA NA NA NA NA NA ...
##  $ gtex_accession        : chr [1:7136] NA NA NA NA ...
##  $ gtex_sra_run_accession: chr [1:7136] NA NA NA NA ...
##  $ Perturbation          : chr [1:7136] NA NA NA NA ...
##  $ Age_Years             : num [1:7136] NA NA NA NA NA NA NA NA NA NA ...
##  $ Sex                   : chr [1:7136] NA NA NA NA ...
##  $ Sex_ML                : chr [1:7136] "Male" "Male" "Female" "Female" ...
##  $ Sex_Score             : num [1:7136] 0.835 1.013 0.987 0.807 0.938 ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   BioSample = col_character(),
##   ..   run_accession = col_character(),
##   ..   sample_accession = col_character(),
##   ..   study_accession = col_character(),
##   ..   Cohort = col_character(),
##   ..   Tissue = col_character(),
##   ..   Age = col_character(),
##   ..   Sub_Tissue = col_character(),
##   ..   Source = col_character(),
##   ..   Source_details = col_character(),
##   ..   study_title = col_character(),
##   ..   study_abstract = col_character(),
##   ..   sample_title = col_character(),
##   ..   sample_attribute = col_character(),
##   ..   data_location = col_character(),
##   ..   region = col_character(),
##   ..   Library_Notes = col_logical(),
##   ..   Comment = col_logical(),
##   ..   Origin = col_character(),
##   ..   Age_Days = col_character(),
##   ..   Sample_comment = col_character(),
##   ..   geo = col_logical(),
##   ..   gtex_accession = col_character(),
##   ..   gtex_sra_run_accession = col_character(),
##   ..   Perturbation = col_character(),
##   ..   Age_Years = col_double(),
##   ..   Sex = col_character(),
##   ..   Sex_ML = col_character(),
##   ..   Sex_Score = col_double()
##   .. )
##  - attr(*, "problems")=<externalptr>
```

An additional method for examining the structure of eiad is to use the `summary` function. This function can be used on various objects in R. For data frames, `summary` yields a numeric, tabular, or descriptive summary of each column. Numeric or integer columns are

described by the descriptive statistics (quartiles and mean), and character columns by its length, class, and mode.

```
summary(eiad)
```

```
##   BioSample         run_accession      sample_accession   study_accession
##  Length:7136        Length:7136        Length:7136        Length:7136
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##     Cohort             Tissue              Age             Sub_Tissue
##  Length:7136        Length:7136        Length:7136        Length:7136
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##     Source           Source_details     study_title        study_abstract
##  Length:7136        Length:7136        Length:7136        Length:7136
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##  sample_title       sample_attribute   data_location         region
##  Length:7136        Length:7136        Length:7136        Length:7136
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##  Library_Notes  Comment            Origin            Age_Days
##  Mode:logical   Mode:logical   Length:7136        Length:7136
##  NA's:7136      NA's:7136      Class :character   Class :character
##                                Mode  :character   Mode  :character
##
##
##
##
##  Sample_comment       geo          gtex_accession     gtex_sra_run_accession
##  Length:7136        Mode:logical   Length:7136        Length:7136
##  Class :character   NA's:7136      Class :character   Class :character
##  Mode  :character                  Mode  :character   Mode  :character
##
##
##
##
##  Perturbation         Age_Years         Sex               Sex_ML
##  Length:7136        Min.   : 16.0   Length:7136        Length:7136
##  Class :character   1st Qu.: 68.0   Class :character   Class :character
##  Mode  :character   Median : 78.0   Mode  :character   Mode  :character
##                     Mean   : 75.5
##                     3rd Qu.: 86.0
##                     Max.   :107.0
##                     NA's   :6517
##    Sex_Score
##  Min.   :0.5040
##  1st Qu.:0.7957
##  Median :0.8848
##  Mean   :0.8765
##  3rd Qu.:0.9674
##  Max.   :1.1949
##
```

Along with the `str` and `summary` functions, we can examine individual columns of the data frame with our `typeof` function:

```
typeof(eiad$sample_accession)
```

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

```
typeof(eiad$Sex_Score)
```

```
## [1] "double"
```

```
str(eiad$Sex_Score)
```

```
##  num [1:7136] 0.835 1.013 0.987 0.807 0.938 ...
```

We can also interrogate the data frame for information about its dimensions; remembering that `str(gapminder)` said there were 1704 observations of 6 variables in gapminder, what do you think the following will produce, and why?

```
length(eiad)
```

```
## [1] 29
```

A fair guess would have been to say that the length of a data frame would be the number of rows it has (LOTS), but this is not the case; remember, a data frame is a *list of vectors and factors*:

```
typeof(eiad)
```

```
## [1] "list"
```

When `length` gave us 29, it's because eiad is built out of a list of 29 columns. To get the number of rows and columns in our dataset, try:

```
nrow(eiad)
```

```
## [1] 7136
```

```
ncol(eiad)
```

```
## [1] 29
```

Or, both at once:

```
dim(eiad)
```

```
## [1] 7136   29
```

We'll also likely want to know what the titles of all the columns are, so we can ask for them later:

```
colnames(eiad)
```

```
##  [1] "BioSample"             "run_accession"         "sample_accession"
##  [4] "study_accession"       "Cohort"                "Tissue"
##  [7] "Age"                   "Sub_Tissue"            "Source"
## [10] "Source_details"        "study_title"           "study_abstract"
## [13] "sample_title"          "sample_attribute"      "data_location"
## [16] "region"                "Library_Notes"         "Comment"
## [19] "Origin"                "Age_Days"              "Sample_comment"
## [22] "geo"                   "gtex_accession"        "gtex_sra_run_accession"
## [25] "Perturbation"          "Age_Years"             "Sex"
## [28] "Sex_ML"                "Sex_Score"
```

At this stage, it's important to ask ourselves if the structure R is reporting matches our intuition or expectations; do the basic data types reported for each column make sense? If not, we need to sort any problems out now before they turn into bad surprises down the road, using what we've learned about how R interprets data, and the importance of *strict consistency* in how we record our data.

Once we're happy that the data types and structures seem reasonable, it's time to start digging into our data proper. Check out the first

few lines:

```
head(eiad)
```

```
## # A tibble: 6 × 29
##   BioSample   run_accession sample_accession study_accession Cohort Tissue Age
##   <chr>       <chr>         <chr>            <chr>           <chr>  <chr>  <chr>
## 1 donor_1_CHO donor_1_CHO   donor_1_CHO      ScheetzInternal Eye    Choro… Adult
## 2 donor_1_RPE donor_1_RPE   donor_1_RPE      ScheetzInternal Eye    RPE    Adult
## 3 donor_2_CHO donor_2_CHO   donor_2_CHO      ScheetzInternal Eye    Choro… Adult
## 4 donor_2_RPE donor_2_RPE   donor_2_RPE      ScheetzInternal Eye    RPE    Adult
## 5 SAMN265651… SRR18292952   SRS12239005      SRP363470       Eye    RPE    <NA>
## 6 SAMN265651… SRR18292953   SRS12239004      SRP363470       Eye    RPE    <NA>
## # ℹ 22 more variables: Sub_Tissue <chr>, Source <chr>, Source_details <chr>,
## #   study_title <chr>, study_abstract <chr>, sample_title <chr>,
## #   sample_attribute <chr>, data_location <chr>, region <chr>,
## #   Library_Notes <lgl>, Comment <lgl>, Origin <chr>, Age_Days <chr>,
## #   Sample_comment <chr>, geo <lgl>, gtex_accession <chr>,
## #   gtex_sra_run_accession <chr>, Perturbation <chr>, Age_Years <dbl>,
## #   Sex <chr>, Sex_ML <chr>, Sex_Score <dbl>
```

Challenge 2

It's good practice to also check the last few lines of your data and some in the middle. How would you do this?

Searching for ones specifically in the middle isn't too hard, but we could ask for a few lines at random. How would you code this?

Solution to Challenge 2

To check the last few lines it's relatively simple as R already has a function for this:

```
tail(gapminder)
tail(gapminder, n = 15)
```

What about a few arbitrary rows just in case something is odd in the middle?

Tip: There are several ways to achieve this.

The solution here presents one form of using nested functions, i.e. a function passed as an argument to another function. This might sound like a new concept, but you are already using it! Remember my_dataframe[rows, cols] will print to screen your data frame with the number of rows and columns you asked for (although you might have asked for a range or named columns for example). How would you get the last row if you don't know how many rows your data frame has? R has a function for this. What about getting a (pseudorandom) sample? R also has a function for this.

```
eiad[sample(nrow(eiad), 5), ]
```

To make sure our analysis is reproducible, we should put the code into a script file so we can come back to it later.

::: challenge

Challenge 3

Go to file -> new file -> R script, and write an R script to load in the gapminder dataset. Put it in the `scripts/` directory and add it to version control.

Run the script using the `source` function, using the file path as its argument (or by pressing the "source" button in RStudio).