Accessing Data

Datasets Available in Base R

Some datasets are easily available in R and can be accessed with the data function.

```
data(rivers)
```

```
rivers[1:5]
```

```
[1] 735 320 325 392 524
```

We can get a list of all available datasets by entering data() into the console.

Note: Base R refers to R with no packages loaded.

Using the `data` Function

When we first `data(rivers)`, `R` creates one or more objects that have value `<Promise>`.

Once we have actually used the objects, the contents become available in the environment.

Try it out to see!

`data` Examples

- `islands`
- `volcano`
- `WorldPhones`

What types of objects are contained in this data?

The `data` Function Can Create Multiple Objects

How many and what types of objects are contained in the `euro` data?

The Data Frame

```
data(stackloss)
```

```
str(stackloss)
```

```
'data.frame':   21 obs. of  4 variables:
 $ Air.Flow  : num  80 80 75 62 62 62 62 62 58 58 ...
 $ Water.Temp: num  27 27 25 24 22 23 24 24 23 18 ...
 $ Acid.Conc.: num  89 88 90 87 87 87 93 93 87 80 ...
 $ stack.loss: num  42 37 37 28 18 18 19 20 15 14 ...
```

```
str(stack.x)
```

```
 num [1:21, 1:3] 80 80 75 62 62 62 62 62 58 58 ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:3] "Air.Flow" "Water.Temp" "Acid.Conc."
```

```
str(stack.loss)
```

```
 num [1:21] 42 37 37 28 18 18 19 20 15 14 ...
```

Data Frames are Like Matrices

For the data frame `stackloss`:

- Find number of rows
- Find number of columns
- Find column means
- Find column standard deviations
- Extract the first 3 columns
- Extract the last row

Data Frames are Not Like Matrices

We cannot do linear algebra on data frames without transforming them.

```
rep(1/nrow(stackloss), nrow(stackloss))%*%stackloss
```

Data frames can contain columns of different modes.

Columns of data frames can be accessed using $.

```
stackloss$stack.loss
```

Using $ creates a vector - we can confirm.

```
all.equal(stackloss$stack.loss, stack.loss)
```

Data Frames are Not Like Matrices

You can grow a data frame by adding columns.

You can create a new variable by assigning a vector with the same number of elements as the data frame has rows to the data frame using $ and the name of the new variable.

For example, we can create a new variable in the `stackloss` dataframe called `variable` that contains the order that each observation appears in.

```
stackloss$variable <- 1:nrow(stackloss)
```

You can call the new variable whatever you want, it doesn't have to be `variable`.

Can we turn a data frame back into a matrix?

Sometimes!

```
rep(1/nrow(stackloss), nrow(stackloss))%*%
  as.matrix(stackloss)
```

```
     Air.Flow Water.Temp Acid.Conc. stack.loss
[1,] 60.42857   21.09524   86.28571   17.52381
```

This will only work when the data frame does not contain characters or factors (which are a special way of storing characters that we're about to learn about).

How do we make a data frame?

We can construct a data frame by combining matrices with n rows and vectors with n elements.

```
sl <- data.frame(stack.x, stack.loss)
```

We can confirm they are the same!

```
all.equal(sl, stackloss)
```

Data Frames with Non-Numeric Data

Again, data frames can contain columns of different modes.

```
data(chickwts)
```

```
head(chickwts)
```

```
  weight      feed
1    179 horsebean
2    160 horsebean
3    136 horsebean
4    227 horsebean
5    217 horsebean
6    168 horsebean
```

You can also look at a data frame as if it is a spreadsheet in R by using the View function. Try it!

```
View(chickwts)
```

Factors

What appears to be a character vector is actually stored as a "factor."

```
str(chickwts)
```

```
'data.frame':   71 obs. of  2 variables:
 $ weight: num  179 160 136 227 217 168 108 124 143 140 ...
 $ feed  : Factor w/ 6 levels "casein","horsebean",..: 2 2
```

We will talk about this more in a bit.

Datasets Available in Packages

Different packages contain additional datasets, often for use as a demonstration of certain functions.

There are also some `R` packages that exist for the sole purpose of helping you load in specific datasets from online sources, e.g. `yahoofinancer` for downloading Yahoo Finance data.

Loading in Datasets

In real life, we probably want to load in data that's not already available in base `R` or some `R` package.

The most common format is a `.csv` file, where `.csv` stands for "commma separated value." These are often directly available or indirectly available, e.g. as an option for saving an spreadsheet in Excel.

The downside of `.csv` files is that they can be a bit big - we'll talk about dealing with big files soon.

Downloading a .csv file

Let's download files from a recent Evolution paper that have been made available:
https://datadryad.org/dataset/doi:10.5061/dryad.rs610.

The paper is available here, if you're curious:
https://academic.oup.com/evolut/article/69/10/2662/6851963.

Filepaths

In order to read in data, you need to figure out where the data is.

In `R`, you can identify the current file path using `getwd()`.

```
getwd()
```

You can see what files are there using `list.files()`.

```
list.files()
```

You can also set a new working directory using the `setwd` function and providing a file path.

Filepath Help

If you're having a hard time finding your file path, you can load in a `.csv` file by going to the "File" menu, navigating down to "Import Dataset" and clicking "From Text (base)." Then find your `.csv` file.

Once you do this, a line of code that starts with `read.csv` will get sent to your console. Save it! It will include the path to your file.

Reading in a .csv file

The `read.csv` function reads a `.csv` file into `R`, creating a data frame.

```
data <- read.csv("~/Downloads/seawater.archive.data.csv")
```

```
str(data)
```

```
'data.frame':   63 obs. of  9 variables:
 $ Plate                : int  1 1 1 1 1 1 2 2 2 2 ...
 $ Well.Name            : chr  "B2" "C2" "D2" "E2" ...
 $ Line                 : chr  "31" "22" "52" "43" ...
 $ Concentration        : chr  "seawater" "seawater" "seawater" "seawate
 $ Replicate            : int  1 1 1 1 1 1 2 2 2 2 ...
 $ History              : chr  "salt" "salt" "dark" "dark" ...
 $ initial.cell.density: num  41748 54755 15594 224336 15070 ...
 $ final.cell.density   : num  1119 10315 874 3077 455 ...
 $ rate.increase        : num  -1.688 0.262 -0.95 -2.358 -1.57 ...
```

Characters versus Factors

```
data <- read.csv("~/Downloads/seawater.archive.data.csv",
                 stringsAsFactors = TRUE)
```

```
str(data)
```

```
'data.frame':   63 obs. of  9 variables:
 $ Plate                 : int  1 1 1 1 1 1 2 2 2 2 ...
 $ Well.Name             : Factor w/ 6 levels "B2","C2","D2",..: 1 2 3 4
 $ Line                  : Factor w/ 33 levels "20","21","22",..: 13 3 26
 $ Concentration         : Factor w/ 1 level "seawater": 1 1 1 1 1 1 1 1
 $ Replicate             : int  1 1 1 1 1 1 2 2 2 2 ...
 $ History               : Factor w/ 4 levels "dark","marine",..: 3 3 1 1
 $ initial.cell.density: num  41748 54755 15594 224336 15070 ...
 $ final.cell.density    : num  1119 10315 874 3077 455 ...
 $ rate.increase         : num  -1.688 0.262 -0.95 -2.358 -1.57 ...
```

What the heck is a factor??

Factors are a mode that we haven't talked about yet. They can be thought of as fancy vectors.

Factors are a way of storing elements as positive integers with each integer value associated with a character label. The character labels are called "levels."

Generally, factors are annoying. However a nice thing about them is that they can clearly convey the total set of values that a variable could take on, even if certain values are not observed in the data. They are also sometimes convenient for plotting, summarizing, and analyzing data.

An Example of a Factor

The variable `History` is treated as a level when we specify `stringsAsFactors = TRUE`.

```
levels(data$History)
```

```
[1] "dark"   "marine" "salt"   "wild"
```

```
unclass(data$History)
```

```
 [1] 3 3 1 1 1 1 1 1 1 1 3 3 1 1 1 1 1 1 3 3 3 3 1 1 1 1 1
[39] 1 1 3 3 3 3 3 1 1 1 1 1 1 3 3 3 3 3 1 2 1 4 3 3 2
attr(,"levels")
[1] "dark"   "marine" "salt"   "wild"
```

```
as.numeric(data$History)
```

```
 [1] 3 3 1 1 1 1 1 1 1 1 3 3 1 1 1 1 1 1 3 3 3 3 1 1 1 1 1
[39] 1 1 3 3 3 3 3 1 1 1 1 1 1 3 3 3 3 3 1 2 1 4 3 3 2
```

Converting a Factor to its Values

```
levels(data$History)[as.numeric(data$History)]
```

```
 [1] "salt"   "salt"   "dark"   "dark"   "dark"   "dark"
 [9] "dark"   "dark"   "salt"   "salt"   "dark"   "dark"
[17] "dark"   "dark"   "salt"   "salt"   "salt"   "salt"
[25] "dark"   "dark"   "dark"   "dark"   "salt"   "salt"
[33] "dark"   "dark"   "dark"   "dark"   "dark"   "dark"
[41] "salt"   "salt"   "salt"   "salt"   "salt"   "dark"
[49] "dark"   "dark"   "dark"   "salt"   "salt"   "salt"
[57] "dark"   "marine" "dark"   "wild"   "salt"   "salt"
```

Being Careful Converting Factors to Numeric Values

Sometimes, quantities that should be numeric are treated as factors. For instance, the `Seed` variable in the `Loblolly` data.

```
data(Loblolly)
```

If we want to convert a factor back to a number, we need to be careful about how we do it. Just applying `as.numeric` returns the integers associated with each level. That's not what we want!

```
head(as.numeric(Loblolly$Seed))
```

```
[1] 10 10 10 10 10 10
```

We want to make the labels themselves to numbers.

```
head(as.numeric(levels(Loblolly$Seed))[as.numeric(Loblolly$Seed)])
```

```
[1] 301 301 301 301 301 301
```

More Importing Data

We will now import some basketball data to give some more examples.

Go here: https://www.basketball-reference.com/teams/BOS/2024.html#all_per_minute_stats

Specifically, we'll focus on per 36 minute statistics.

You can download these as an Excel Workbook or a `.csv`.

Try both approaches and import the data. If you download the Excel Workbook version, open it up and then save it as a `.csv`. Then load it into `R`. Try it!

.csv Examples

On my teaching page, I have posted four examples of `.csv` files that will require a bit more care to read in.

- `sportsref_download_1.csv`
- `sportsref_download_2.csv`
- `sportsref_download_3.csv`
- `sportsref_download_4.csv`

One at a time, let's try importing them using `read.csv`.

Note - you can read in a `.csv` file from the internet!

```
data <-
  read.csv("https://.../sportsref_download_1.csv")
```

(You need the complete link though!)

Other Formats - Other Delimiters

A `.csv` file is a plain text file where each line corresponds to a new row and each element within a row is **delimited** or separated by a comma.

There are other types of delimiters that are sometimes used, e.g. tabs and spaces.

`read.csv` versus `read.delim`, etc.

The `read.delim` function is more general - it does not assume that elements within a row are **delimited** (i.e. separated by) commas, like `read.csv`.

It tries to be smart and guess the delimiter being used, but sometimes it has trouble and requires specification of a specific delimiter using the `sep` argument, which takes on many values including:

- `"\t"` for tab delimited data
- `""` for white space delimited data
- `";"` for semicolon delimited data

Other Dedicated `read.` Functions

There are also several dedicated functions like `read.csv` that apply for other delimiters,

- `read.table` for white space delimited data
- `read.csv2` for semicolon delimited data where commas are used to denote decimals
- `read.delim2` for tab delimited data where commas are used to denote decimals

(Commas are sometimes used to denote decimals in other countries.)

More Data Examples

On my teaching page, I have also posted four examples of other files that we can read in:

- `sportsref_download_5.txt`
- `sportsref_download_6.tsv`
- `sportsref_download_7.txt`
- `sportsref_download_8.txt`

Using `read.delim` or other related functions, you can read in these alternative versions of the same data we've been working with. Try it!

Other Formats - `xls`

Sometimes, it may be better to avoid converting an `.xls` file to `.csv` before reading it in.

There are ways to do this! But they require additional packages and (in my experience) can be unstable across years of updates to Excel and `R`. This is why I taught you the brute force convert to `.csv` way.

I recommend the `read_excel` function in the `readxl` package for functions to read in Excel files.

Using `readxl`

On my teaching page, I have posted an example `.xls` file that is related to the data we've been working with.

Download it and load `readxl`!

```
library(readxl)
data <- read_excel("~/Downloads/sportsref_download.xls")
```

There are two primary advantages to directly reading in files from Excel:

- Fewer opportunities for things to make mistakes
- You can access multiple sheets from one file

Sheets Using `readxl`

By default, `read_excel` just reads in the first sheet.

You can check how many sheets and what their names are using the `excel_sheets` or by manually inspecting the workbook.

```
excel_sheets("~/Downloads/sportsref_download.xls")
```

Then you can change the `sheet` argument of `read_excel` to read in whichever sheet you like. Try it!

What the heck is a tibble?

Data read in using `read_excel` is stored as a tibble.

A tibble is an alternative to a data frame. For our purposes, there's no need to work with tibbles. You can convert a tibble to a data frame easily using `as.data.frame`.

```
data <- as.data.frame(data)
```

On Google Sheets

You need to download them anyway, just download them as `.csv` files, one per sheet.

Other Formats - General

The `foreign` package includes functions to read in:

- `.dta` files from Stata
- `.spss` files from SPSS

We won't do this here but you may need it in your future!

Other Formats - Compressed

Some of the formats we saw described in the `foreign` package are compressed, e.g. `.dta`.

This means that they are made smaller than they would be if stored as `.csv` files.

The file will be easier to move around and keep on your computer, but once it's in `R` it still takes up the same amount of memory because it is all read into memory at once.

Other Formats - Compressed

Many compressed file types are software specific, e.g. `.dta` files are specific to Stata. `R` has two compressed data file types, which differ in terms of how much they can store:

- `.RDS` files, which are read in with `readRDS`
- `.RData` files, which are read in with `load`

A single `.RDS` file can store one object, whereas a single `.RData` file can store multiple objects.

Other Compressed File Types

Working in teams that use multiple programming languages is common, and for this reason compressed file types that are not software specific are popular.

Two popular file types are part of the Apache Arrow project, which has an associated `R` package `arrow`.

They are:

- `.feather` files, which are read in with `read_feather`
- `.parquet` files, which are read in with `read_parquet`

Faster Ways to Load `.csv` Files

If you don't want to compress your data but do want to get it into `R` faster, there are at least two functions:

- `read_csv` in the `readr` package
- `fread` in the `data.table` package
- `read_csv_arrow` in the `arrow` package

We did not focus on these functions in this class. Rather, we focused on functions that are available in base `R` because functions that are available via packages are more likely to become obsolete, cease to be maintained, or run into issues working across platforms or operating systems.

However, they're great options as long as they work!

A Data Analysis Reminder

Some of these storage formats we have talked about let you edit data outside of `R`. For instance, you could open `sportsref_download.xls` in Excel, modify the data, and then work in `R`.

DO NOT DO THIS!!!

Changes made in Excel or directly to raw data files are not documented. No record of how you modified the data is created and stored.

In general, **all** manipulations of data should be done in `R` and only **raw** unmodified data should be stored.